Algorithmic Analysis of Name-Bounded Programs

From Java programs to Petri Nets via $\pi$-calculus

Matteo Settenvini

Supervisors:
Reiner Hüchting$^1$  Roland Meyer$^1$  Mikael Svahnberg$^2$

This thesis is presented as part of Degree of European Master in Software Engineering

$^1$Technische Universität Kaiserslautern
$^2$Blekinge Tekniska Högskola

School of Computing
Blekinge Institute of Technology
SE-371 79 Karlskrona
Sweden
This thesis is submitted to the School of Computing at Blekinge Institute of Technology in partial fulfillment of the requirements for the degree of Master of Science in Software Engineering. The thesis is equivalent to 20 weeks of full time studies.

Contact Information:
Author: Matteo Settenvini (840428P431)
E-mail: mcse10@student.bth.se

University advisor(s):
Prof. Mikael Svahnberg
School of Computing

School of Computing
Blekinge Institute of Technology
SE-371 79 Karlskrona
Sweden

Internet : www.bth.se/com
Phone : +46 455 38 50 00
Fax : +46 455 38 50 57

All contents of this opera, unless noted otherwise, are made available under the “Attribution-ShareAlike 3.0 Unported (CC BY-SA 3.0)” license, which is included in full in Appendix A.

Typeset in \LaTeX.
Abstract

Context. Name-bounded analysis is a type of static analysis that allows us to take a concurrent program, abstract away from it, and check for some interesting properties, such as deadlock-freedom, or watching the propagation of variables across different components or layers of the system.

Objectives. In this study we investigate the difficulties of giving a representation of computer programs in a name-bounded variation of \( \pi \)-calculus.

Methods. A preliminary literature review is conducted to assess the presence (or lack thereof) of other successful translations from real-world programming languages to \( \pi \)-calculus, as well for the presence of relevant prior art in the modelling of concurrent systems.

Results. This thesis gives a novel translation going from a relevant subset of the Java programming language, to its corresponding name-bounded \( \pi \)-calculus equivalent. In particular, the strengths of our translation are being able to dispose of names representing inactive objects when there are no circular references, and a transparent handling of polymorphism and dynamic method resolution. The resulting processes can then be further transformed into their Petri-Net representation, enabling us to check for important properties, such as reachability and coverability of program states.

Conclusions. We conclude that some important properties that are not, in general, easy to check for concurrent programs, can be in fact be feasibly determined by giving a more constrained model in \( \pi \)-calculus first, and as Petri Nets afterwards.

Keywords: pi-calculus, static analysis, concurrency, petri nets, name boundedness.
Contents

1 Introduction .............................................................. 1
  1.1 Static verification and Software Engineering ................. 2
  1.2 Background .......................................................... 5
    1.2.1 About the π-calculus .................................. 5
    1.2.2 Petri Nets .................................................. 13
  1.3 Preliminary literature review .................................... 13
    1.3.1 Review questions ......................................... 13
    1.3.2 Review methods ........................................... 14
    1.3.3 Discussion .................................................. 14
  1.4 Getting the code .................................................. 16
2 A Translation for Java Programs .................................... 17
  2.1 Useful tools ......................................................... 19
  2.2 Handling objects and classes .................................. 20
    2.2.1 Object instantiation .................................... 21
    2.2.2 Reference counting and object disposal ............... 23
    2.2.3 Field and method resolution ............................ 31
    2.2.4 Object locking ............................................. 37
    2.2.5 The null object ........................................... 43
  2.3 Control structures and program flow .......................... 44
    2.3.1 Method implementation ................................. 44
    2.3.2 Assignment ................................................ 45
    2.3.3 Branching and loops .................................... 46
    2.3.4 Exception handling ...................................... 49
    2.3.5 Spawning threads ....................................... 50
    2.3.6 Synchronization .......................................... 50
  2.4 The starting agent ................................................ 50
  2.5 An example of translation ....................................... 52
3 From π-calculus to Petri Nets ................................... 59
4 Conclusions and Further Research ................................. 69
<table>
<thead>
<tr>
<th>CONTENTS</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1 Conclusions</td>
<td>69</td>
</tr>
<tr>
<td>4.2 Ideas for Further Research</td>
<td>69</td>
</tr>
<tr>
<td>A License</td>
<td>71</td>
</tr>
<tr>
<td>⇒ Bibliography</td>
<td>81</td>
</tr>
<tr>
<td>⇒ Index</td>
<td>85</td>
</tr>
</tbody>
</table>
Introduction

We may always depend on it that algebra, which cannot be translated into good English and sound common sense, is bad algebra.

— William Kingdon Clifford, *4 May 1845 – † 3 March 1879

The $\pi$-calculus is a tool for modelling concurrent systems, and a formalism part of the family of process calculi. It has a simple yet expressive grammar, making it a good candidate to reason about inherently concurrent behaviours, such as security protocols and multi-threaded applications. Its strength lies in the capability of dynamically establishing and terminating communication links at runtime.

A few variations of $\pi$-calculus have been proposed. We are interested in a somehow “restrictive” version which, being less powerful than others, allows us to successfully reason about some important properties found in concurrent programs, such as deadlock-freedom. These properties can be checked by looking at coverability or reachability of markings in Petri Nets, for which – under certain assumptions that we will see later on – an automated and name-bounded translation from $\pi$-calculus exists.

In particular, putting a bound $b \in \mathbb{N}$ on the number of restricted names for all reachable processes, models quite naturally what happens in a concrete system. It simulates closely the abstraction that a certain system is constrained to use only a finite number of resources during any instant of its execution. It might be the number of active connections in a networked system, or the number of objects allocated on the heap in another.

The contribution that our work adds to the original theory of name-boundedness presented in [P4], is of a concrete translation from a real-world programming language – Java – to its corresponding name-bounded, $\pi$-calculus-based version. This translation can be used to perform static verification on an important subset of existing programs.

This thesis is structured as follows. The next section will contextualize the problem of performing static verification of concurrent programs from a software engineering standpoint.

In the subsequent sections we will then introduce the tools we need to proceed
with our translation from (a subset of) the Java programming language to its correspondent \( \pi \)-calculus representation.

Section 1.2 will present the specific variation of \( \pi \)-calculus we use, why this is necessary, how structural congruence works, and why we want to enforce name-boundedness. We then proceed to introduce Petri Nets, and to briefly talk about how coverability and reachability can be checked with them.

We then present our findings in the existing literature, and relate them to our work. This is done in Section 1.3, where we present a series of review questions and attempt to find relevant papers that respond to them.

Our work is accompanied with an concrete implementation of these concepts. How to get access to the source code is given in Section 1.4.

We then move on to the main part of our thesis, which is getting a translation for a relevant subset of Java. We thus start with introducing a handful of useful tools in Section 2.1, and then translate both the code that defines classes, methods and instantiate objects (in Section 2.2), and the code which is found inside actual methods, with control structures and the normal program flow (in Section 2.3).

In Chapter 3 we proceed then to see how our \( \pi \)-calculus translation can be further transformed into Petri Nets. This allows it to be fed to a generic model checker for verifying many important properties, such as deadlock freedom.

A final summary of our conclusions, as well ideas for further research, is finally presented in Chapter 4.

1.1 Static verification and Software Engineering

Concurrent programming is becoming more and more pervasive in modern multi-programmed systems. Recent advances in hardware design and in the miniaturization of electronic dies have not been enough to keep up with the expectations set by the market demand and Moore’s law. The ongoing trend is to add several cores to new systems, so that the computational workload may be split across multiple processor units [P18]. To take advantage of these features to their fullest extent, the programmer must also take action and adapt her code to a different programming model, which in turn means confronting oneself with problems such as concurrent access to shared resources, synchronization of non-atomic primitives, establishing and keeping small critical sections, and handling issues with deadlocks, fairness, and resource starvation [P3], [P11].

Additionally, in order to share as much physical resources as possible (for instance, memory caches), relaxed assumptions such as weak-memory ordering have been implemented in modern systems; ARM and PowerPC processors, much in use in embedded platforms, are examples of widespread architectures sporting such features. The price to pay for a potential increase in speed due to relaxed instruction ordering, is an additional burden for the programmer, which must now think also about out-of-order execution, and memory barriers [P11].
Unfortunately, the interleaving, and the non-deterministic order in which instructions are executed in a concurrent program, mean that the programs themselves are hard to debug, and problems difficult to catch and correct. It is often not necessary or enough to just add locks in the code, as they might in fact increase the possibility of deadlocks instead of resolving them [P9]. Specific concurrency bug categories ask for different actions.

Moreover, the brute-force approach to model checking, generating all possible configurations of a program and looking for issues, is almost always unfeasible, even for many of the simplest programs, as it quickly explodes factorially in size. For many programs, the state space might actually even be of infinite size [P1]. Therefore, a straightforward approach to static verification is bound to fail.

Alas, the inherent non-deterministic nature of concurrent programs often renders futile attempts of just running some tests, and checking the correctness of expected values at runtime. Only failures can tell us something interesting about our system (e.g. that a problem in fact exists). However, a matching expectation gives no guarantee about the correctness of all other possible program configuration; configurations that might depend on the input of the program, adding more complexity into the picture.

Hence, to ameliorate the issues we are confronted with, we would like to have more powerful tools which can perform static verification on concurrent programs, and thus give us a better degree of safety about their correctness.

Another promising avenue of research, when not employing static model checking, is represented by having the tester manually defining a set of invariants that are known to hold before and after entering a critical section, and then checking at runtime portions of the program for correctness through instruction tracing. This is the approach for instance taken by [P17], by the use of Linear Temporal Logic (LTL) expressions in Haskell code.

A further runtime approach which attempts to minimize the number of program state configuration that need to be examined, is contained in [P2], which checks directly Java bytecode. The main idea is to generate the trace for a run of the program, and then to predict possible alternative configurations which might lead to problems such as data races. The explosion in the state-space size to be explored is still considerable, so many times runtime and static verification approaches are used in concert to reduce the number of paths to be searched in the state graph [P1].

In this thesis we also attempt to bridge partially the gap, by providing a static verification method which partially emulates the runtime state of the program. In particular, we offer a way to keep track of references and object life-time. In a sense, the model we build is a virtual machine for simulating the steps taken by the program specified by its source code. We however take some shortcuts when needed; these and their motivation will be presented in later chapters. The task of giving a simulation relation which formally defines our approximation in the
The categories in which static or dynamic analysis of programs falls are summarized in [P1]:

- **Model-checking**: this is a static method consisting of verifying all reachable program states for the holding of some properties, starting from an abstract model of the system. It has mainly two drawbacks: the possibility that the actual implementation of the system will not match closely the model, and the need to define aforesaid formal model. When the model is too detailed, the distinction between the model and the actual implementation can blur; therefore, there is always some loss of information implied in the use of a model description.

- **Static analysis**: this method is based on deducing the properties of a program starting from its actual implementation, in source or binary form. This goes in the opposite direction of model checking; a strong point in favour of static analysis is being able to assert that the property tested match the actual implementation.

  Our own **pi-translate** tool and its underlying theory fall in this category. Applying the name-bounded theory, while excluding the possibility of an infinite state-space for name-bounded programs, can still lead to a related Petri-Net of non-primitive-recursive size. However, the resulting Petri-Net can be checked for coverability and reachability relatively easily [P4]. In the context of the building of the KM-Tree, it is possible to prepare a graph instead of a tree, thus allowing memoization of the generated states.

  We still need to approximate the number of reachable states. Under a set of assumptions, we give an overapproximation instead of an underapproximation. This means we might encounter a number of false positives in our analysis.

- **Non-deterministic testing**: as mentioned before, this method is a dynamic verification technique which just runs the same testsuite multiple times, with different inputs and introducing some random sleep periods between instructions in threads, in order to attempt to catch the maximum number of problems. However, no hard guarantees can be inferred from this method; the same configurations might be reached multiple times, and other never be considered.

- **Deterministic testing**: the idea in this dynamic technique, is to render each execution deterministic. This is achieved by explicitly specifying a series of schedules along with some inputs and expected results. The problem here is being able to determine the input and schedule pairs which might trigger a certain state which must be tested. This is often non-trivial, especially
for the programmer, and the help of a static-verification tool might still be desirable in order to generate a suite of test cases.

- **Behaviour-complete testing**: another dynamic method, it involves determining the reachable states for a certain input at runtime, and from them generating a reduced state-space which is limited to the constraints of the input given. This can somewhat be seen as an improvement over static-analysis, since it often renders the problem more tractable; however, in order to do so, it requires the programmer to manually specify which variables are shared, and add a set of assertions which can be tested for validity. For multi-threaded programs in many programming languages, including Java, threads share the whole memory space; therefore, one would have to assume all variables can be shared, with little to no benefit to the state-space size reduction.

We do not perform symbolic execution analysis \[P19\] during static analysis, since in our translation we do not keep track of the exact data flow of set variables, or do boolean predicate checking. The reasons are discussed in later chapters.

A static verification approach, over the other methods, has the benefit of detecting not only assert violations (and assertions can be hard to write, especially with pre-existing legacy systems), but also other issues such as data races, unwanted states, and deadlocks.

## 1.2 Background

The \(\pi\)-calculus was initially introduced in 1989 by Milner, Parrow and Walker \[P13\] and has by then been used successfully to model concurrent systems that can change their dynamic links at runtime. It can work at different levels of abstractions, and models have been proposed for things ranging from memory models for physical machines, to security protocols, to the interaction of software components at an architectural level.

In the following sections, we will present the needed formalisms about \(\pi\)-calculus that will enable us to introduce the theory of name-boundedness, as well to build a translation for a subset of Java programs.

### 1.2.1 About the \(\pi\)-calculus

In \(\pi\)-calculus, we have two basic operations: sending and receiving messages over channels. Both operations can be used as prefixes \(\pi\).

A send operation can react with a receive operation if they have the same channel and they are unguarded, e.g. if they are not preceded by any other prefix. A more precise definition of the reaction relation is given later on. After reacting, the receive and send operations are intended as consumed, and they are removed from the currently running process.
A name can be used as either a channel or a message: it does not matter. They are taken from the countable set of names $\mathcal{N}$.

We write the send operation of message $a$ over channel $b$ as $b\langle a \rangle$. Conversely, we write the receive operation of $a$ over $b$ as $b(a)$.

We also add the non-deterministic choice operator $\cdot$, and the parallel operator $\mid$. The choice operator allows any of its operands to run, but once one of the branch prefixes reacts, the other branches are ignored. The parallel operator instead allows any of its operands to react at any time, after it becomes unguarded.

Agents are process identifiers: names for processes, that are used to implement recursion. Agent names are written in uppercase (as in $\text{NAME}_A\text{ME}\text{E}$), and range over $\mathcal{K}$. An agent $\kappa$ can be defined through a defining equation such as $\kappa(\bar{x}) \overset{\text{def}}{=} P$, where $P$ is a process and $\bar{x}$ is a sequence of distinct names.

An agent can be invoked, for instance as $\kappa[\bar{a}]$, with a number of parameters that matches its definition $\kappa(\bar{x})$: $|\bar{a}| = |\bar{x}|$. In this case, a substitution from $\mathcal{N} \rightarrow \mathcal{N}$ happens in the process $P$ given in definition of $\kappa$, where the names in $\bar{a}$ “overwrites” the names $\bar{x}$, and as such is denoted with $P\{\bar{a}/\bar{x}\}$. It changes all names in $\bar{x}$ by the corresponding name in $\bar{a}$, after all bound names have been $\alpha$-converted so that there is no clash with the substituting names $\bar{a}$.

Processes $P$, $Q$, etc. from the set of processes $\mathcal{P}$ are defined by the following grammar:

$$
M ::= 0 \mid M + M \mid \pi.P \quad P ::= M \mid \kappa[\bar{a}] \mid P_1 \mid P_2 \mid \nu a. P
$$

We call processes $M$ and $\kappa[\bar{a}]$ sequential, as they are the operands in use for parallel composition. $\mathcal{S}$ is the set of all sequential processes.

Receive operations bind the name(s) they receive as a message for the rest of their sequential process. The same do restrictions $\nu a$, which are also prefixes, and are used to introduce new names. We will call these names either bound, or restricted\(^1\).

We use $\mathcal{R}$ for the set of names that are restricted (bound) in processes, and $\mathcal{R}(P)$ the names that are restricted in $P$. Those names that are not restricted, are considered free; they are part of the set $\mathcal{T}$, and $\mathcal{T}(P)$ is the set of free names in $P$.

### Structural congruence

The structural congruence is a relation which tells us when two processes can be considered to behave exactly the same. There are different definitions of structural congruence in the literature, depending on how much strict we want this relation to be. Different definitions lead to different properties and easiness to prove theorems connected to the $\pi$-calculus. Here, we settle for the following definition.

\(^1\)Rarely, also “private names”, in the sense that, unless distributed to other processes through send operations, they are visible only to the current one.
We denote the structural congruence with \( \equiv \), and we define it as the smallest relation \( \subseteq P \times P \), which is subject to these rules:

1. If \( P \) and \( Q \) are variants of \( \alpha \)-conversion, then \( P \equiv Q \).
2. We allow the choice (\( + \)) and parallel (\( | \)) operators to be both associative and commutative.
3. \( 0 \) is the neuter element for choice and parallel operations, so that for instance \( P | 0 \equiv P \).
4. Restrictions satisfy the following laws:
   \[
   \nu a.0 \equiv 0 \quad (1.2) \\
   \nu a.\nu b.P \equiv \nu b.\nu a.P \quad (1.3) \\
   \nu a.(P | Q) \equiv P | \nu a.Q \text{ if } a \notin \mathcal{F}(P) \quad (1.4)
   \]

Checking for structural congruence can be solved in different ways; one of them, and the one we implemented in our tool \( \pi \)-translate, is by building a term-equality (TE) tree, as described in \[P6\]. An example of such tree is shown in Figure 1.1.

In this sample TE-tree, we have nodes of different types: for restrictions (whose scope is the subtree under a \( \nu \) node), send (\( s \) nodes) and receive (\( r \) nodes) operations, names, and operators (choice, and parallel – not shown here –, as well as sequence). Non commutative operations (send and receive, and sequence operands) have outgoing edges marked with a natural number \( > 0 \), which indicates the right order of operands.

In our example, we are showing the intermediate form – that is, before coalescing the empty nodes and the restricted names they point to, and maximizing the restriction scope. A process is in normal form after the restriction scope is maximized \[P12\]: a) the parallel composition of sequential processes is already in standard form. b) if \( P \) is in standard form, then also \( \nu x.P \) is, given that \( x \in \mathcal{F}(P) \). We also remove useless restrictions \( \nu x.P \) that happen over names that are not free in their scope: \( x \notin \mathcal{F}(P) \). This is a consequence of equation 1.4 and 1.2, when \( Q \equiv 0 \), and because \( 0 \) is the neuter element for parallel composition. Once this is done, checking for structural congruence is just a matter of checking for graph isomorphism.

Initially, we tried implementing some graph isomorphism algorithms ourselves (the Boost libraries that we used did not provide an adequate algorithm ready for consumption, since it did not support matching also on edge properties), such as those presented in \[P10\]. However, in the end, we settled for the most naïve approach: just matching nodes through a depth-first search, exploring recursively all permutations of children for a certain node in search for subgraph-isomorphism.
Figure 1.1: Intermediate TE-tree representation for $\nu a. (\overline{a}(b).a(b)) . \nu b. (\overline{b}(a).b(a))$, as generated by $\pi$-translate
While this might strike as a poor choice, computationally speaking, in practice it is not as bad as it looks. We can check for equality between pairs of nodes or edge properties quite fast (including discriminating on the number of outgoing edges), and once the root of a subtree, during our descent, has been found to be different than our reference root in the second subtree, we can skip to looking for a match in the next permutation of the root and its siblings. This works well-enough since the TE-tree is a direct acyclic graph (DAG), with no disconnected components.

The reaction relation

We want to enable transitions among different states for our processes, and we need a relation to do so. Thus, we introduce the reaction relation as the smallest relation \( \rightarrow \subseteq \mathcal{P} \times \mathcal{P} \), that satisfies:

\[
\begin{align*}
\alpha(b).P + M \mid \overline{\alpha(c)}.Q + N & \rightarrow P \{c/b\} \mid Q & \kappa[\tilde{a}] \rightarrow P \{\tilde{a} / \tilde{x}\} \text{ with } \kappa(\tilde{x}) \overset{\text{def}}{=} P \\
\end{align*}
\]

and that is closed over restriction, parallel composition, and structural congruence. The transition system of a process, is given by the set of reachable processes modulo the structural congruence among them: \( \Gamma(P) := (\text{Reach}(P) / \equiv, \rightarrow, P) \), where \( P \rightarrow Q \) if and only if \( P \rightarrow Q \).

Name boundedness

A process \( P \) is said to be name bounded if and only if there is a bound on the number of restricted names that holds for all reachable processes in \( \text{Reach}(P) \) [P4].

We can also define the number of active restrictions, \( \text{arn} \), as:

\[
\begin{align*}
\text{arn}(S) & := 0 \quad (1.5) \\
\text{arn}(P \mid Q) & := \text{arn}(P) + \text{arn}(Q) \quad (1.6) \\
\text{arn}(\nu x. P) & := \begin{cases} 
1 + \text{arn}(P) & \text{if } a \in \mathcal{F}(P) \\
\text{arn}(P) & \text{otherwise} 
\end{cases} \quad (1.7)
\end{align*}
\]

**Definition 1.2.1.** A process \( P \in \mathcal{P} \) is \( b \)-name-bounded for a certain \( b \in \mathbb{N} \) if, \( \forall Q \in \text{Reach}(P) \), \( \text{arn}(Q) \leq b \). We say that process \( P \) is name-bounded, if it is \( b \)-name-bounded for some \( b \in \mathbb{N} \).

An example of a name-bounded process is \( \nu x. K_1[x] \), with \( K_1(x) \overset{\text{def}}{=} (K_1[a] \mid K_1[a]) \), because although it will recurse on itself, and spawn infinitely many new processes, it does not introduce new names. In fact, this process is even 1-name-bounded.

In contrast, process \( \nu x. K_2[x] \), with \( K_2(x) \overset{\text{def}}{=} \nu x. (K_2[x] \mid K_2[x]) \) is not name-bounded, since each new invocation to \( K_2 \) will increase the count of the active restricted names.
An interesting result from [P4] is that, while determining whether \( P \in \mathcal{P} \) is name-bounded is in general not decidable, checking whether a process \( P \in \mathcal{P} \) is \( b \)-name-bounded, given a fixed bound \( b \in \mathbb{N} \), instead is decidable.

### The reachability and coverability set

We say that a process \( Q \) is reachable from \( P \), if \( P \xrightarrow{} Q \), where \( \xrightarrow{} \) is the reflexive and transitive closure of \( \rightarrow \).

The **reachability set** of \( P \) is defined as the set of all processes reachable from \( P \), and we write it as \( \text{Reach}(P) \).

**Embedding** is defined as a quasi-ordering \( \preceq \subseteq \mathcal{P} \times \mathcal{P} \), which is the smallest relation for which \( \nu \bar{x}.P \preceq \nu \bar{x}.(Q \mid R) \) holds. Additionally, it has to be closed under structural congruence: \( P \equiv P' \preceq Q' \equiv Q \Rightarrow P \preceq Q \).

The downwards closure of a set of processes \( \mathcal{P}' \in \mathcal{P} \) is defined as: \( \mathcal{P}' \downarrow := \{Q \mid \exists P \in \mathcal{P}' : Q \preceq P \} \).

A process \( Q \) is **covered** from \( P \), if \( \exists R \in \text{Reach}(P) : Q \preceq R \). The coverability set of \( P \) is then defined as the set of all processes coverable from \( P \). In other words, if we can reach from \( P \) a process that embeds \( Q \), we have covered \( P \). If we extend this concept to get all the coverable processes, we get that the coverability set is the downward closure, with respect to the embedding order, of the reachability set (\( \text{Reach}(P) \downarrow \)).

### Identity-awareness

As we have seen, different reactions might be enabled at the same time: for instance, in the process \( (a(b) \mid \overline{a}(c) \mid \overline{a}(d)) \), two different reactions are possible. Depending on the one that fires, we reach different processes; thus, if we keep track of all possible reachable processes, it is only natural to represent these reactions with a tree.

When transitioning from a process to the next in such reachability tree, it is convenient keeping track of the identity of restricted names, so that we can better check if we are respecting the bound.

Thus, we rely on the notion of **identity-aware processes**, where restricted names \( \nu a \) are replaced by **instances** of the form \((a, i)\). The set of all instances is \( \mathcal{J} := \mathcal{R} \times \mathbb{N} \). These pairs are chosen so that the index \( i \in \mathbb{N} \) is the smallest index not present in the target process. Note how this permits the same tuple \((a, i)\) to potentially appear more than once in a reachability set, for different restricted names – these tuples can be thought of as being garbage collected, and being handed out again after being returned to a common pool. However, while a restricted name is in use, a tuple is bound to it.

We define the set of all identity-aware processes as \( \mathcal{P}_{ia} \), and we define the set of all instances in \( P_{ia} \) as \( \mathcal{J}(P_{ia}) := \mathcal{F}(P_{ia}) \cap \mathcal{J} \).
We also label transitions among identity-aware processes with newly generated instances, so that we can distinguish them from old ones. For example, for $K_1(x) \overset{df}{=} K_1[x]^2$, and invoked with $\nu a.K_1[a]$, we have the transitions:

$$K_1[(a,0)] \xrightarrow{\emptyset} K_1[(a,0)]^2 \xrightarrow{\emptyset} K_1[(a,0)]^3 \xrightarrow{\emptyset} \ldots \tag{1.8}$$

Instead, for $K_2(x) \overset{df}{=} \nu x. (K_2[x]^2)$, when invoked with $\nu a.K_2[a]$ we obtain the transitions:

$$K_2[(a,0)] \xrightarrow{(a,0)} K_2[(a,0)]^2 \xrightarrow{(a,1)} K_2[(a,1)]^2 \xrightarrow{(a,0)} \ldots \tag{1.9}$$

At this point, two different transitions are enabled. Either the first term gets invoked:

$$K_2[(a,1)]^2 \xrightarrow{(a,2)} K_2[(a,0)]^2 \xrightarrow{(a,1)} K_2[(a,1)]^2 \xrightarrow{(a,0)} \ldots \tag{1.10}$$

...or the last one (we “forget” the previous $(a,0)$ instance, since the formal parameter $x \notin \mathcal{F}(K_2)$):

$$K_2[(a,1)]^2 \xrightarrow{(a,0)} K_2[(a,0)]^2 \xrightarrow{(a,1)} K_2[(a,0)]^2 \ldots \tag{1.11}$$

Limit processes

If we take a closer look to Equation 1.8, we can already see where we are headed. The name $(a,0)$ will be known to arbitrarily many processes. We can thus accelerate this situation, and represent it with: $K_1[(a,0)]^\omega$, to represent the class of processes $K_1^j$ for an arbitrarily large $j$.

More precisely, we define a limit process as being either a sequential process $S$, a process of the form $L^\omega$, or the parallel composition $L_1 | L_2$ of limit processes [P4].

Extending structural congruence to cover also limit processes means adding the following rules:

$$S^\omega | S = S^\omega \quad S^\omega | S = S^\omega \quad (S^\omega)^\omega = S^\omega \quad (L_1 | L_2)^\omega = L_1^\omega | L_2^\omega$$

We call a limit process $L$ in standard form, if $L = S_1^{k_1} | \ldots | S_n^{k_n}$, where $S_i \neq S_j$ for $i \neq j$, and $k_i \in \mathbb{N} \cup \{\omega\}$. It is now possible to give an order over sequential processes grouping them over structurally congruent classes, joining them together.
Algorithm 1 Karp & Miller tree construction, as presented in \[P4\]

1: procedure KarpMiller\( (P_{ia}) \)
2: \( V := \{\text{root} : P_{ia}\}; \quad \rightarrow_{KM} := \emptyset; \quad \text{Work} := \text{root} : P_{ia}; \)
3: while Work not empty do
4: \( \text{Pop } n_1 : L_1 \text{ from Work}; \)
5: for all \( L_1 \rightarrow_{ia} L_2 \) up to \( \equiv \) do
6: if there is \( n : L \xrightarrow{KM} n_1 : L_1 \) such that \( L_2 \equiv L \mid L_{rem} \) and \( \mathcal{F}(L_{rem}) \cap \mathcal{F}(L, L_2) = \emptyset \) then
7: \( L_2 := L \mid L_{rem}^\omega; \)
8: end if
9: let \( m \) be a new node
10: \( V := V \cup \{m : L_2\}; \)
11: \( \rightarrow_{KM} := \rightarrow_{KM} \cup \{n_1 : L_1 \xrightarrow{\mathcal{F}(L_1, L_2)}_{KM} m : L_2\}; \)
12: \( \text{Work} := \text{Work} \cdot (m : L_2) \)
13: provided \( L_2 \) does not occur from root to \( n_1; \)
14: end for
15: end while
16: return \( (V, \rightarrow_{KM}, \text{root} : P_{ia}). \)
17: end procedure

Karp-Miller tree construction

We are now ready to build our tree representing reachable processes. For resemblance to the structure found in Petri Net coverability analysis, the name “Karp & Miller tree” has been proposed.

The procedure used to build the tree for identity aware processes, is shown in Algorithm 1. Starting from a root process \( P_{ia} \), a Work queue is built to allow the progressive exploration of reachable limit processes, and the set \( V \) of vertices is initialized with the root itself.

Then, the algorithm iteratively deepens the tree. For each vertex in the queue, representing a reachable limit process, all possible transitions are explored; unless the current process has already stabilized, and new reachable processes are structurally congruent to the current one.

The condition in the if statement allows accelerating a limit process only if the reachable process embeds the current one, and if the fresh instances introduced during the transition – \( \mathcal{F}(L, L_2) \) – are not already instances in \( L_{rem} \). This is needed to avoid accelerating those processes that will not distribute restricted names arbitrarily often.

An example of this can be seen in Equation 1.9. After the first transition takes place, we have that \( \kappa_2[(a, 0)] \preceq \kappa_2[(a, 0)]^2 \). However, \( \mathcal{F}(\kappa_2[(a, 0)]) \cap \{(a, 0)\} \neq \emptyset \), so we cannot accelerate \( \kappa_2[(a, 0)]^2 \) to \( \kappa_2[(a, 0)]^\omega \). This is in fact correct, since
a further transition shows how \((a,0)\) does not get distributed to arbitrarily many processes.

After that, the algorithm just updates the set of vertices, the set of edges representing known transitions, and the worklist.

[P4] goes further, and shows also that this approach is both sound and complete. Moreover, it notes how Algorithm 1 terminates with input \(P_{ia} \in \mathcal{P}_{ia}\), if and only if \(P_{ia}\) is name-bounded.

Of course, this has the practical implication that, if we produce a translation which is inherently not name-bounded, trying to construct the KM-tree will result in a program which will never terminate.

### 1.2.2 Petri Nets

Petri Nets are a useful formalism for representing concurrent systems. They are defined as tuples \((S, T, W, M_0)\), where \(S\) is a finite set of places, \(T\) is a finite set of transitions, \(W\) is a weight function from places to transitions and from transitions to places: \((S \times T) \cup (T \times S)\), and \(M_0\) is the initial marking, a function assigning a natural number to each place.

We require the set of places and of transitions to be disjoint. This leads effectively to a bipartite graph. Places are drawn as circles, and transitions as boxes. Markings, functions \(S \to \mathbb{N}\), are represented by circular tokens that are drawn within each place, or alternatively as a vector of values.

We say that a transition is enabled in a certain marking \(M\), if \(\forall s \in S, M(s) \geq W(s, t)\). If an enabled transition fires, we transition from the marking \(M\) to the marking \(M'\), so that \(M'(s) = M(s) + W(t, s) - W(s, t), \forall s \in S\). The firing relation is then denoted by \(\rightarrow : \rightarrow \subseteq \mathbb{N}^S \times T \times \mathbb{N}^S\).

### 1.3 Preliminary literature review

Following on the leads presented in the introduction, we would like to perform a systematic literature review upon the topic of translations from real-world programming languages to a \(\pi\)-calculus counterpart.

We thus identify a small set of research questions we would like to better explore, and follow a reproducible strategy so that our results can be validated (or, in alternative, reasonably confuted) by other interested third parties.

Specifically, in Section 1.3.1 we list our review questions; in Section 1.3.2 we explain the methodology followed to perform the review; and finally, in Section 1.3.3 we give an overview of all included papers, discussing the contributions they provide that support our thesis.

#### 1.3.1 Review questions

The leading questions that we are interested in answering are:
1. Are there other translations from Java to \( \pi \)-calculus? Are these name-bounded or not?

2. Are there translations for other languages into \( \pi \)-calculus?

3. How are recursive-locks represented in \( \pi \)-calculus? Is the solution name-bounded?

### 1.3.2 Review methods

To conduct this literature review, we employed roughly the guidelines which can be found in [P7]. We first identified the set of databases in Table 1.1, and performed a search on them, based on the queries presented in Table 1.2.

<table>
<thead>
<tr>
<th>ID</th>
<th>Search engine</th>
<th>Fields to search</th>
</tr>
</thead>
<tbody>
<tr>
<td>DB1</td>
<td>IEEE Xplore</td>
<td>Meta-data only</td>
</tr>
<tr>
<td>DB2</td>
<td>Inspec</td>
<td>Title, abstract, and keywords</td>
</tr>
<tr>
<td>DB3</td>
<td>Compendex</td>
<td>Title, abstract, and keywords</td>
</tr>
</tbody>
</table>

Table 1.1: Databases and fields searched

As can be seen in Table 1.2, we had to use a different, simpler query for IEEE Xplore, since it forces us to use a simplified search string. The query interface limits the number of search terms to 15.

ACM Digital Library’s interface is even more limiting, as it does not offer an “expert” interface where to input an expression with boolean operators. Thus, searching the ACM database was done through Compendex, which indexes all ACM transactions [W2].

It should be noted that, even after de-duplicating results among databases, some papers were listed more than once. Therefore, the true number of hits is slightly lower than the one stated in the table.

We took however care to have a look at the references listed in papers returned by this search. It helped us to discover more relevant articles, which we used in this thesis as secondary sources of information. All references are of course cited at relevant places in the text.

As for excluded studies, the number of results was small enough to be scanned through by hand, discarding those papers that were not matching our research questions.

### 1.3.3 Discussion

Literature does not offer many frameworks for translating from widely-used programming languages to \( \pi \)-calculus. In [P16], a translation is provided only for a
small, artificial language which does not offer many of the features known in modern programming languages, such as polymorphism, monitors, etc. Work seems to have been done mainly for the Erlang [P14] and Java [P5] programming languages.

The translation given in [P5], however, makes use of an internal state, encoding many entities – such as names for methods, classes, and the like – as integers. It usually leads to name-unbounded processes, since names representing references are not let go. Moreover, it employs a version of the $\pi$-calculus that makes use of the match and mismatch operators, something we cannot rely on (or the result would be difficult to translate into a Petri Net).

While working on our translation, it became apparent that the main place where these operators would be needed is in the implementation or recursive locks, because we need to check if the context attempting to acquire an already-taken lock is its current owner or not. Also a more specific analysis of locking than the language-wide one in [P5], found in [P15], shows the need for match/mismatch, in the form of the if-then-else construct.

We then explored if it was possible to add the match and mismatch operators as direct extensions of our $\pi$-calculus dialect, using only available primitives. It turns out it can be done by using a specific encoding that assigns a string of increasing length to each new name by a local generator [P8]. The problem is
that, in order to match/mismatch over a name \( n \), a process is spawned where \( n \) is bound. This would prevent us to reuse names in identity-aware processes. They would add to the bound, preventing us to accelerate some branches and leading us to an unbounded translation.

## 1.4 Getting the code

Our work is complemented by a concrete implementation to enable a user to translate Java programs into their name-bounded \( \pi \)-calculus representation.

Hence, we introduce here \( \pi \)-translate, a modular C++ program (and accompanying framework) to convert source code to its equivalent \( \pi \)-calculus representation. From there, it is possible to further process the resulting formulas. The default plugin will try to produce a name-bounded version and, on request, the accompanying Petri-Net representation.

The main goal is to enable the user to run a series of static verification tests on the original code, such testing for possible deadlocks, or whether a certain object can be seen by other unrelated components (something which could pose a security risk).

While still in the early stages of development, you can clone the main git repository with the following command:

```bash
git clone http://anonvcs.montecristossoftware.eu/pi-translate.git
```

Alternatively, were the main repository down for some reason or another, you can fetch its backup mirror with:

```bash
git clone git://gitorious.org/pi-translate/pi-translate.git
```

Installation requirements and instructions can be found in the “INSTALL.md” file, while the full license text is available in the “COPYING.md” file.

Patches, as always, are welcome! \( \pi \)-translate is released under the terms of the GNU General Public License v3 or, at your option, any other later version. It is, and always will be, \textit{libre software}. 

A Translation for Java Programs

A good translation should capture the essence of the original semantics, abstracting away those details we are not concerned about in our context. We want to retain information such as logical branching, and scoping of variables as performed by the original program. On the other hand, we can safely apply a series of simplifications, discarding e.g. things such as keeping track of how many times a loop repeats, or how exactly exception are thrown and matched. In particular, we are content to provide full path coverage of the source code in the translation. That is, running the translation should make possible to exercise at least the same code paths that would be walked by running any instance of the original program.

Java was chosen as the first language for which to offer a corresponding π-calculus translation, since it has a relatively easy-to-understand formal definition, complete with well-documented operational semantics. Moreover, some prior work in this direction already existed; not many “real world” programming languages, at the time of this writing, had been explored by other researchers for their π-calculus counterpart\(^1\). Also, Java relies heavily on the object-oriented paradigm which, as we will see, is particularly suited for a translation.

This work takes inspiration from the work done by [P16, part VII], and later expanded to cover the Java language in [P5]. However, we tailored the translation to our context, simplifying it at places, adding some constraints at others, so that the result can be modelled by name-bounded processes.

The reasons that brought us to develop a different translation from the one in [P5], are:

1. we needed to use a different set of π-calculus’s grammar rules so that our work is translatable to Petri Nets. In particular, we did not want to rely on

\(^1\)In particular, a translation for the functional programming language Erlang exists [P14].
the match and mismatch operators.

2. we want our whole translation is built with stateless processes, making easy to construct the corresponding Petri Net. The translation by Jacobs and Piessens instead relies on an `IdentityServerChannel` which returns subsequent natural numbers at each call, thus effectively acting as a counter.

3. we are not (at least at this time) interested in some fine-grained operations and facets modelled in the other translation, such as retaining accessibility or downcasting of object references.

4. we need to make sure that, under certain assumptions, our final translation is still name-bounded (if the original program of course makes use of a bounded number of resources). This has several implications, the most important one being that, since each created object is referred to with a new name, we need to make sure that they are “garbage collected” upon their last use.

In particular, our approach has the following limitations, mostly dictated by the theory of name-boundedness found in [P4]:

1. in the current form, we are not able to dispose of objects holding circular references. This is due to our simple reference-counting approach. Therefore, Java programs with an unbounded number of terminating threads, creating and disposing of objects with circular references, will produce an unbounded π-calculus representation.

While this might seem limiting, it still covers the modeling of a good set of practical problems. It is probably possible to implement some kind of mark-and-sweep garbage collection mechanism, by binding newly created objects also to contextes and using an object’s field as a marker, but this goes out of the scope of this thesis; the present approach should work well in many cases.

2. we do not provide semantics for reentrant locks, only for “normal” locks. Why this is less a problem than it would seem at a first glance, is discussed in more detail in Section 2.2.4.

3. we cannot determine how many times a loop has exactly to be run, since that would in many cases involve some sort of zero-test. We thus introduce a parameter given by the user to our translation, $\text{loop}_k$, which represents the maximum number of times any loop in the program can run. If this parameter is not high enough to render some problem observable, the user would need to increase it and run the generator for the translation once again. This is discussed in more detail in Section 2.3.3.
Additionally, it should be noted that we do not translate features from Java that are either used only at compile time, or that are not relevant because of the way we model dynamic method resolution. In particular, this does mean that we do not take in account both interfaces, generics’ type parameters, and downcasting (which we assume for the sake of our translation that will always succeed).

2.1 Useful tools

To make our translation simpler, we allow for a bit of syntactic sugar. In particular, the set $A$ being the set of known agent names, we define the notation $*P$ as a shorthand for:

$$Q \cdot *P \equiv Q \cdot \kappa[\vec{x}], \quad \kappa(\vec{x}) \overset{def}{=} (P).\kappa[\vec{x}], \quad A = A \cup \{\kappa\}$$

where $P ::= \pi.P \mid 0$, $\vec{x} = \mathcal{R}(Q) \cap \mathcal{F}(P)$, and initially $\kappa \notin A$. Here, $\cdot$ is any operator valid in $\pi$-calculus (choice, parallel composition, and the sequential operator). Note that this is not the definition of repetition (a.k.a. bang operator), because repetition uses parallel composition instead than the sequential operator. It is important to remember that this kind of syntax could quickly bring us to introduce unbounded processes if not used with care; we thus require that $\mathcal{R}(P) \cap \vec{x} = \emptyset$.

We also allow a shorthand for literal substitution of agent names inside the body of a macro. This helps us factor out some concepts to better focus on a higher-level picture. Since it can be carried out in a separate pre-processing stage, it does not change our results. Therefore, when we encounter a macro defined as: $#A_1(M_1, \ldots, M_n, \vec{x})$, and called with $#A_1[\kappa_1, \ldots, \kappa_n, \vec{x}]$, we read it as a new agent $A_2(\vec{x})$, with each occurrence of $M_i$ inside the body of $#A_1$ replaced by the corresponding agent name $\kappa_i$, $\forall i \leq n$. We have to be careful if we use recursive macros, so that their expansion will always terminate.

We also define a number of global free names that are necessary to our translation, as we need them to ensure correct semantics for some Java language features, such as exceptions handling and for catching abrupt termination of a program execution.

Firstly, we define the unit value as $\star$. This will be familiar to readers that have used functional languages such as ML. It is mostly employed for those functions whose return value is not interesting, and can be discarded. In Java, functions returning void will make use of $\star$ instead. In $\pi$-calculus, if we use unit for introducing a bound name in a prefix $\pi$ of $P$, it is as if we choose a new name $\notin \mathcal{F}(P)$. In other words, when checking if names are the same so that a reaction may occur, $\star \neq \star$: both the restriction $\nu$ and the receive operation will behave just as if $\star$
corresponds to a name which does not appear anywhere else in the expression. In practice, we will never use $\star$ as a channel, so this problem can be safely ignored.

We then define the free names \texttt{stdin} and \texttt{stdout} to represent respectively standard input and standard output. On the first of them, we can only receive values, and over the second one we can only send values.

Furthermore, within our goal of limiting the number of private names we employ in our translation, we introduce the notion of global locks. These will help us to model those operations that would either \textit{a}) be handled already at compile time, such as vtable offset resolution for matching dynamically bound methods, or \textit{b}) atomic operations at runtime, such as protecting a send operation and a choice over some receive operations, to emulate the “match” operator over a finite set of possible matching names (some sort of \texttt{switch} statement transposed in $\pi$-calculus, if you prefer).

To make it clearer that some name is in fact a global lock, we denote it with the letter $\lambda$. For each global lock $L$, we thus have a running process of the form:

$$GLOBALLOCK_L \overset{\text{def}}{=} \star \left( \lambda_L(\star), \lambda_L(\star) \right)$$

In our translation, we have the following always-active locks, which will be described in more detail in the following sections:

$$ALLGLOBALLOCKS \overset{\text{def}}{=} GLOBALLOCK_{\text{REF}} \mid GLOBALLOCK_{\text{FIELDS}} \mid GLOBALLOCK_{\text{INVOKES}}$$

### 2.2 Handling objects and classes

The agent names introduced in this section will need to begin with some kind of unique prefix when implemented, to avoid clashing with other agents that are the product of the translation. However we assume, for the scope of this document, that no clash is possible.

The same is true for different types: method names are different than field names, so in the final translation they should have different prefixes, or they will clash (as they are used as free names in $\pi$-calculus processes). Hence, for instance, a field could start with the letter “@”, and delimit a method name with a pair of square brackets: “[name]”.

We can have an indefinite amount of objects per each run, depending on the current configuration; clearly, assigning them free names is out of the question. Thus, once an object gets instantiated, it will need to be referred by some kind of restricted name in the corresponding $\pi$-calculus translation.
However, this introduces a problem. In fact, even quite common Java programs, never having more than $k$ objects in memory at any time of their execution, would result unbounded in their translation if we do not stop distributing unreachable names when we are done with them.

Consider for instance a web server, spawning new threads to handle incoming requests. Each separate thread instantiates a certain number of objects, and thus creates for each object a restricted name in the $\pi$-calculus translation. Then, for each object, we start sending over the object’s name channel a tuple containing locks, the vtable, and field values. This is necessary for an object’s users: to invoke methods on it, get its dynamic type, access or change its fields, lock on it.

When our webserver finishes handling a request, the corresponding thread exits. The webserver then takes the next request from the queue, and starts a new handler thread. In the JVM, old variables from finished execution contexts get garbage collected, and we can potentially continue handling new requests forever, as long as we have a bounded thread pool or a bound on the number of maximum concurrent requests.

We said that, in our translation, we need to continuously send an object’s class, data, etc. over a name representing the object. This needs to be done for all the time an object is active, since we do not know when and how often a method will be invoked, or a field accessed. However, if we never stop sending this restricted name, it will never be recycled, thus adding to the name bound. If we attempt to model the above web server without a way to free unused objects and their associated restricted names, we would clearly get an unbounded representation.

This fact brings in the need to somewhat emulate the behaviour of the garbage collector. We do so through a receive channel called $\text{dispose}$, one for each object. Receiving on it causes the $\text{OBJLIFE}$ process spawned when an object is created, to stop sending the name representing the object over and over again, and to terminate. To handle the reference count, we introduce the $\text{GCContaNT}$ agent. The idea is, when this agent ends its execution, we can dispose of the original object and associated resources.

### 2.2.1 Object instantiation

To be able to invoke methods on an object, change its fields, or lock onto it, we need to get hold of a name which acts as the object identity. Thus, whenever an object is allocated during the execution of a Java program through use of the $\text{new}$ keyword, we want to be able to query its properties indefinitely often. The $\text{MAKEOBJ}$ agent, given a return channel and a class name, will return a new private name that can be used to access such elements. Then, a static constructor will need to be invoked passing the name received on the return channel as the “$\text{this}$” parameter.

---

$^2$That is, bar other leaks in the JVM itself or in native methods.
So, let us assume that in a Java program we encounter the `MyClass p = new MyClass (o);` statement. That would translate to:

\[
\{\ldots\}. \text{vret.vret}_2.(\text{MAKEOBJ}[\text{ret, MyClass}]
  \mid \text{ret}(p).\text{MYCLASS}_{\text{CONSTRUCTOR}}[\text{ctx, } p, e, \text{ret}_2, o]
  \mid \text{ret}_2(p).\{\ldots\})
\]

where `ctx` was already known as the current context, and `e` is the current exception channel. The exact need of these extra parameters will become clear in the following sections.

\[
\text{MAKEOBJ}(\text{ret, class}) \overset{\text{def}}{=} \text{v0.vrefcount.vctxlock.vcleanup.vdispose.vfreelocks.}
\]

\[
\text{class}(\text{allocator, super, vtable, sdata}).\text{allocator}(data, \text{deallocate}).
\]

\[
\text{(OBJLIFE}[\text{class, data, refcount, ctxlock, dispose}]
  \mid \text{MAKESIMPLELOCK}[\text{ctxlock, freelocks}]
  \mid \text{GCCOUNT}[\text{refcount, cleanup}]
  \mid \text{refcount}(\text{add}, \text{ret}(o))
  \mid \text{cleanup}(\ast). (\text{dispose}(\ast) \mid \text{deallocate}(\ast) \mid \text{freelocks}(\ast))
\]

In Java, creating an object comprises allocating some space on the heap to hold field values. We model that by asking a new `data` name through a class’s `allocator`, which will be introduced in Section 2.2.3.

We then start repeatedly sending over a closure the object’s dynamic type (`class`), its entry point for `data` access and modifications, its reference count `refcount`, and a context lock `ctxlock` which can be used to synchronize with other execution contexts over critical sections. This is done by `OBJLIFE`.

When an object is created, it has a reference immediately added to it. However, whenever the reference count is empty (how this happens will be explained in more detail in Section 2.2.2), the `cleanup` channel is signalled, disposing of resources, and stopping `OBJLIFE`’s execution. This effectively enables us to “forget” about an object’s name which is not in use anymore. It goes without saying, that a Java program creating an unbounded number of objects but never releasing any of them, is going to soon run out of memory. Therefore, if the original program uses an unbounded number of resources, our translation will be, unsurprisingly, unbounded too.

We have a series of actions\(^3\) that get signalled or signal when it is time to stop some process’s execution. They are:

\(^3\)We will call actions those send and receive operations being performed over channels that
• *cleanup*, which is an action signalled when *refcount* terminates (modelling when the reference count for this object reaches zero). *cleanup* only unguards the send operations for *dispose*, *deallocate*, and *freelocks*.

• the *dispose* action stops sending over the channel with the object’s name the closure hold by *OBJLIFE*.

• *deallocate* is used by *FIELDOPS* to know it has to stop allowing accesses or writes to the object’s fields. This models removing the object from the heap in Java.

• the *freelocks* action stops the process that allows translations of Java’s synchronized blocks to lock on this object.

After these four actions reacted, no process should attempt to use the object name for accessing its fields or methods, or to lock on it. Trying to do so would block the program and prevent it from continuing.

\[
\text{OBJLIFE}(\text{class, data, refcount, ctxlock, dispose}) \overset{\text{def}}{=} \\
(\sigma(\text{class, data, refcount, ctxlock}). \\
\text{OBJLIFE[\text{class, data, refcount, ctxlock, dispose}]} \\
+ \text{dispose}(\star))
\]

*OBJLIFE* is pretty much straightforward: it just continuously sends over the object channel the values it received upon its first invocation, until it receives *dispose* in order to terminate.

### 2.2.2 Reference counting and object disposal

As already mentioned in the previous sections, it is necessary for our translation to find a way to terminate distributing private names corresponding to objects which are no longer reachable from other processes. The simplest and yet general enough way to do so, is to allow for a reference counting mechanism to take place.

One of the obvious limitations of this approach, common in many frameworks that allow for shared pointers implemented through reference counting, is that it does not work for freeing circular references.

That is, if we hold a reference to A, which holds a reference to B, which in turn holds back a reference to A again, the reference counts for A and B will be respectively \(rc_A \geq 2\), \(rc_B \geq 1\). If we let go of our reference inside the program, use the \(\star\) unit value as their only parameter. We stretch the language slightly, to say that we *signal an action* when it is a send operation; we instead call the corresponding receive operation *reacting to an action*. While not rigorous, this language should prove intuitive enough.
$rc_A$ will be decremented by one, but will not reach 0. This effectively prevents B from being disposed of.

While it might be possible in principle to allow for a more complex and fine-grained garbage-collection method, by associating references to contexts and using a mark-and-sweep approach, the current reference-counting method should prove enough to cover many real-world cases that do not rely on circular references.

The first step in building such a GC, would be to build a queue of known objects within each context, plus a global pool representing the heap, and use \texttt{freeref} on all objects in the queue when a context goes out of scope. Object fields would not need to be explicitly inserted in these queues, since they are reachable through some private name representing their containing object. However, this is considerably more difficult to handle, so we decided to stick with a simpler approach that still works for a relevant subset of Java programs.

As a consequence, bounded Java programs sporting circular references, that spawn an unbounded number of threads, will result in a name-unbounded translation in $\pi$-calculus.

Our translation did not want to force developers to touch their sources, but instead to perform a static analysis on the original code only. Relaxing this constraint might lead to one other possible workaround: it might be feasible to ask the programmer to help resolving circular references, through the use of weak references which do not, in fact, contribute to increasing the reference count. This might be done via annotations in the source code. However note that this method requires a good understanding of the program data structures and their interactions, something which is not always possible.

We now start our treatise of how to handle reference counting by our per-object set-up agent, \texttt{gccount}. It works over the \texttt{refcount} name, which is associated with an object tuple, as can be seen in \texttt{makeobj} above. It also accepts another parameter, \texttt{onend}, which is an action to be signalled when all references have been released. In our case, it will be \texttt{cleanup} from \texttt{makeobj}. 
\begin{align*}
\text{GCCCOUNT} \left( \text{refcount}, \text{onend} \right) & \triangleq \\
& \text{vaddop, vremoveop. (}
\text{refcount}(\text{op}).\lambda_{\text{ref}}(\star).\overline{\text{op}}(\star) \\
& \mid \text{add}(\star).\overline{\lambda_{\text{ref}}(\star)}.\nu\text{spawner. vhalter. (}
\text{GCCCOUNT}_{\text{DMX}}[\text{refcount, halter, addop, removeop}]
\mid \text{GCCCOUNT}_{\text{ADD}}[\text{refcount, spawner, halter, addop}]
\mid \nu l. (\text{GCCCOUNT}_{\text{REMOVE}}[\text{refcount, l, spawner, removeop}]
\mid l(\star). (\overline{\text{halter}}(\star) \mid \overline{\text{halter}}(\star) \mid \overline{\text{onend}}(\star))
\)}
\end{align*}

\text{refcount} is used as a channel to discriminate whether we are adding or removing a reference. It thus accepts the free names \text{add} and \text{remove} as parameters. At the beginning, the only operation we allow on \text{refcount} is an add; it is immediately done in \text{MAKEOBJ}, so that we are sure to return a name to an object which can be disposed of through \text{FREEREF}.

We thus spawn three processes:

1. the result of an invocation to \text{GCCCOUNT}_{\text{DMX}}. It is a process which listens over \text{refcount}, accepts an operation as a parameter (the free names \text{add} or \text{remove}), and translate them into the restricted names \text{addop} or \text{removeop}.

This is necessary, so that \text{GCCCOUNT}_{\text{ADD}} and \text{GCCCOUNT}_{\text{REMOVE}} are not beginning with receive operations over free names which might be unguarded in other processes spawned by other calls to \text{GCCCOUNT} during other objects’ instantiation. Else, any \overline{\text{add}}(\star) or \overline{\text{remove}}(\star) might react with any instance of \text{GCCCOUNT}_{\text{ADD}} currently active – specifically, one pertaining to a different object reference counting mechanism.

\text{GCCCOUNT}_{\text{DMX}} receives as a parameter, other than the private names corresponding to the add and remove operations, also a \text{halter} action. When this action will react, this process will stop.

2. the result of an invocation\textsuperscript{4} to \text{GCCCOUNT}_{\text{ADD}}. It also receives \text{halter} as a parameter. We use the same name, since both \text{GCCCOUNT}_{\text{DMX}} and \text{GCCCOUNT}_{\text{ADD}} will always be stopped together, and we do not care about the order by which this happens.

\textsuperscript{4}For shortness, we will, from now on, also use the locution “the process \text{AGENTNAME}” as an abbreviation to “the result of the invocation to agent \text{AGENTNAME}”.
The `spawner` parameter is used as an action by `GCCOUNTADD` to signal `GCCOUNTRemove` it has to take a new reference into account.

3. we create a new name `l` to represent a lock or barrier of some sort. We then proceed to invoke `GCCOUNTRemove` passing this name as a parameter, as well as the `spawner` action.

In parallel, we start a process guarded on `l` that will then take care of stopping the other two processes (`GCCOUNTdMx` and `GCCOUNTADD`), and call the `onend` action, when it reacts.

The main idea behind it is as follows. Both `GCCOUNTdMx` and `GCCOUNTADD` will be available and responding to send operations for all the time the reference counter has not reached zero, i.e. `GCCOUNTRemove` terminates and unguards the halters guarded by `l`.

`GCCOUNTRemove`, as we will see, will be able to spawn new instances of itself, building a queue and signalling an action each time told to pop an element from it. The last action to be signalled will be exactly the restricted name `l` from this agent.

Let us now go into the details of each one of the ancillary agents that are invoked in the body of `GCCOUNT`, starting with `GCCOUNTdMx`, where “DMX” stands for “demultiplex”.

```latex
\begin{align*}
\text{GCCOUNTdMx} \ (\text{refcount}, \text{halter}, \text{addop}, \text{removeop}) & \overset{\text{def}}{=} \\
\text{halter}(\star) & + \\
(\text{refcount}(\text{op}).\lambda_{\text{ref}}(\star). (\text{ref}(\star) \mid \\
((\text{add}(\star).\text{addop}(\star). \\
\text{GCCOUNTdMx}[\text{refcount}, \text{halter}, \text{addop}, \text{removeop}]) \ + \\
(\text{remove}(\star).\text{removeop}(\star). \\
\text{GCCOUNTdMx}[\text{refcount}, \text{halter}, \text{addop}, \text{removeop}]) \\
}))
\end{align*}
```

The role of `GCCOUNTdMx` is to receive whichever operation we want to perform over `refcount`, namely the free names `add` or `remove`, and to convert them into the corresponding restricted name, respectively `addop` or `removeop`, as given by its invoker.

This is important, as we already noted, to avoid having active processes unguarded on the free names `add` or `remove`; instead, `GCCOUNTADD` and `GCCOUNTRemove` can rely to be activated only by operations coming from the same object. That we can have only one possible conversion at a time from the
free name to a local restricted name, is ensured by the global $\lambda_{ref}$ lock, which is taken here and released in either $GCCCOUNT_{ADD}$, or $GCCCOUNT_{REMOVE}$.

$GCCCOUNT_{ADD}$ is indeed quite simple, as it only tells $GCCCOUNT_{REMOVE}$ that a new reference appeared, and to spawn more processes to match the number of active references:

\[
GCCCOUNT_{ADD}(\text{refcount}, \text{spawner}, \text{halter}, \text{addop}) \triangleq \\
\text{halter}(\star) + \\
(\text{addop}(\star). (GCCCOUNT_{ADD}[\text{refcount}, \text{spawner}, \text{halter}, \text{addop}] \\
\mid \text{spawner}(\star).\lambda_{ref}(\star)))
\]

An attentive reader will notice that $GCCCOUNT_{ADD}$ can be subsumed inside $GCCCOUNT_{DMX}$ (thus dispensing us to pass the restricted $\text{addop}$ name around), since what it does is just signalling $GCCCOUNT_{REMOVE}$ to spawn two more processes, and returning to its initial quiescent state. We have decided to keep it for more clarity about the role of these agents, but it is indeed an optimization that can be easily made.

$GCCCOUNT_{REMOVE}$ is designed so that it can either “pop” a reference from the conceptual queue, or enlarge the queue by spawning a new process that will call $l_2$ when signalled for removal, and replacing itself with a guarded version in $l_2$.

\[
GCCCOUNT_{REMOVE}(\text{refcount}, l_1, \text{spawner}, \text{removeop}) \triangleq \\
(\text{spawner}(\star).l_2. (l_2(\star).GCCCOUNT_{REMOVE}[\text{refcount}, l_1, \text{spawner}, \text{removeop}] \\
\mid GCCCOUNT_{REMOVE}[\text{refcount}, l_2, \text{spawner}, \text{removeop}]])
\]

\[
+ (\text{removeop}(\star).\lambda_{ref}(\star).l_1(\star))
\]

If asked to decrease the counter (by receiving the $\text{removeop}$ action), it will signal the next process in the queue so that it can become unguarded. For the last reference, this means that $\text{halter}^2$ and $\text{onend}$ can be signalled.

Each time $\text{spawner}$ is signalled, two processes are spawned. The first one is nothing else that a recursive call to the agent itself, with the same parameters, but guarded by a pending receive operation on $l_2$; the second process, instead, receives $l_2$ as a parameter. Then, if someone attempts to remove a reference, $l_2$ will be signalled, and the first of these two processes will become unguarded.

Note how, if the reference count is not zero, there is always exactly one unguarded $GCCCOUNT_{REMOVE}$ instance at a time per each object / $\text{refcount}$, so there is only one possible receiver for a $\text{spawner}$ or $\text{removeop}$ action.
This agent is thus capable of conceptually increase and decrease the counter until it reaches zero, when the last `onend` action is signalled.

Let us now give an example run for the `GCCOUNT` agent. We will add a second reference after the first coming from `MAKEOBJ`, and remove both to check that `cleanup` is called. We will note how, after adding the first reference, and before removing the last (when `rc_obj` = 1), we will have two structurally congruent processes in our hands – specifically, they are one the α-conversion of the other.

Our run starts inside `MAKEOBJ`, where we have:

\[
\text{GCCOUNT}[\text{refcount, cleanup}] \mid \text{refcount}(\text{add})
\]

Adding the first reference reacts to, by mere expansion:\footnote{We do not explicitly mention whose names are restricted and whose free, since it should be clear by looking at the original agent definitions.}

\[
\text{halter}(\star) + (\text{refcount}(\text{op}).\lambda_{ref}(\star). \langle \overline{\sigma}\mathcal{P}(\star) \rangle | \\
( \langle \text{add}(\star), \text{addop}(\star), \text{GCCOUNT}_{\text{DMX}}[\text{refcount, halter, addop, removeop}] \rangle + \\
(\text{remove}(\star), \text{removeop}(\star). \text{GCCOUNT}_{\text{DMX}}[\text{refcount, halter, addop, removeop}])))
\]

| \text{halter}(\star) + (\text{addop}(\star). (\text{GCCOUNT}_{\text{ADD}}[\text{refcount, spawner, halter, addop}] \\
| \quad | \text{spawner}(\star). \lambda_{ref}(\star))) | \\
| (\text{spawner}(\star). \nu l_2. (l_2(\star). \text{GCCOUNT}_{\text{REMOVE}}[\text{refcount, l_1, spawner, removeop}] \\
| \quad | \text{GCCOUNT}_{\text{REMOVE}}[\text{refcount, l_2, spawner, removeop}])))
\]

\[
+ (\text{removeop}(\star). \lambda_{ref}(\star). l_1(\star)) \\
l_1(\star). (\text{halter}(\star) \mid \text{halter}(\star) \mid \text{cleanup}(\star))
\]

We can notice how this process is ready to receive either a signal to add a new reference, thus spawning some more processes, or to remove it, thus signalling on `l_1`, which in turn would unguard the two `halter` actions and `cleanup`.

Now we proceed to add another reference; we have to react with another `refcount(add)`. For illustrative purposes, we will already expand the invocation to `GCCOUNT_{REMOVE}[\text{refcount, l_1, spawner, removeop}]`, even if guarded by `l_2`. This will make understanding what is going on a bit easier.

This results in the following process:
halter(*) + (refcount(op).\lambda_{ref}(\cdot). (\overrightarrow{op}(\cdot) | \\
( (add(*).addop(*).GCCOUNT_DMIX[refcount, halter, addop, removeop]) + \\
( remove(*).removeop(*). \\
GCCOUNT_DMIX[refcount, halter, addop, removeop])) \\
)) \\
| halter(*) + (addop(*). (GCCOUNT_ADD[refcount, spawner, halter, addop] \\
| _{\cdot}spawner(\cdot). \lambda_{ref}(\cdot))) \\
| l_2(*). (spawner(*).vl_3. (l_3(*).GCCOUNT_REMOVE[refcount, l_1, spawner, removeop] \\
| GCCOUNT_REMOVE[refcount, l_3, spawner, removeop])) \\
+ (removeop(*).\lambda_{ref}(\cdot).l_1(\cdot)) \\
| (spawner(*).vl_4. (l_4(*).GCCOUNT_REMOVE[refcount, l_2, spawner, removeop] \\
| GCCOUNT_REMOVE[refcount, l_4, spawner, removeop])) \\
+ (removeop(*).\lambda_{ref}(\cdot).l_2(\cdot)) \\
| l_1(*). (halter(*) | halter(*) | cleanup(\cdot))

For clarity, we have chosen to give different names ($l_3$ and $l_4$) to the actions guarding other processes. Now a remove operation will bring $l_2$ to be signalled (in green above), thus unguarding its sibling (in red above).

We can still see that the process corresponding to the expansion of the invocation of the demultiplexer, and the one corresponding to the adder, are always the same, as they are tail-recursive on themselves.

Removing a reference will indeed bring us back to when the counter was $1$, by reacting with the second branch of the choice inside the green process:

\[
halter(*) + (refcount(op).\lambda_{ref}(\cdot). (\overrightarrow{op}(\cdot) | \\
( (add(*).addop(*).GCCOUNT_DMIX[refcount, halter, addop, removeop]) + \\
( remove(*).removeop(*). \\
GCCOUNT_DMIX[refcount, halter, addop, removeop])) \\
)) \\
| halter(*) + (addop(*). (GCCOUNT_ADD[refcount, spawner, halter, addop] \\
| _{\cdot}spawner(\cdot). \lambda_{ref}(\cdot))) \\
| (spawner(*).vl_3. (l_3(*).GCCOUNT_REMOVE[refcount, l_1, spawner, removeop] \\
| GCCOUNT_REMOVE[refcount, l_3, spawner, removeop])) \\
+ (removeop(*).\lambda_{ref}(\cdot).l_1(\cdot)) \\
| (spawner(*).vl_4. (l_4(*).GCCOUNT_REMOVE[refcount, l_2, spawner, removeop] \\
| GCCOUNT_REMOVE[refcount, l_4, spawner, removeop])) \\
+ (removeop(*).\lambda_{ref}(\cdot).l_2(\cdot)) \\
| l_1(*). (halter(*) | halter(*) | cleanup(\cdot))
\]

As can be seen, this process is still able to add new references if we want to. However, removing the last reference will result in $l_1$ to be signalled:
\[ halter(\star) + (\text{refcount}(op) \cdot \lambda_{ref}(\star) \cdot (\bar{\sigma}(\star)) | \]
\[ (\text{add}(\star) \cdot \text{addop}(\star) \cdot \text{GCCCOUNT}_{DMX}[\text{refcount}, \text{halter}, \text{addop}, \text{removeop}] + \text{remove}(\star) \cdot \text{removeop}(\star). \]
\[ \text{GCCCOUNT}_{DMX}[\text{refcount}, \text{halter}, \text{addop}, \text{removeop}])}
\]}
\]
\[ \text{halter}(\star) + (\text{addop}(\star). \text{GCCCOUNT}_{ADD}[\text{refcount}, \text{spawner}, \text{halter}, \text{addop}]
\]
\[ | \text{spawner}(\star) \cdot \lambda_{ref}(\star))}
\]}
\[ \text{halter}(\star) | \text{halter}(\star) | \text{cleanup}(\star)
\]

This will in turn signal \text{cleanup} and end the current process, by sending \text{halter} twice.

Since \text{GCCCOUNT} simulates a counter, it does not check if a reference pertains to the object that contains \text{refcount}. We have to be careful, in our translation, to add and remove references in the right order, so that an object can never be disposed of by mistake.

To our toolbox we add two last agents, to add and remove references. They take an additional \text{done} parameter to ensure they can be synchronized with the rest of the program flow (so that the program cannot continue if a \text{refcount} has not been told to increase or decrease its counter).

This is of course done to ensure that we do not have a counter which can reach zero because of a bad interleaving of \text{MAKEREF} and \text{FREEREFER} invocations.

\[ \text{MAKEREF}(o, \text{done}) \overset{\text{def}}{=} \]
\[ o(\text{class}, \text{data}, \text{refcount}, \text{ctxlock}). \text{refcount}(\text{add}). \text{done}(\star) \]

\[ \text{FREEREFER}(o, \text{done}) \overset{\text{def}}{=} \]
\[ o(\text{class}, \text{data}, \text{refcount}, \text{ctxlock}). \text{refcount}(\text{remove}). \text{done}(\star) \]

Of these two, only \text{MAKEREF} has a stronger requirement of being synchronized with the rest of the execution context body; if preferred, \text{FREEREFER} can be done in parallel to other operations, since we do not care when an object will exactly be disposed of by the garbage collector. Here we have chosen to synchronize both, since also decreasing the counter inside a single execution context (a single thread), is conceptually a sequential operation inside a Java thread.
2.2.3 Field and method resolution

As a modern object-oriented language, Java offers such much-needed features as polymorphism through inheritance and dynamic method resolution. No translation would therefore be interesting enough without taking them into account.

In this section, we are going to see how the following mechanisms are modelled in our $\pi$-calculus translation: a) an object’s data allocation, b) inheritance and dynamic method resolution, and c) accessing and changing fields.

We have already seen how each object name is able to respond, while the object’s reference count has not reached zero, with a tuple containing the names: class, data, refcount, ctxlock. We will now see specifically the two names that are related to classes themselves: class and data.

class is nothing else than a name which is used to assess an object dynamic type at runtime. As an aside, thanks to this model, this part of our translation would require only little adjustments to be adapted to other languages which implement pure message-passing semantics, and that are not strongly typed, such as Ruby or Python.

data is instead a new name sent over an allocator upon request. It represents a new object’s data allocated on the heap during a call to new. Each class allocates a different amount of memory for holding fields upon object instantiation, and this is why each class has a different allocator.

As the first part of our solution, we need a process able to continuously respond to queries about each class. We assume that classes are not dynamically loaded upon their first use, but rather that all known classes in use by the program are known at compile time. This is not unreasonable for many Java programs; however the translation could be expanded at a later time to take in account class loading semantics if needed (in most Java implementations, a loaded class cannot be unloaded anyway).

Since we assume that we have a clear bound on the number of classes known at compile time, the resulting translation will still be bounded even if these processes never stop, provided we do not allocate an unbounded number of objects without never de-referencing them. In this latter case, $\text{ALLOCATE}_k$ would always provide an unbounded number of new restricted data names; however, even the original Java program would be unbounded in the number of active objects, and thus sooner or later crash from memory exhaustion.

When we initialize a class, we start responding on a free name which is equal to the class name. Usually, class names in Java are written in CamelCase but start with an uppercase letter; they should be easy to spot in our translation. An actual implementation might want to prefix these names with a special character which renders them invalid Java syntax for variable or constant names. This should ensure we do not introduce name clashes.
For each class \( k \) in use by the program we are translating, we have a corresponding agent \( \textsc{initclass}_k \):

\[
\textsc{initclass}_k \overset{\text{def}}{=} \text{allocator.\nuvtatable.\nu\text{data}}.
\]

\[
\overset{\ast}{\text{class-name}}(\text{allocator}, \\text{superclass-name}, \\nu\text{table}, \nu\text{data})
| \text{allocate}_k[\text{allocator}]
| \text{methods}_k[\nu\text{table}]
| \nu\text{notused}.\text{fieldops}_{k_{\text{static}}}[\text{sdata}, \text{notused}, \text{defaultValue}^n]
\]

The super-class name is the name of the class coming after the \texttt{extends} keyword in a Java class declaration, or \texttt{Object} if it is not present. We set the super-class for \texttt{Object} as \texttt{Object} itself, to indicate it is the root of the whole class hierarchy.

We also provide access to the static data of a class through the \( \text{sdata} \) name. This space on the heap is never freed, and the \text{fieldops}_{k_{\text{static}}} \) process does not stop.

At present, we do not provide a way to check if an object is an instance of a certain class type, because we do not need it for the type of translation we are building. Since, as we will see in section 2.3, we always explore all branches in the program in a non-deterministic way, and since the \texttt{instanceof} operator is most of the times used as a branch selector (being used as a boolean condition), translating it would be superfluous because \( a \) it bears no side-effects, and \( b \) we explore all branches anyway.

We now introduce \text{allocate}_k \), which will prepare the data structure and allow us to access and change the object state:

\[
\text{allocate}_k(\\text{allocator}) \overset{\text{def}}{=} \nu\text{deallocation.\nu\text{data.\text{allocate}}}(\\nu\text{data.\text{deallocate}}).
\]

\[
\nu\text{data.\text{allocate}} = \text{allocate}_k[\\text{allocator}] | \text{fieldops}_{k_{\text{static}}}[\\text{data}, \\text{deallocate}, \text{defaultValue}^n]
\]

Each time a new object is created, a receive operation takes place in \texttt{makeobj} over the \texttt{allocate} channel. The allocator sends back a new private name to refer to the object’s data as a whole, representing the memory allocated on the heap. \texttt{deallocate} is used to signal when a specific object is to be garbage-collected, so that the process we use for holding the monadic state\(^6\), \text{fieldops}_{k_{\text{static}}} \), is free to terminate.

\(^6\text{It is interesting to note how \( \pi \)-calculus’s processes are free from side-effects (since no name can be re-assigned, but new names can hide others from the outer scope). This leads us to employ some techniques that, at an abstract level, are comparable to other pure functional languages, such as Haskell. In this case, changing state after a \texttt{set} operation, or keeping the old state after a \texttt{get} operation, is closely related to the work done by the State monad.}\)
We have seen two calls to \texttt{FIELDOPS} so far: \texttt{FIELDOPS}_{k,\text{STATIC}} works only over the static fields of a class; \texttt{FIELDOPS}_{k} over all the others.

Initially, \texttt{FIELDOPS}_{k} gets called with \( n \) default values, where \( n \) is the number of fields specified in the declaration of a class \( k \). The default value for references to other objects is \texttt{null}; this will be introduced separately in Section 2.2.5. Albeit not particularly interesting for us (and thus they could be omitted by this translation), we can introduce the free name \texttt{zero} for the value 0, and \( \bot \) to represent the value \texttt{false} (by extension, \( \top \) would be \texttt{true}), and use those as default values for literals.

\[
\texttt{FIELDOPS}_{k}(\texttt{data, dealloc, }\bar{x}) \overset{\text{def}}{=} \\
(\texttt{dealloc}(\ast).\texttt{RELEASEOBJFIELDS}_{k}[\bar{\bar{x}}]) + \\
(\texttt{data}(\texttt{op, f, v, e}).\lambda_{\text{fields}}(\ast). (\bigtriangledown \texttt{f}\langle f, v \rangle | \texttt{FIELDOPS}_{k}[\texttt{data, dealloc, }\bar{x}] ) \\
+ \texttt{set}(f, v). (\bigtriangledown v | \sum_{i=0}^{n} f_{i}(v).\lambda_{\text{fields}}(\ast)\texttt{FIELDOPS}_{k}[\texttt{data, dealloc, }\bar{x}_{j<i}, v, \bar{x}_{j\geq i} ] ) )
\]

When started, the process spawned by an invocation of \texttt{FIELDOPS}_{k} will prepare to receive an operation – either the free name \texttt{get} or the free name \texttt{set} – over the \texttt{data} channel. Recall that \texttt{data} is a private name which can be retrieved through a receive operation over an object name.

Again, we employ a global lock, \( \lambda_{\text{fields}} \), to protect an operation which would be atomic in Java since already resolved at compile time: retrieving the location in memory which stores a particular field, as an offset from the start of the object data structure on the heap. Fields are in fact resolved statically. Since however we need to do some matching above the operation (\texttt{get} or \texttt{set}) and the field name, we need to guard this to prevent interference from other running processes. This of course works as long as \texttt{get} and \texttt{set} are not used as unguarded receive operations in other processes in our translation (which we made sure is indeed the case).

The \texttt{get} branch just retrieves the wanted field among all \( n \) fields; it is akin to the pattern matching over tuples so common in many functional programming languages. We also respawn the same process, since the current state is unchanged. This is comparable to the \texttt{get} operation over a State monad.

The \texttt{set} branch does not return anything, but instead replaces then current state as the \texttt{put} function would do in the State monad. It spawns a new process, altering the \( \bar{x} \) parameters this agents closes over: when setting a field \( i \) with the new value \( v \), the new process receives \( v \) in place of \( x_i \).

Note that both these operations can interleave as soon as the global lock is released; this is by design, since we still want to catch possible deadlocks or data inconsistencies present in the original program.

\texttt{FIELDOPS}_{k} will call \texttt{RELEASEOBJFIELDS}_{k} when signalled by \texttt{MAKEOBJ} because the \texttt{refcount} of an object has reached zero.
A small drawback of this technique is that it does not immediately allow for complete field hiding semantics. In fact, in a subclass, when a field is declared with the same name of one present in the class it is extended from, it is hidden in the subclass but still accessible by methods of the superclass. If we wanted to extend our translation accordingly, we would need to make field resolution hierarchical, with each superclass allocating separately its own set of fields on the stack. This, however, is more fine-grained than what we need, so it is deferred to a later revision.

However, there is an easy workaround: to rename fields that hide others from superclasses so that in our translation so that they are unique per class hierarchy. This preserves the notion that fields are statically resolved anyway, and as such simulates Java’s behaviour well enough. Take, for instance, the Java program shown in Figure 2.1.

There, our translation would use the name $f$ for the field $f$ in C, and $f_2$ for the $f$ visible within D. A translation for a call on `new D().m()` will then access immediately $f_2$, then statically invoke C.m, which will in turn access and retrieve

---

Outside the subclass, the field from the static type will always be the one resolved by the compiler when accessed.
the value of $f$.

$RELEASEOBJ$ is quite straightforward. We just need to free all references we have, so that object that were known only to their container can now be disposed of. We need just one restricted name, as we do not care in which order the action $done$ is signalled.

$$RELEASEOBJ(x) \overset{def}{=} \nu\text{done.}\prod_{i=0}^{N}(\text{FREEREF}[x_i, \text{done}] \mid \text{done}(\star))$$

where $\forall i$, $x_i$ is a reference to an object.

We then add some agents to help us to get or set field values. For getting literals, this is really easy:

$$\text{GETFIELD}_{\text{LITERAL}}(o, \text{field}, \text{ret}) \overset{def}{=} o(\text{class}, \text{data}, \text{refcount}, \text{ctxlock}.)\nu\text{ret}_2.$$  

Objects need just slightly more care, so that the returned value has its reference counter increased. This can be thought of as if we were “making public” to another process the private name $v$, which corresponds to a certain field value. We synchronize on $done$, so that the object cannot be returned over $\text{ret}$ before its reference count has been incremented.

$$\text{GETFIELD}_{\text{OBJ}}(o, \text{field}, \text{ret}) \overset{def}{=} o(\text{class}, \text{data}, \text{refcount}, \text{ctxlock}.)\nu\text{ret}_2. (\text{data}(\text{get}, \text{field}, \text{ret}_2)$$

$$\mid \text{ret}_2(v)\nu\text{done.}(\text{MAKEREF}[v, \text{done}] \mid \text{done}(\star).\overline{\text{ref}}(v)))$$

Unsurprisingly, setting a literal is just akin to getting it. We ask for a $done$ parameter so that the caller can synchronize over it if it wants. This is important because all operations inside the same execution context should happen conceptually in sequence (as it represents only one thread in Java).

$$\text{SETFIELD}_{\text{LITERAL}}(o, \text{field}, v, \text{done}) \overset{def}{=} o(\text{class}, \text{data}, \text{refcount}, \text{ctxlock}.)\nu\text{data}(\text{set}, \text{field}, v)\nu\text{done}(\star)$$

Now, $\text{SETFIELD}_{\text{OBJ}}$ asks for a bit more care. Since we are replacing a value, the old one should have its reference count decreased. At the same time, the
value we are setting should see it increased, as now it is known also to (via) its containing object.

The order by which we perform these operations is important to ensure correct operations. In particular, we need to first increase the counter for the new value, and then decrease it for the older. To see why, imagine the assignment \texttt{obj.field = obj.field}. If the order was reversed, and initially the counter was 1, \texttt{obj.field} could see it decreased to 0, and be garbage collected. Instead, the other way round, it will go up to 2 before returning to 1.

These operations are synchronized over some private \texttt{done}_i names.

\[
\text{SETFIELD}_\text{OBJ}(o, field, v, done) \overset{\text{def}}{=} o(\text{class}, data, ref\_count, ctx\_lock).vdone_2.
\]

\[
(\text{MAKEREF}[v, done_2] \mid done_2(\ast).vret_2, (\text{data}(\text{get}, field, ret_2).vdone_3.(ret_2(\text{old}).\text{FREEREF}[\text{old}, done_3] \\
| done_3(\ast).\text{data}(\text{set}, field, v).\text{done}(\ast)))))
\]

At this point, it is time to move our attention to dynamically resolving method calls to an object. This is done via a per-class \texttt{vtabe} channel, which accepts the exception channel as a parameter (this is useful mainly for the \texttt{null} object that we will see later on).

We use another global lock \(\lambda_{\text{invoke}}\) to protect the matching of the method name. It is taken during an instance of \(\#\text{INVOKE}\), and released as soon as the correct method to call has been chosen. Once more, this is an operation that is already decidable at compile time: the compiler needs only to add an offset, corresponding to the wanted field, to the virtual table linked to an object. This will result in the location where to jump with a context’s IP (instruction pointer) after setting up the stack frame for the function call.

\[
\text{METHODS}_K(vtable) \overset{\text{def}}{=} vtable(e). \sum \left(\text{method-name}_j(ctx, o, e, ret, \tilde{x}_i) \cdot \text{METHODS}_K[vtable] \mid M_j[ctx, o, e, ret, \tilde{x}_i]\right)
\]

where each method-name\(_j \in \text{Methods}_k \setminus (\text{PrivateMethods}_k \cup \text{StaticMethods}_k) \cup \text{Methods}_p\), with \(p\) being the immediate superclass of \(k\).

Note that static, private, and methods resolved through the \texttt{super} keyword do not use dynamic method resolution. Rather, they are resolved statically at compile time; as such, they do not employ \(\#\text{INVOKE}\) but are called directly by invoking the agent offering the translation of their implementation. Hence, they will not appear inside the method names of a class \(k\) in its instance of \(\text{METHODS}_K\).
The virtual table (\(vtable\)) enables us to resolve methods dynamically for an object of a certain type. There is one per class, and it contains all public, protected, and package-visible non-static methods. The visibility rules for overriding a non-static method are dictated by the Java Language Specification, so we do not need to do any specific work here: incorrect override directives will simply not compile.

The vtable for an object of class \(D < C\) will contain all non-overridden, non-static methods of \(C\) in addition to its own non-static methods. Overridden methods will of course share the same method-name, but result in the invocation of a different agent \(M_{K'}\) with the actual implementation provided in \(k\).

Static methods will not of course have a “\(this\)” parameter. Constructors are a special type of static methods, since they do, in fact, receive a “\(this\)” object as parameter! A constructor gets called just after \(MAKEOBJ\) returns; and inside it, the constructor from its superclass is implicitly called as the first thing that happens (if we do not want the default constructor, a different one can be specified by the programmer via the \texttt{super} keyword).

\[
#\text{INVOLVE}(ctx, o, e, \text{method\_name}, ret, \tilde{x}) \overset{\text{def}}{=} \\
oalign{\smallskip}
o(class, data, ref\_count, ctx\_lock). \\
\text{class}(\text{allocator}, \text{super}, vtable, sdata).\lambda_{\text{invoke}}(\ast) \\
\overline{\text{vtable}}(e).\overline{\text{method\_name}}(ctx, o, e, ret, \tilde{x})
\]

We do not allow calling a non-existing method on an object, and we rely on the Java compiler to bail out if the programmer tries to do so. If we wanted to implement also reflection, we would need to find a way to handle the \texttt{MethodNotFoundException}, something which might prove difficult without the if-then-else construct inside our version of \(\pi\)-calculus.

### 2.2.4 Object locking

Since we are interested in looking at problems that arise in a concurrent world, it would be strange to forego translating at least the most common synchronization primitives found in the Java language. Specifically, we would like to introduce some processes that cover \texttt{synchronized} block semantic, or monitors.

One of the notable requirements of monitors in Java, is of being \textit{re-entrant} (or \textit{recursive}). Namely, a certain execution context can acquire the same lock over an object as many times it needs to; however, once taken within a certain execution context, a lock cannot also be taken in another one, until all locks held by the former have been freed.

It turns out that providing a translation of re-entrant locks in our version of \(\pi\)-calculus, bare of the match and mismatch operators often found in other variations, is less than trivial. Thus, we will provide only a translation for non-
recursive locks. To motivate why this is not necessarily a bad limitation, we will need to take a short excursus.

Re-entrant locks, aka recursive mutexes, are not necessarily a good design decision. They often encourage the programmer to retain the lock longer than necessarily, thus limiting the performance benefits of multi-threaded programming. Also, they make keeping track of invariants in code more difficult, as a lock might already have been taken on the same object before, and this goes unnoticed because running the program leads to no deadlock. David R. Butenhof, author of “Programming with POSIX Threads” and POSIX thread consultant, also comments on the matter at hand [W1]:

The biggest of all the big problems with recursive mutexes is that they encourage you to completely lose track of your locking scheme and scope.

and also:

Sometimes you need to unlock. Even if you’re using recursive mutexes. But how do you know how to unlock? Threaded programming is built around predicates and shared data. When you hand a recursive mutex down from one routine to another, the callee cannot know the state of predicates in the caller. It has to assume there are some, because it cannot verify there aren’t; and if the caller had known that there were no broken predicates, it should have allowed concurrency by unlocking.

The program in Figure 2.2 shows how recursive locks can make our code slightly shorter, at the expense of clarity. In this example, f() synchronized on this before calling m(), which does the same. Having recursive locks here ensures no deadlock can occur, as the lock was already acquired in the same context.

On the other hand, let us stop for a moment and think about the conceptual meaning of “locks”. They are used to protect critical section of the code, i.e. a part of our program that has a shared mutable state. Their goal is to make operations inside the critical section appear as atomic, so that no external interference can happen. Hence, when a lock is not held we are stating that the current state is consistent (all of our invariants will always hold). If we need a lock, conversely, the state might not be consistent after some steps of computation, and we need to protect us from this happening.

Figure 2.2 shows how recursive locks make understanding whether invariants are holding or not more difficult; when we acquire the lock in m(), do invariants hold? And likewise, when we release it in m(), do invariants start holding again?

A possible way to make our code clearer (albeit slightly longer), with non-recursive lock semantics, is to introduce a separately named method, whose contract is of always being ran only after a lock on the current object has been taken.
We give an example in Figure 2.3, re-implementing the code of Figure 2.2, by introducing the private method `safe_m()`. While this is not always possible, it is often a good compromise which makes the programmer’s intent clearer, potentially reduces the size of critical sections, and improves debugging time in case of a logical programming mistake.

Therefore, when possible the programmer should ideally avoid using the re-entrant properties of locks. Giving a translation that only offers non-recursive locks then:

- on one hand, means that we are not modelling exactly what Java permits to do, and that we cover only a subset of the programs that can be written directly in the language without manual intervention;

- on the other, it will detect a deadlock when recursive locks are used, giving a chance to the programmer to fix their code to make their intent clearer.

To show the problem about translating recursive locks into $\pi$-calculus, we will now proceed to give a re-entrant version of object locks that employs the if-then-else construct, and comment on our solution. Afterwards, we will give another, less powerful formulation for locks that is not re-entrant, and we will stick with that.
class NonReentrantVersion
{
    private int i = 2;
    public int f ()
    {
        synchronized (this)
        {
            this.i += safe_m () + 1;
            return i;
        }
    }
    public int m ()
    {
        synchronized (this)
        {
            return safe_m ();
        }
    }
    private int safe_m ()
    {
        this.i *= 2;
        return i;
    }
}
We start by introducing the \texttt{makereentrant} agent, which is supposed to be called from \texttt{makeobj} (in place of the call to \texttt{makelock}) with an invocation like:

\[
vcancel.\texttt{makereentrant}[ctxlock, cancel, freelocks]
\]

\texttt{makereentrant} itself is quite easy, and it allows being stopped from another process through signalling over the \texttt{halter} action, in a pattern that by now should be familiar.

The main idea is that \texttt{makereentrant} is the state we are in when no context has yet taken the lock – in other words, no context has claimed the lock ownership.

If this is the case, we receive the requesting context in a receive operation over \texttt{ctxlock}, create a new name \texttt{lock} for being signalled the lock is to be freed, and send it back to the context. We then use \texttt{lockover} to bind future requests to the context \texttt{ctx}, until the last lock is released; then \texttt{l1} is signalled, and we spawn another instance of \texttt{makereentrant}.

\[
\texttt{makereentrant}(ctxlock, cancel, halter) \overset{\text{def}}{=} \texttt{halter}(\star) +
\]

\[
(\texttt{ctxlock}(ctx).\nu l.\nu lock.
\]

\[
\texttt{ctx}(\texttt{lock}).\texttt{lockover}[ctx, ctxlock, lock, l1, cancel, l1] \mid cancel(\star) + (\texttt{l1}(\star).\texttt{makereentrant}[ctxlock, cancel, halter])))
\]

As you can see, the \texttt{cancel} restricted name is passed from outside, because all processes spawned starting from an invocation of \texttt{makereentrant} should share it. It is used to cancel and replace the recursive call to \texttt{makereentrant}.

We have in fact two possible developments after a context claims the lock: either the lock is released (after eventually being taken more than once by the same context), or another context tries to get the lock before it is released.

In the former case, we want just to make the lock available again for taking, hence the recursive invocation to \texttt{makereentrant} itself.

In the latter case, however, we want this second, late-to-arrive context to block until the lock is not owned anymore. We thus queue this new request, and then continue waiting for the context that holds the lock ownership to release previous locks or request new locks.

This is done inside \texttt{lockover}:
\(\text{LOCKOVER}(\text{ctx}, \text{ctxlock}, \text{lock}, l_1, \text{cancel}, l_{last}, \text{halter}) \overset{def}{=} \)
\[
\begin{align*}
& (\text{lock}(\ast) . I_1(\ast)) + \\
& (\text{ctxlock}(\text{ctx}_2) . \text{if } \text{ctx} = \text{ctx}_2 \text{ then} \\
& \quad \nu \text{lock}_2 . \text{ctx}(\text{lock}_2) . \nu l_2 . \\
& \quad (\text{LOCKOVER}[\text{ctx}, \text{ctxlock}, \text{lock}, l_2, \text{cancel}, l_{last}, \text{halter}] \\
& \quad \mid l_2(\ast). \text{LOCKOVER}[\text{ctx}, \text{ctxlock}, \text{lock}, l_1, \text{cancel}, l_{last}, \text{halter}]) \\
& \quad \text{else} \nu l_2 . \\
& \quad (\text{LOCKOVER}[\text{ctx}, \text{ctxlock}, \text{lock}, l_1, \text{cancel}, l_2, \text{halter}] \\
& \quad \mid \overline{\text{cancel}(\ast)} . l_{last}(\ast). \nu \text{lock}. \\
& \quad (\text{ctx}_2(\text{lock}) . \text{LOCKOVER}[\text{ctx}_2, \text{ctxlock}, \text{lock}, l_2, \text{cancel}, l_2, \text{halter}] \\
& \quad \mid \text{cancel}(\ast) + (l_2(\ast) . \text{MAKERENTRANT}[\text{ctxlock}, \text{cancel}, \text{halter}])) \\
& )
\end{align*}
\]

It is inside \(\text{LOCKOVER}\) that this replacement of the recursive call to \(\text{MAKERENTRANT}\) takes place, and more precisely in the \textbf{else} branch. If, in the new request for a lock, \(\text{ctx} \neq \text{ctx}_2\), we:

- prepare a new instance of \(\text{LOCKOVER}\) to accept new requests, however using a new name \(l_2\) as the \(l_{last}\) parameter. \(l_{last}\) is used to remember which is the last action that signals that all locks of the current context have been freed, and here it needs to be changed.

- cancel the recursive call of \(\text{MAKERENTRANT}\), in order to replace it. The replacement does:

1. listen on \(l_{last}\) as the process we cancelled would have done. When the last lock for \(\text{ctx}\) is released, \(l_{last}\) will be signalled.
2. Then, \(\text{ctx}_2\) will already be considered the next bound context – as you can see, the green part of \(\text{LOCKOVER}\) is the same as \(\text{MAKERENTRANT}\), but for the prefix we already consumed, \(\text{ctxlock}(\text{ctx})\) (in red above). It will be finally the new owner of the object’s lock. The recursive invocation to \(\text{MAKERENTRANT}\) that we cancelled before is re-spawned, but this time it is protected by our new “last lock in the locking queue”, \(l_2\).

Thus, conceptually, we are handling two queues: one for contexts, and one for locks of the current lock owner. Note that, since a context either succeeds taking the lock, or blocks until it can, we cannot have more than one request from \(\text{ctx}_2 \neq \text{ctx}\) until \(\text{ctx}\) releases the last lock it holds.

Unfortunately, as we already said, it is rather difficult to simulate the same behaviour without the if-then-else primitive in \(\pi\)-calculus. We therefore settle for
one of the simplest implementations of locks. This still guarantees blocking if the
lock is taken more than once without being first released.

\[
\text{MAKELOCK}(\text{ctx}\text{lock}, \text{halter}) \overset{\text{def}}{=} \text{halter}(\star) + \\
(\text{ctx}\text{lock}(\text{ctx}).\nu\text{lock}.\text{ctx}(\text{lock}).\text{lock}(\star).\text{MAKELOCK}[\text{ctx}\text{lock}, \text{halter}])
\]

For the counterpart to handling out locks, acquiring them viz. synchronized
blocks, see section 2.3.6 at page 50.

2.2.5 The null object

In this section we introduce the “null object”, an artifact that helps us to emulate
the behaviour of Java’s null value for references.

There are mainly three sources of errors related to null values in Java pro-
grams: a) attempting to access a field of a null reference, b) attempting to call a
method of a null reference, and c) trying to lock over a null reference.

Rather than testing separately whether a reference is null before invoking a
method, accessing a field, etc. (something which would probably require a mis-
match operator in \(\pi\)-calculus), we prefer to create a proxy object that, when any
of the previous three actions is performed, throws unconditionally a NullPointerException.

A null reference, in our translation, is then a proxy object of type NullClass. It has no counterpart in Java, and it is represented by the following dedicated agent:

\[
\text{ALLOCATE}\text{NullClass}(\text{allocator}) \overset{\text{def}}{=} \text{vdata}.(\star\text{allocator}(\text{data}, \star) \\
| \text{vdata}(\text{op}, \text{f}, \text{v}, \text{e}).\text{THROW}[\text{e}, \text{NullPointerException}])
\]

Here, we just notice how the parent class of NullClass is NullClass itself. But what happens when someone tries to access some data by following a null reference? Well, we can just use one private name for the data of all null references, since it can be considered a singleton value. And when an operation is invoked on this channel, we throw a new NullPointerException:
Similarly, any attempt to access its \( vtable \) to invoke a method will have a similar result:

\[
METHODS_{\text{NullClass}}(vtable) \overset{\text{def}}{=} vtable(e).THROW[e, \text{NullPointerException}]
\]

Then, we just need a process akin to \( \text{MAKEOBJ} \), to respond to receive operations over the singleton \text{null} free name. Also, we model directly here the \text{NullPointerException} thrown if attempting to lock on a \text{null} reference.

\[
\text{MAKENULL} \overset{\text{def}}{=} \text{refcount}.\text{ctxlock}.
\]

\[
\text{NullClass}(\text{allocator, super}, vtable, sdata).\text{allocator}(\text{data, }\star).
\]

\[
(* \text{null}(\text{NullClass}, \text{data, refcount, ctxlock})
\]

\[
| *(\text{ctxlock}(\text{ctx, }e).\text{THROW}[e, \text{NullPointerException}])
\]

\[
| \text{refcount}(\star)
\]

Reference counting is not meaningful for the \text{null} object, so any operation received over \( \text{refcount} \) (be it \text{add} or \text{remove}) will just be ignored.

\section*{2.3 Control structures and program flow}

In this section we give a translation for the executable part of code that composes a Java program: control structures and other instructions that are found inside method bodies.

We start by looking at how, in general, we translate method implementations, and then move to branching instructions and loops.

\subsection*{2.3.1 Method implementation}

Every method implementation \( M_J \) should start by invoking \( \text{MAKEREF} \) on all the parameters \( x_i \) that are references to objects. Upon returning a value, or the closing of a scope, we also need to remember to invoke \( \text{FREEREF} \).

Instructions are usually “chained” by synchronizing a series of parallel operations on the return value of their preceding one, so that they effectively run in sequence within the same execution context.

Let us define an overloaded macro to help us in this kind of chaining.

\[
\forall \sigma_1 \leq j \leq m \in \text{Instructions}_{M_J}:
\]

\[
\#\text{EXEC}() \overset{\text{def}}{=} 0
\]

\[
\#\text{EXEC}(\sigma_i, \bar{\sigma}_{j>i}) \overset{\text{def}}{=} \text{ret}_{i-1}(v_{i-1}).\text{vret}_{i}.(\sigma_i | \#\text{EXEC}_{\sigma}[\bar{\sigma}_j])
\]
Then, translating a method $M_j$ results in something akin to (the instruction set $\tilde{\sigma}$ has been put in evidence in red):

\[
M_j(ctx, o, e, ret, \tilde{\alpha}) = \\
vunwind. (vret_0.\overline{ret}_0(\star). \\
\#EXEC_{\sigma} \{ \{ \text{MAKEREF}[x_j, ret_i] : 0 \leq j \leq n \land x_j \in \tilde{\alpha} \text{ is a reference} \}, \tilde{\sigma} \} \\
| \text{unwind}(\star).\overline{vdone}. \prod_{\text{ref}_j \in \text{Refs}} (\text{FREEREF}[\text{ref}_j, done] | done(\star)))
\]

where $\text{Refs} = \{ \text{ref} : \text{ref} \in \tilde{\alpha} \cup \mathcal{R}(\#EXEC[\tilde{\sigma}]) \land \text{ref} \text{ is a reference} \}$.

That is, we first increase the reference count for all parameters that are references to objects, we execute all instructions sequentially, and then we finally decrease the count for those references. This is done when we encounter a $\text{return var}$ statement within the set of instructions $\tilde{\sigma}$, which will translate in something like:

\[
\text{ret}_{i-1}(v_{i-1}). (\overline{\text{ret}} \langle \text{var} \rangle | \text{unwind}(\star))
\]

If there are other scoped blocks inside the method body that e.g. assign a variable which is a parameter, we still want to free the same names at the end, so unwinding always has to happen. A block will create a new scope, so at its end $\text{FREEREF}$ will need to be called on those objects that have been instantiated within the block itself.

### 2.3.2 Assignment

Assigning values relies on the caller to receive over a $\text{ret}$ channel that binds a parameter which shadows the previous name, since in $\pi$-calculus we do not have side-effects.

We here give the translation for an assignment between references, since it requires to increase the reference counter of the new value, and decrease the one of the old associated value.

\[
\text{ASSIGN}_{OBJ}(lhs, rhs, ret) = \\
vdone. (\text{MAKEREF}[rhs, done] \\
| done(\star).vdone_2. (\text{FREEREF}[lhs, done_2] \\
| done_2(\star).\overline{\text{ret}}(rhs))
\]

Thus, if we want to change the value of a certain reference represented by $a$ to instead point to the same object referenced to $b$, we would call:

\[
\pi.\overline{\text{ret}}.\text{ASSIGN}_{OBJ}[a, b, ret] | \text{ret}(a).P
\]

where $a$ is already known in $\pi$, and it is replaced by substitution by $b$ in the rest of our scope $P$. 

Branching and loops

Branching and loop statements inherently work, in a programming language like Java, upon taking a deterministic choice about which branch to take.

In our $\pi$-calculus, we want only to make sure that, when looking at all possible runtime configurations of a program, all branches can be evaluated, so that we can catch deadlocks or other concurrency-related problems. Thus, it is not necessary to exactly determine the boolean result of evaluating a condition. Just using the choice operator will enable all possible branch to be run. Of course, this will mean we will have some false positives among our results. For instance, in the piece of code shown in Figure 2.4, the first branch will never execute. Any related problem which we detect during static analysis will never result in bad behaviour at runtime. However, most static analysis tools produce a number of false positives, and we believe that this kind of false problems we are able to detect do not pose a significant hindrance.

That said, translating the main control structure (if-then-else) of the Java programming language proves to be simple enough. For an $\textbf{if} \ (\text{cond}) \{ \ a; \ \} \ \textbf{else} \ \{ \ b; \ \}$ statement, the corresponding translation might be:

$$\tilde{\sigma}_{\text{cond},\text{vendblock}}. (\tilde{\sigma}_a.\text{endblock}(\ast) + \tilde{\sigma}_b.\text{endblock}(\ast) | \text{endblock}(\ast)...)$$

As already mentioned, also an $\textbf{if}$ block, like it is the case with any other, will need to call $\texttt{FREEREF}$ on local references that go out of scope.
**while** statements are more complex, they just have one branch which might or might not be executed. The obvious thing to do would be to allow a loop like

```java
while (cond){ body; }
```

to run for an undeterministic number of times:

\[ W_{\text{while}}(\bar{x}) \]

\[ \text{vend.while.} \ (\text{RE} \text{C} \text{W} \text{H} \text{I} \text{L} \text{E} \text{B} \text{L} \text{O} \text{C} \text{K} \ [\text{end}, \text{halter}, \bar{x}] \]

\[ \mid \text{end}(\star).\text{halter}(\star). \ldots) \]

\[ \text{RE} \text{C} \text{W} \text{H} \text{I} \text{L} \text{E} \text{B} \text{L} \text{O} \text{C} \text{K} \ (\text{end}, \text{halter}, \bar{x}) \]

\[ \text{def} = \nu \text{repeat.} \ (\bar{\sigma}_\text{cond}.\bar{\sigma}_\text{body} \overline{\text{end}(\star)} + \overline{\text{repeat}(\star)} \]

\[ \mid (\text{repeat}(\star).\text{RE} \text{C} \text{W} \text{H} \text{I} \text{L} \text{E} \text{B} \text{L} \text{O} \text{C} \text{K} \ [\text{end}, \text{halter}, \bar{x}] ) + \text{halter}(\star) ) \]

However, this might lead to an unbounded translation: imagine for instance a loop which creates a new list node, and always inserts it at as the head of its container object. In Java, if we execute the loop body a finite, and small enough, number of times, we do not run out of memory.

However, if the loop can run an unbounded number of times in our translation, it will continuously create new names for objects (in this case, list nodes), without ever forgetting their names, since they are set to fields of other, external objects (so they do not go out of scope when the block ends). So, this route cannot be taken.

The alternative would be to consider each **while** loop as if it is run either 0 or 1 times. Then, we would obtain:

\[ \#W_{\text{while}}(\bar{\sigma}_\text{cond},\bar{\sigma}_\text{body}) \]

\[ \text{vendblock.} \ (\bar{\sigma}_\text{cond}.\text{endblock}(\star) + (\bar{\sigma}_\text{body}.\text{endblock}(\star)) \]

\[ \mid \text{endblock}(\star). \ldots) \]

But what if the loop, for instance, spawns a number of threads? And what if some deadlock or other issue can only be reached when the loop is ran more than \( k \) times?

Since we do not want to fall back into the realm of implementing a Turing-complete translation, we apply the solution of asking the user to give a global parameter to the translation, \$\text{loop}_k \in \mathbb{N}^+\). Our \#W_{\text{while}} definition will then be unrolled to be executed up to \$\text{loop}_k \) times. The user can run the translation multiple times with increasing bounds if they want to make sure properties still hold.
\#\textsc{whileblock}\ ((\bar{\sigma}_\text{cond}, \bar{\sigma}_\text{body}) \overset{\text{def}}{=} \\
\text{vendblock}. (\#\textsc{recwhileblock}[\bar{\sigma}_\text{cond}, \bar{\sigma}_\text{body}].\#\textsc{recwhileblock}\ (\$\text{loop}_k-1)) \\
| \text{endblock}(\star). \ldots)

\#\textsc{recwhileblock}\ ((\bar{\sigma}_\text{cond}, \bar{\sigma}_\text{body}, \text{car}, \overline{\text{cdr}}) \overset{\text{def}}{=} \\
\bar{\sigma}_{\text{cond}}.\text{vendblock}(\star) + (\bar{\sigma}_{\text{body}}.\text{car}[\bar{\sigma}_{\text{cond}}, \bar{\sigma}_{\text{body}}, \overline{\text{cdr}}]))

\#\textsc{recwhileblock}\ ((\bar{\sigma}_\text{cond}, \bar{\sigma}_\text{body}) \overset{\text{def}}{=} \\
\bar{\sigma}_{\text{cond}}.\text{vendblock}(\star) + (\bar{\sigma}_{\text{body}}.\text{vendblock}(\star))

Note the last parameter to the \#\textsc{recwhileblock} invocation in \#\textsc{whileblock}. It is \#\textsc{recwhileblock} itself \$\text{loop}_k\ - \ 1\ times.

The net effect is, \#\textsc{recwhileblock} will be expanded \$\text{loop}_k\ times. The last time the recursive expansion will happen with an empty \text{cdr}, and thus will result in its second, overloaded version. At each step, we have a chance that the loop might terminate after running \(\bar{\sigma}_{\text{cond}}\).

\begin{verbatim}
do { body; } while (cond) loops are guaranteed to always be executed at least once. Hence, we can just invert \(\bar{\sigma}_{\text{body}}, \bar{\sigma}_{\text{cond}}\), and then jump to the beginning again, at most \$\text{loop}_k\ times.

for statements can be traced to \textit{while} statements quite easily, since they are just syntactic sugar on top of them. Thus, we consider \texttt{for (init; cond; inc)} \{ body; \} as being the same as:

```java
init;
while (cond)
{
   body;
   inc;
}
```

The enhanced form of \texttt{for} statements found in Java, namely \texttt{for (ObjClass o: container){ body; \}} can be in turn rewritten as:
They also allow for a version over arrays, which does not use iterators but just the indexes up to the array size. We omit the translation here, since it is trivial.

2.3.4 Exception handling

We only give a partial translation of exception semantics here. We are not particularly interested in providing full coverage in our translation at the time, also because proper exception handling requires to model the stack of the application so that it can be unwound.

We let each try block to create a new exception channel to be passed to methods called inside the block itself (the global exception channel Ξ will be introduced with the description of the starting agent).

Matching the right exception when multiple catch clauses are present can be slightly difficult because of polymorphism, but the number of classes in a hierarchy is limited, so we can still use the choice operator to match the exception type. But even multiple catch statements will still start by listening over the exception channel for the exception object reference. Then, given a try { a; } catch (T e){ b; } statement, we have something like:

\[\nuexchann\nuoex\nuendblock.\]

\[\begin{aligned}
    (\sigma_{a,\nnoex}(\star) & | (\nnoex(\star).\nendblock(\star) + (\nnoex(e).\sigma_{b,\nendblock(\star)})) \\
    | \nendblock(\star). ...)
\end{aligned}\]

Throwing an exception, then, is just a matter of:

\[\text{THROW} (exchannel, exclass) \overset{def}{=} vret. (\text{makeobj}[ret, exclass] | ret(exception).exchannel(exception))\]

The most obvious limitation of this method is that, when long-jumping to the corresponding exception channel, it does not automatically clean up references in stack frames between the current one and the one where the catch block was defined. This contributes to the name bound, and can potentially make the translation unbounded.

A more powerful handling of exceptions is deferred to future work, which will have to take stack frames in account. For now, we are content to simulate a subset of Java functionality that only detects exceptions that are caught by the global exception channel Ξ, causing execution to abort.
2.3.5 Spawning threads

In Java, a new thread is created by instantiating an object of a subclass of Thread, and calling the `start()` method on it. This will in turn run some native code, spawn a new execution context, and invoke the `run()` method. If `run()` was not overridden, it will delegate the `run()` method of a `Runnable` instance passed as the constructor parameter.

When the user creates an object of a subclass of `Thread`, and calls `start()`, we handle this as an asynchronous method:

\[
\text{THREAD}_{\text{START}}(ctx, \text{this}, e, ret) \overset{\text{def}}{=} \\
\text{MAKEREF}[\text{this}, ret] \\
| \nu ret_2.\nu done.\nu ctx_2. \ (#\text{INVOKE}[ctx_2, \text{this}, e, [\text{run}], ret_2] \\
| \nu ret_2.\nu done. \ (#\text{FREEREF}[	ext{this}, done] \\
| \text{done}(\star))
\]

Since `Thread` objects will return asynchronously from their `start()` method, also the implicit parameter `this` needs to have its reference count incremented.

2.3.6 Synchronization

The most common way to protect critical sections, in Java, is to use monitors with a defined scope, which take a lock and release it at their end. This protects somewhat the programmer from forgetting to release the lock manually. Other languages have solved this problem in analogue ways; for instance, in C++, one can define lock objects, and rely on RAII semantics and the destructor to free the lock.

We have already introduced the topic of locking in Section 2.2.4. The `synchronized` keyword in Java is their complement for taking ownership of a lock inside the code.

\[
#\text{SYNCHRONIZED}_{\text{BLOCK}}(ctx, obj, \text{BODY}) \overset{\text{def}}{=} \\
\text{obj}(\text{class}, \text{data}, \text{refcount}, ctxlock).\nu \text{blockguard}. ( \\
\overline{ctxlock}(ctx).ctx(lock).\text{blockguard}(\star).\text{blockguard}(\star).\text{lock}(\star) \\
| \text{blockguard}(\star).\{\text{BODY}\}.\text{blockguard}(\star))
\]

2.4 The starting agent

Before going on and giving the agent for representing a full program, we define Ξ as the `system exception channel`.
This is the channel in use by the Java Virtual Machine to listen for unhandled exceptions, and it is passed as the initial exception channel, for instance through the static `main` method of a class. Errors caught by this \( \Xi \) are those that will cause execution to abort.

After having done so, it is time to define our main entry point, \(#PROGRAMCONFIG\). This agent can be adapted to run more than one program in parallel in the same virtual machine: we are free to tailor execution to match our needs (e.g. invoking different methods on the last line), but there are some processes that need to be running because they emulate the initial setup done by the JVM, or because they need to be always responsive for the whole time of execution.

We therefore represent a program’s execution as:

\[
#PROGRAMCONFIG (\tilde{M}) \overset{def}{=} \prod_K INITCLASS_K
\]

\[
\mid MAKENULL
\mid ALLGLOBALLOCKS
\mid \Xi(exception)
\mid * (\nu.s.stdin(s))
\mid * stdout(s)
\mid \prod_j (\nuctx.vret. (M_j|ctx,\Xi,ret,initialValues^a_j) \mid ret(\star)))
\]

where \( K \in \text{Classes} \cup \{\text{NullClass}\} \), \( \tilde{M} \) is a set of static methods we would like to call to start executing their code, and \( \forall j, M_j \in \tilde{M} \). A good example is a class’s `main` method implementation. Note that we assume that the set of Java classes is known and bounded at compile time. We do not allow for dynamic class loading (although our translation could be made to work with that, with some adjustments), since it is not a widely relied-on feature by the end-programmer\(^8\).

If needed, global locks and other global resources can be disposed of when the main program exits. We will not do so here when creating our program configuration, because we do not need it for the kind of property checking we want to perform. It is however very much akin to what we do for the object life-cycle management, and implementing it is fairly trivial.

\(^8\)It is mostly in use by frameworks such as OSGi, but the actual configuration can be usually determined beforehand for a set of test runs to be explored.
2.5 An example of translation

It is now time to give a translation for a slightly bigger example, so that we can showcase how what we wrote up until now ties together.

We will go on something “classic”, and provide the translation for the program in Figure 2.5. This code comes directly from the Java tutorial, and is simple enough to be manageable on paper.

We start by giving our corresponding starting agent (already expanded to invoke the main method at start):

\[
\text{EXAMPLECONFIG def } = \\
\text{INITCLASS_DEADLOCK} \\
\text{INITCLASS_DEADLOCK_FRIEND} \\
\text{INITCLASS_THREAD} \\
\text{INITCLASS_ANONYMOUS_1} \\
\text{INITCLASS_ANONYMOUS_2} \\
\text{MAKENULL} \\
\text{ALLGLOBALLOCKS} \\
\Xi(\text{exception}) \\
\nu. \text{stdin}(s) \\
\nu. \text{stdout}(s) \\
\nu \text{ctx. } \nu \text{ret. (MAIN_DEADLOCK} | \text{ctx}, \Xi, \text{ret}, \text{null} \\
\text{ | ret(\star)})
\]

We will call this agent when we want to start the actual execution.

In order to make the translation slightly more readable, we use the $ character as a sign that its right-hand side is to be enclosed in parenthesis, as it happens in Haskell. Then, for instance, \( \nu a. (a(c) | \nu b. (b(a) | a(a))) \) can be rewritten as: \( \nu a. $ a(c) | \nu b. $ b(a) | a(a) \).

We also use quoted strings ("string") as names marking interned strings that appear as literals in the source code, without resorting to create a separate temporary object each time they are used in-text.

The translation for MAIN_DEADLOCK is shown in Figure 2.6 at page 55. Some things to note:

1. we have two inlined, <Anonymous> classes that implements Runnable.
2. the constructor for the anonymous classes is the one from Object, since Runnable is only an interface, so we omit it.
public class Deadlock {
    static class Friend {
        private final String name;
        public Friend (String name) {
            this.name = name;
        }
        public String getName () {
            return this.name;
        }
        public synchronized void bow (Friend bower) {
            System.out.format ("%s: %s
+ "%s: %s",
            this.name, bower.getName ());
            bower.bowBack (this);
        }
        public synchronized void bowBack (Friend bower) {
            System.out.format ("%s: %s
+ "%s: %s",
            this.name, bower.getName ());
        }
    }
    public static void main (String[] args) {
        final Friend alphonse = new Friend ("Alphonse");
        final Friend gaston = new Friend ("Gaston");
        new Thread (new Runnable() {
            public void run() {
                alphonse.bow (gaston);
            }
        }).start ();
        new Thread (new Runnable() {
            public void run() {
                gaston.bow (alphonse);
            }
        }).start ();
    }
}

Figure 2.5: An example of a program which might end in deadlock, taken from http://docs.oracle.com/javase/tutorial/essential/concurrency/deadlock.html.
3. final fields are visible inside the body of inline anonymous classes. To emulate this behaviour, we set them as fields inside them.

4. we know implicitly that Thread is going to keep a reference to a Runnable object, and by default use it inside its start() method to invoke the run() method of the latter object. Instead of giving the full translation for the Thread class, we only work on this assumption.

5. The last line of our main method is considered to be an extra and implicit return; statement.

Class Deadlock has a simple version of initclass, which has no static data:

\[
\text{INITCLASS}_{\text{DEADLOCK}} \overset{\text{def}}{=} \text{validator}.\text{vtable}.\text{vdata}. \\
(\text{\texttt{DeadLock}}(\text{allocator}, \text{Object}, \text{vtable}, \text{sdata}) \\
| \text{ALLOCATE}_{\text{DEADLOCK}}[\text{allocator}] \\
| \text{METHODS}_{\text{DEADLOCK}}[\text{vtable}] \\
| \text{vnotused}.\text{FIELDOPS}_{\text{DEADLOCK, STATIC}}[\text{sdata}, \text{notused}] )
\]

We forego giving \text{ALLOCATE}_{\text{DEADLOCK}} since it bears no differences to \text{ALLOCATE}_{K}, except that it invokes \text{FIELDOPS}_{\text{DEADLOCK}}. However, Deadlock has no fields, so we can shorten it to:

\[
\text{FIELDOPS}_{\text{DEADLOCK}}(\text{data}, \text{dealloc}, \vec{x}) \overset{\text{def}}{=} \text{dealloc}(\star)
\]

Then we have class Thread to take care of.

\[
\text{INITCLASS}_{\text{THREAD}} \overset{\text{def}}{=} \text{validator}.\text{vtable}.\text{vdata}. \\
(\text{\texttt{Thread}}(\text{allocator}, \text{Object}, \text{vtable}, \text{sdata}) \\
| \text{ALLOCATE}_{\text{THREAD}}[\text{allocator}] \\
| \text{METHODS}_{\text{THREAD}}[\text{vtable}] \\
| \text{vnotused}.\text{FIELDOPS}_{\text{THREAD, STATIC}}[\text{sdata}, \text{notused}] )
\]

We assume Thread has a just a private runnable field of type Runnable, which is enough for this example. As before, we skip giving a definition for
where LocalRefs := \{\texttt{alphonse, gaston, anon}\textsubscript{Runnable\_1}, anon\textsubscript{Runnable\_2}, anon\textsubscript{Thread\_1}, anon\textsubscript{Thread\_2}\}.

Figure 2.6: The main method in our sample translation
ALLOCATE\_THREAD; focusing on FIELDOPS\_THREAD instead.

FIELDOPS\_THREAD (data, dealloc, runnable) \( \text{def} \) =

\( \) dealloc(\( \star \)) . RELEASEOBJ\_FIELDS\_THREAD[\( \bar{x} \)] +

\( \) (data(op, f, v, e), \lambda_{\text{fields}}(\( \star \)).(\( \sigma_f \)(f, v) |

\( \) get(f, ret). (\( \bar{f} \)(ret) \mid \@\text{runnable}(ret).\lambda_{\text{fields}}(\( \star \)).

\( \) \( \bar{r} \)(runnable) \mid FIELDOPS\_THREAD[data, dealloc, runnable])))

\( + \) set(f, v). (\( \bar{f} \)(v) \mid \@\text{runnable}(v).\lambda_{\text{fields}}(\( \star \)).

\( \) FIELDOPS\_THREAD[data, dealloc, runnable])))

By now, the translation of INITCLASS\_DEADLOCK\_FRIEND, INITCLASS\_ANONYMOUS\_1, and INITCLASS\_ANONYMOUS\_2 should be clear, so we will skip it. Just remember that Deadlock.Friend has a field named \@name, Anonymous\_1 has a field named \@alphonse, and Anonymous\_2 a field named \@gaston.

We will give just the virtual table implementation for DEADLOCK\_FRIEND. The one for Thread contains the start and run methods, while the one for DEADLOCK has no dynamic methods\(^9\).

METHODS\_DEADLOCK\_FRIEND(vtable) \( \text{def} \) = vtable(e).

\( ([\text{name}](ctx, o, e, ret).

\( \lambda_{\text{invoke}}(\bar{x}) . (\text{METHODS\_DEADLOCK\_FRIEND[vtable]}

\( \mid \text{GETNAME\_DEADLOCK\_FRIEND[ctx, o, e, ret]})

\) + ([\text{bow}](ctx, o, e, ret, bower).

\( \lambda_{\text{invoke}}(\bar{x}) . (\text{METHODS\_DEADLOCK\_FRIEND[vtable]}

\( \mid \text{BOW\_DEADLOCK\_FRIEND[ctx, o, e, ret, bower]})

\) + ([\text{bowBack}](ctx, o, e, ret, bower).

\( \lambda_{\text{invoke}}(\bar{x}) . (\text{METHODS\_DEADLOCK\_FRIEND[vtable]}

\( \mid \text{BOWBACK\_DEADLOCK\_FRIEND[ctx, o, e, ret, bower]})

Thread’s start() implementation has already been given in Section 2.3.5. Its run() method implementation will just invoke the run() method of its runnable field, whose existence we postulated before.

We now have a look at Deadlock.Friend’s constructor. Thread’s constructor is on the same lines, as it will only need to set its runnable field to its only actual

---

\(^9\)Actually, a complete translation, with access to the JDK source code, would need to take Object’s public methods into account.
parameter’s value. We do not bump the reference count on its argument because we assume it is a literal string that has been interned – and not “real” reference to a \texttt{String} object. This in general is not true (we should really create a temporary, local \texttt{String} object inside the \texttt{main} method, and then increase its \texttt{refcount}), but doing otherwise here would only make our example a bit more cumbersome, without adding extra value.

\begin{align*}
\text{\texttt{FRIEND}_DEADLOCK\_FRIEND}(\texttt{ctx, this, e, ret, name}) & \overset{\text{def}}{=} $ \\
& \quad \text{\texttt{vret}_0.} \text{\texttt{\$ SE\textsc{tFIELD}_{OBJ}[this, @name, name, ret]}} \\
& \quad \mid \text{\texttt{ret}_0(\texttt{\$})}, \text{\texttt{\$}}
\end{align*}

We have omitted \texttt{unwind} as it was not necessary.

Let us now translate also the \texttt{bow} method. The \texttt{bowBack} method is actually the same, except it lacks the last line, so it should be trivial to translate.

\begin{align*}
\text{\texttt{BOW}_DEADLOCK\_FRIEND}(\texttt{ctx, this, e, ret, bower}) & \overset{\text{def}}{=} \text{\texttt{unwind.}} $ \\
& \quad \text{\texttt{vret}_0.} \text{\texttt{\$ MAKE\textsc{EREF}[bower, ret]}} \\
& \quad \mid \text{\texttt{vret}_1.} \text{\texttt{\$ \texttt{ret}_0(\texttt{\$})}.\texttt{this(class, data, refcount, ctxlock)}.vblockguard $} \\
& \quad \mid \text{\texttt{ctxlock(\texttt{ctx}).ctx(lock).blockguard(\texttt{\$})}, \text{\texttt{blockguard(\texttt{\$})}}}.\text{\texttt{lock(\texttt{\$})}} \\
& \quad \mid \text{\texttt{blockguard(\texttt{\$})}.\texttt{GET\textsc{FIELD}_{OBJ}[this, @name, ret]}} \\
& \quad \mid \text{\texttt{vret}_2.} \text{\texttt{\$ \texttt{ret}_1(\texttt{immediate}_1)}.\#\texttt{IN\textsc{VOKE}[ctx, bower, e, [get\texttt{Name}, ret]}} \\
& \quad \mid \text{\texttt{vret}_3.} \text{\texttt{\$ \texttt{ret}_2(\texttt{immediate}_2)}.\#\texttt{FORMAT}_{SYSTEM\_OUT}[ctx, e, ret, 3, \texttt{\"%s has bowed to me! \%n\texttt{\"}}, \texttt{\texttt{immediate}_1, immediate}_2]} \\
& \quad \mid \text{\texttt{vret}_4.} \text{\texttt{\$ \texttt{ret}_3(\texttt{\$}).FRE\textsc{EREF}[\texttt{immediate}_1, ret]}} \\
& \quad \mid \text{\texttt{vret}_5.} \text{\texttt{\$ \texttt{ret}_4(\texttt{\$}).FRE\textsc{EREF}[\texttt{immediate}_2, ret]}} \\
& \quad \mid \text{\texttt{vret}_6.} \text{\texttt{\$ \#\texttt{IN\textsc{VOKE}[ctx, bower, e, \texttt{[bowBack], ret, this]}}} \\
& \quad \mid \text{\texttt{ret}_0(\texttt{\$}).\texttt{blockguard(\texttt{\$})}.(\texttt{\$ret(\texttt{\$})} \mid \texttt{unwind(\texttt{\$})}} \\
& \quad \mid \text{\texttt{unwind(\texttt{\$})}.vdone.\texttt{FRE\textsc{EREF}[bower, done]} \mid \texttt{done(\texttt{unit})}}
\end{align*}

Finally, let us give a simplified implementation of the static \texttt{System.out}. \texttt{format} function, just enough to make our translation work – even if its output will not be the same as in Java:

\begin{align*}
\#\text{\texttt{FORMAT}}_{\text{\texttt{SYSTEM\_OUT}}}(\texttt{ctx, e, ret, format, \$}) & \overset{\text{def}}{=} $ \\
& \quad \text{\texttt{\$ stdout(format).stdout(s}_1).stdout(s}_2)\ldots \text{stdout(s}_n)\texttt{\$ret(\$)}
\end{align*}
From $\pi$-calculus to Petri Nets

A diagram is a graphic shorthand. Though it is an ideogram, it is not necessarily an abstraction. It is a representation of something in that it is not the thing itself. In this sense, it cannot help but be embodied. It can never be free of value or meaning, even when it attempts to express relationships of formation and their processes. At the same time, a diagram is neither a structure nor an abstraction of structure.

— Peter Eisenman, “Written Into the Void”, ch. 11, 2007

The steps needed to pass from our translation to the corresponding Petri Net, as detailed in [P4], are: (i) convert from the traditional $\pi$-calculus representation we used in Chapter 2 to identity-aware processes, (ii) guess a bound on names $b$, (iii) if the process is name-bounded by $b$, we can compute the bound $p$, where $p$ is the largest number of sequential processes that knows an instance $(a, i)$ in a limit process of the KM-tree:

$$p := \max \left\{ |L|_{(a, i)} \mid L \in \text{KM} (P_{ia}) \land (a, i) \notin \text{Inf} (L) \right\},$$

$$\text{Inf} (L) := \{S \in S : L (S) = \omega\}$$

(iv) construct the Petri Net $N (P_{ia}, b, p)$.

Let us assume we are constructing the Petri Net some processes sporting locks, such as those we use for objects. We will now work on a simplified version of the example seen in the previous chapter, which will enable us to better focus on the translation without rendering it incomprehensible on paper.

We have two objects $o_1$ and $o_2$, and the $\text{bow}$ and $\text{bowBack}$ methods:

$$\nu o_1 . o_2 . \nu l_1 . \nu l_2 . (\text{OBJLIFE} [o_1, l_1] \mid \text{OBJLIFE} [o_2, l_2] \mid \text{MAKELOCK} [l_1] \mid \text{MAKELOCK} [l_2] \mid \nu ret_1 . (\text{bow} [o_1, ret_1, o_2] \mid ret_1 (\star)) \mid \nu ret_2 . (\text{bow} [o_2, ret_2, o_1] \mid ret_2 (\star)))$$

$$\text{OBJLIFE} (o, l) \overset{\text{def}}{=} \sigma (l). \text{OBJLIFE} [o, l]$$
\[ \text{MAKELOCK}(l) \equiv \nu \text{unlock}.l(\text{unlock}).\text{unlock}(\star).\text{MAKELOCK}[l] \]

\[ \text{BOW}(\text{this}, \text{ret}, \text{bower}) \equiv \nu \text{ret}_1.((\text{this}(l)).l(\text{unlock}).(\text{BOWBACK}[\text{bower}, \text{ret}_1, \text{this}] \mid \text{ret}_1(\star).\overline{\text{unlock}}(\star).\overline{\text{ret}}(\star))) \]

\[ \text{BOWBACK}(\text{this}, \text{ret}, \text{bower}) \equiv \text{this}(l).l(\text{unlock}).\overline{\text{unlock}}(\star).\overline{\text{ret}}(\star) \]

Instead of building the KM-tree in full here, we give just the longest path for \( P_{ia} \overset{*}{\rightarrow}_{ia} \), which incidentally is the one where the deadlock does not take place. We consider \( \star \) being just a free name. Hence (the next reaction is in red):

\[
\begin{array}{c|c}
\text{Path} & \text{Reaction} \\
\hline
P_{ia} & (o_1,0)(l_1,0).\text{OBJLIFE}[(o_1,0),(l_1,0)] \\
& (o_2,0)(l_2,0).\text{OBJLIFE}[(o_2,0),(l_2,0)] \\
& (l_1,0)(\text{unlock},0).\text{unlock}(\star).\text{MAKELOCK}[(l_1,0)] \\
& (l_2,0)(\text{unlock},0).\text{unlock}(\star).\text{MAKELOCK}[(l_2,0)] \\
& (o_1,0)(l).l(\text{unlock}).(\text{BOWBACK}[(o_2,0),(\text{ret}_1,0),(o_1,0)] \\
& \quad \quad \mid (\text{ret}_1,0)(\star).\overline{\text{unlock}}(\star).\overline{\text{ret}}(\star)) \\
& (\text{ret},0)(\star) \\
& (o_2,0)(l).l(\text{unlock}).(\text{BOWBACK}[(o_1,0),(\text{ret}_1,1),(o_2,0)] \\
& \quad \quad \mid (\text{ret}_1,1)(\star).\overline{\text{unlock}}(\star).\overline{\text{ret}}(\star)) \\
& (\text{ret},1)(\star) \\
\hline
P_{ia} \overset{0}{\rightarrow}_{ia} P_1 & (o_1,0)(l_1,0).\text{OBJLIFE}[(o_1,0),(l_1,0)] \\
& (o_2,0)(l_2,0).\text{OBJLIFE}[(o_2,0),(l_2,0)] \\
& (l_1,0)(\text{unlock},0).\text{unlock}(\star).\text{MAKELOCK}[(l_1,0)] \\
& (l_2,0)(\text{unlock},0).\text{unlock}(\star).\text{MAKELOCK}[(l_2,0)] \\
& (l_1,0)(\text{unlock}).(\text{BOWBACK}[(o_2,0),(\text{ret}_1,0),(o_1,0)] \\
& \quad \quad \mid (\text{ret}_1,0)(\star).\overline{\text{unlock}}(\star).\overline{\text{ret}}(\star)) \\
& (\text{ret},0)(\star) \\
& (o_2,0)(l).l(\text{unlock}).(\text{BOWBACK}[(o_1,0),(\text{ret}_1,1),(o_2,0)] \\
& \quad \quad \mid (\text{ret}_1,1)(\star).\overline{\text{unlock}}(\star).\overline{\text{ret}}(\star)) \\
& (\text{ret},1)(\star)
\end{array}
\]

Continued on next page...
\[
\begin{array}{c|c}
\pi\text{-CALCULUS TO PETRI NETS} & 61 \\
\hline
\begin{align*}
\pia & \xrightarrow{\ast} \pi_1 \\
\emptyset & \xrightarrow{} \pi_2 \\
\hline
\pi_1 & [\langle o_1, 0 \rangle, \langle l_1, 0 \rangle, \text{OBJLIFE}].[\langle o_1, 0 \rangle, \langle l_1, 0 \rangle] \\
& \quad | [\langle o_2, 0 \rangle, \langle l_2, 0 \rangle, \text{OBJLIFE}].[\langle o_2, 0 \rangle, \langle l_2, 0 \rangle] \\
& \quad | [\langle l_1, 0 \rangle, \langle unlock, 0 \rangle].[\langle unlock, 0 \rangle] . \text{MAKELOCK}.[\langle l_1, 0 \rangle] \\
& \quad | [\langle l_2, 0 \rangle, \langle unlock, 1 \rangle].[\langle unlock, 1 \rangle] . \text{MAKELOCK}.[\langle l_2, 0 \rangle] \\
& \quad | [\langle l_1, 0 \rangle, \langle unlock \rangle].[\langle bowback, 0 \rangle, \langle ret_1, 0 \rangle, \langle o_1, 0 \rangle] \\
& \quad \quad | (ret_1, 0) . (\text{unlock} \langle \ast \rangle).[\langle ret, 0 \rangle].[\langle \ast \rangle] \\
& \quad | (ret, 0) . (\ast) \\
& \quad | [\langle l_2, 0 \rangle, \langle unlock \rangle].[\langle bowback, 0 \rangle, \langle o_1, 0 \rangle, \langle ret_1, 1 \rangle] \\
& \quad \quad | (ret_1, 1) . (\text{unlock} \langle \ast \rangle).[\langle ret, 1 \rangle].[\langle \ast \rangle] \\
& \quad | (ret, 1) . (\ast) \\
\hline
\pi_2 & [\langle o_1, 0 \rangle, \langle l_1, 0 \rangle, \text{OBJLIFE}].[\langle o_1, 0 \rangle, \langle l_1, 0 \rangle] \\
& \quad | [\langle o_2, 0 \rangle, \langle l_2, 0 \rangle, \text{OBJLIFE}].[\langle o_2, 0 \rangle, \langle l_2, 0 \rangle] \\
& \quad | [\langle l_1, 0 \rangle, \langle unlock, 0 \rangle].[\langle unlock, 0 \rangle] . \text{MAKELOCK}.[\langle l_1, 0 \rangle] \\
& \quad | [\langle l_2, 0 \rangle, \langle unlock, 1 \rangle].[\langle unlock, 1 \rangle] . \text{MAKELOCK}.[\langle l_2, 0 \rangle] \\
& \quad | [\langle l_1, 0 \rangle, \langle unlock \rangle].[\langle bowback, 0 \rangle, \langle ret_1, 0 \rangle, \langle o_1, 0 \rangle] \\
& \quad \quad | (ret_1, 0) . (\text{unlock} \langle \ast \rangle).[\langle ret, 0 \rangle].[\langle \ast \rangle] \\
& \quad | (ret, 0) . (\ast) \\
& \quad | [\langle o_1, 0 \rangle, \langle l_1, 0 \rangle, \text{OBJLIFE}].[\langle o_1, 0 \rangle, \langle l_1, 0 \rangle] \\
& \quad | [\langle o_2, 0 \rangle, \langle l_2, 0 \rangle, \text{OBJLIFE}].[\langle o_2, 0 \rangle, \langle l_2, 0 \rangle] \\
& \quad | [\langle l_1, 0 \rangle, \langle unlock, 0 \rangle].[\langle unlock, 0 \rangle] . \text{MAKELOCK}.[\langle l_1, 0 \rangle] \\
& \quad | [\langle l_2, 0 \rangle, \langle unlock, 1 \rangle].[\langle unlock, 1 \rangle] . \text{MAKELOCK}.[\langle l_2, 0 \rangle] \\
& \quad | [\langle l_1, 0 \rangle, \langle unlock \rangle].[\langle bowback, 0 \rangle, \langle ret_1, 0 \rangle, \langle o_1, 0 \rangle] \\
& \quad \quad | (ret_1, 0) . (\text{unlock} \langle \ast \rangle).[\langle ret, 0 \rangle].[\langle \ast \rangle] \\
& \quad | (ret, 0) . (\ast) \\
& \quad | [\langle o_1, 0 \rangle, \langle l_1, 0 \rangle, \text{OBJLIFE}].[\langle o_1, 0 \rangle, \langle l_1, 0 \rangle] \\
& \quad | [\langle o_2, 0 \rangle, \langle l_2, 0 \rangle, \text{OBJLIFE}].[\langle o_2, 0 \rangle, \langle l_2, 0 \rangle] \\
& \quad | [\langle l_1, 0 \rangle, \langle unlock, 0 \rangle].[\langle unlock, 0 \rangle] . \text{MAKELOCK}.[\langle l_1, 0 \rangle] \\
& \quad | [\langle l_2, 0 \rangle, \langle unlock, 1 \rangle].[\langle unlock, 1 \rangle] . \text{MAKELOCK}.[\langle l_2, 0 \rangle] \\
& \quad | [\langle l_1, 0 \rangle, \langle unlock \rangle].[\langle bowback, 0 \rangle, \langle ret_1, 0 \rangle, \langle o_1, 0 \rangle] \\
& \quad \quad | (ret_1, 0) . (\text{unlock} \langle \ast \rangle).[\langle ret, 0 \rangle].[\langle \ast \rangle] \\
& \quad | (ret, 0) . (\ast) \\
& \quad | [\langle l_1, 0 \rangle, \langle unlock \rangle].[\langle bowback, 0 \rangle, \langle o_1, 0 \rangle, \langle ret_1, 1 \rangle] \\
& \quad \quad | (ret_1, 1) . (\text{unlock} \langle \ast \rangle).[\langle ret, 1 \rangle].[\langle \ast \rangle] \\
& \quad | (ret, 1) . (\ast) \\
\hline
\end{align*}
\end{array}
\]

Continued on next page...
\[ P_{ia} \xrightarrow{*} P_4 \]
\[ \emptyset \xrightarrow{\pi} P_5 \]

\[
\begin{align*}
&\{o_1,0\}\langle(l_1,0)\rangle.OBJLIFE[(o_1,0),(l_1,0)] \\
&\{o_2,0\}\langle(l_2,0)\rangle.OBJLIFE[(o_2,0),(l_2,0)] \\
&\langle unlock,0 \rangle \langle \ast \rangle.MAKELOCK[(l_1,0)] \\
&\langle unlock,1 \rangle \langle \ast \rangle.MAKELOCK[(l_2,0)] \\
&\langle l_1,0 \rangle (unlock).BOWBACK[(o_2,0),(\text{ret}_1,0),(o_1,0)] \\
&\quad \langle \text{ret}_1,0 \rangle \langle \ast \rangle.unlock(\ast).(\text{ret},0)\langle \ast \rangle \\
&\langle ret,0 \rangle \langle \ast \rangle \\
&\langle ret_1,1 \rangle \langle \ast \rangle.unlock(\ast).\langle ret,0 \rangle \langle \ast \rangle \\
&\langle ret,1 \rangle \langle \ast \rangle
\end{align*}
\]

\[
\begin{align*}
&\{o_1,0\}\langle(l_1,0)\rangle.OBJLIFE[(o_1,0),(l_1,0)] \\
&\{o_2,0\}\langle(l_2,0)\rangle.OBJLIFE[(o_2,0),(l_2,0)] \\
&\langle unlock,0 \rangle \langle \ast \rangle.MAKELOCK[(l_1,0)] \\
&\langle unlock,1 \rangle \langle \ast \rangle.MAKELOCK[(l_2,0)] \\
&\langle l_1,0 \rangle (unlock).BOWBACK[(o_2,0),(\text{ret}_1,0),(o_1,0)] \\
&\quad \langle \text{ret}_1,0 \rangle \langle \ast \rangle.unlock(\ast).(\text{ret},0)\langle \ast \rangle \\
&\langle ret,0 \rangle \langle \ast \rangle \\
&\langle ret_1,1 \rangle \langle \ast \rangle.unlock(\ast).\langle ret,1 \rangle \langle \ast \rangle \\
&\langle ret,1 \rangle \langle \ast \rangle
\end{align*}
\]

\[
\begin{align*}
&\{o_1,0\}\langle(l_1,0)\rangle.OBJLIFE[(o_1,0),(l_1,0)] \\
&\{o_2,0\}\langle(l_2,0)\rangle.OBJLIFE[(o_2,0),(l_2,0)] \\
&\langle unlock,0 \rangle \langle \ast \rangle.MAKELOCK[(l_1,0)] \\
&\langle unlock,1 \rangle \langle \ast \rangle.MAKELOCK[(l_2,0)] \\
&\langle l_1,0 \rangle (unlock).\langle ret,0 \rangle\langle \ast \rangle \\
&\langle ret,0 \rangle \langle \ast \rangle \\
&\langle ret_1,1 \rangle \langle \ast \rangle.unlock(\ast).\langle ret,0 \rangle \langle \ast \rangle \\
&\langle ret_1,1 \rangle \langle \ast \rangle \\
&\langle ret,1 \rangle \langle \ast \rangle
\end{align*}
\]

Continued on next page...
\[
\begin{array}{|c|c|}
\hline
\pi_{ia} \xrightarrow{*} \pi_{ia} P_7 & (0_1, 0) \langle (l_1, 0) \rangle . OBJLIFE[(0_1, 0), (l_1, 0)] \\
\emptyset \xrightarrow{\_} \pi_{ia} P_8 & (0_2, 0) \langle (l_2, 0) \rangle . OBJLIFE[(0_2, 0), (l_2, 0)] \\
& (unlock, 0) (\star). MAKELOCK[(l_1, 0)] \\
& (unlock, 1) (\star). MAKELOCK[(l_2, 0)] \\
& (l_2, 0) (unlock). unlock(\star). (ret_1, 0) (\star) \\
& (ret_1, 0) (\star). (unlock, 0) (\star). (ret, 0) (\star) \\
& (ret, 0) (\star) \\
& (ret_1, 1) (\star) \\
& (ret_1, 1) (\star). (unlock, 1) (\star). (ret, 1) (\star) \\
& (ret, 1) (\star) \\
\hline
\pi_{ia} \xrightarrow{*} \pi_{ia} P_8 & (0_1, 0) \langle (l_1, 0) \rangle . OBJLIFE[(0_1, 0), (l_1, 0)] \\
\emptyset \xrightarrow{\_} \pi_{ia} P_9 & (0_2, 0) \langle (l_2, 0) \rangle . OBJLIFE[(0_2, 0), (l_2, 0)] \\
& (unlock, 0) (\star). MAKELOCK[(l_1, 0)] \\
& (unlock, 1) (\star). MAKELOCK[(l_2, 0)] \\
& (l_2, 0) (unlock). unlock(\star). (ret_1, 0) (\star) \\
& (ret_1, 0) (\star). (unlock, 0) (\star). (ret, 0) (\star) \\
& (ret, 0) (\star) \\
& (unlock, 1) (\star). (ret, 1) (\star) \\
& (ret, 1) (\star) \\
\hline
\pi_{ia} \xrightarrow{\_} \pi_{ia} P_9 & (0_1, 0) \langle (l_1, 0) \rangle . OBJLIFE[(0_1, 0), (l_1, 0)] \\
\{(unlock, 1)\} \xrightarrow{\_} \pi_{ia} P_{10} & (0_2, 0) \langle (l_2, 0) \rangle . OBJLIFE[(0_2, 0), (l_2, 0)] \\
& (unlock, 0) (\star). MAKELOCK[(l_1, 0)] \\
& (l_2, 0) (unlock). unlock(\star). (ret_1, 0) (\star) \\
& (ret_1, 0) (\star). (unlock, 0) (\star). (ret, 0) (\star) \\
& (ret, 0) (\star) \\
& (ret, 1) (\star) \\
& (ret, 1) (\star) \\
\hline
\end{array}
\]
Continued on next page...
$$\begin{array}{c|c}
\pi_{ia} \xrightarrow{\ast} \pi_{10} & (o_1,0)\langle(l_1,0)\rangle.\text{OBJLIFE}\[(o_1,0),(l_1,0)\] \\
\emptyset \xrightarrow{\gamma} \pi_{11} & (o_2,0)\langle(l_2,0)\rangle.\text{OBJLIFE}\[(o_2,0),(l_2,0)\] \\
& (\text{unlock},0)(*)\text{.MAKELOCK}[l_1,0] \\
& (l_2,0)\langle(\text{unlock},1)\rangle.\text{unlock}(*)\text{.MAKELOCK}[l_2,0] \\
& (\text{unlock},0)(*)\text{.unlock}(0)(*) \langle\text{ret}_1,0\rangle(*) \\
& (\text{ret}_1,0)(*)\langle\text{unlock},0\rangle(*) \langle\text{ret},0\rangle(*) \\
& (\text{ret},0)(*) \\
\pi_{ia} \xrightarrow{\ast} \pi_{11} & (o_1,0)\langle(l_1,0)\rangle.\text{OBJLIFE}\[(o_1,0),(l_1,0)\] \\
\emptyset \xrightarrow{\gamma} \pi_{12} & (o_2,0)\langle(l_2,0)\rangle.\text{OBJLIFE}\[(o_2,0),(l_2,0)\] \\
& (\text{unlock},0)(*)\text{.MAKELOCK}[l_1,0] \\
& (\text{unlock},1)(*)\text{.MAKELOCK}[l_2,0] \\
& (\text{unlock},1)(*)\langle\text{ret}_1,0\rangle(*) \\
& (\text{ret}_1,0)(*)\langle\text{unlock},0\rangle(*) \langle\text{ret},0\rangle(*) \\
& (\text{ret},0)(*) \\
\pi_{ia} \xrightarrow{\ast} \pi_{12} & (o_1,0)\langle(l_1,0)\rangle.\text{OBJLIFE}\[(o_1,0),(l_1,0)\] \\
\{\text{unlock},1\} \xrightarrow{\gamma} \pi_{13} & (o_2,0)\langle(l_2,0)\rangle.\text{OBJLIFE}\[(o_2,0),(l_2,0)\] \\
& (\text{unlock},0)(*)\text{.MAKELOCK}[l_1,0] \\
& (l_2,0)\langle(\text{unlock},1)\rangle.\text{unlock}(*)\text{.MAKELOCK}[l_2,0] \\
& (\text{ret}_1,0)(*) \\
& (\text{ret}_1,0)(*)\langle\text{unlock},0\rangle(*) \langle\text{ret},0\rangle(*) \\
& (\text{ret},0)(*) \\
\pi_{ia} \xrightarrow{\ast} \pi_{13} & (o_1,0)\langle(l_1,0)\rangle.\text{OBJLIFE}\[(o_1,0),(l_1,0)\] \\
\emptyset \xrightarrow{\gamma} \pi_{14} & (o_2,0)\langle(l_2,0)\rangle.\text{OBJLIFE}\[(o_2,0),(l_2,0)\] \\
& (\text{unlock},0)(*)\text{.MAKELOCK}[l_1,0] \\
& (l_2,0)\langle(\text{unlock},1)\rangle.\text{unlock}(*)\text{.MAKELOCK}[l_2,0] \\
& (\text{unlock},0)(*)\langle\text{ret},0\rangle(*) \\
& (\text{ret},0)(*) \\
\end{array}$$

Continued on next page...
Within this path, we can see that at most three sequential processes will get to know the \((l_i, 0)\) instances; we therefore set \(p = 3\). Also, since the maximum number of active restricted names for any processes reachable from \(P_{ia}\) is 10, we also set \(b = 10\).

We now follow the Petri Net construction given in [P4]. The Petri Net \(N(P_{ia}, b, p)\) is given by the composition of two nets:

\[
N(P_{ia}, b, p) := \text{Syn} \left( \text{Ctrl}(P_{ia}, b) \right) \times \text{Ref}(P_{ia}, b, p)
\]

In our case, \(N(P_{ia}, 10, 3)\) will be formed by:

- the control flow Petri Net \(\text{Ctrl}(P_{ia}, 10)\), which models how reactions take place inside \(\pi\)-calculus.

- the net for reference counters \(\text{Ref}(P_{ia}, 10, 3)\), which enables us correctly allocate new names upon encountering a restriction, and to recycle them if possible (if their reference counter reaches zero). Since Petri Nets do not allow for zero-tests, the way this is implemented is by having two places for each possible instance \((a, k)\), \(0 \leq k \leq b\): one labelled \((a, k)\), and one labelled as its complement \((a, k)\). We maintain the invariant that the number of tokens for all markings \(M\), \(M((a, k)) + M((a, k)) = p\), while the name is bounded. We add a third place for each instance, labelled \((a, k)^\omega\), that will be used when \(p\) is exceeded; this indicates a certain name can never be recycled. We can check if the reference counter is zero, when \((a, k)\) has \(p\) tokens.
• Syn (⋅) joins transitions among complementary send and receive operations. We contextually allow the labelling of transitions with the function:

\[
Lab := \{send(x, y), recv(x, y), \epsilon \mid x, y \in \mathcal{F}(P_{ia}) \cup (\mathcal{R}(P_{ia}) \times [0, b - 1])\}
\]

Transitions marked with \(\epsilon\) are not synchronized. The command function \(com : T \rightarrow \text{Com}^\ast\) allows to assign a sequence of commands to each transition \(T\):

\[
\text{Com} := \{inc(a, k), dec(a, k), alloc(a, k) \mid (a, k) \in \mathcal{R}(P_{ia}) \times [0, b - 1]\}
\]

The construction for Ref is already present and complete in \([P4]\), and thus omitted from here.

We give the full set of derivatives for our example:

\[
\mathcal{D}(P_{ia}) = \{\text{OBJLIFE}[(o_1, 0), (l_1, 0)], \text{OBJLIFE}[(o_2, 0), (l_2, 0)], \overline{\sigma(l)} \cdot \text{OBJLIFE}[o, l], \text{OBJLIFE}[o, l], \text{MAKELOCK}[(l_1, 0)], \text{MAKELOCK}[(l_2, 0)], \overline{l(unlock)} \cdot \text{unlock}(\ast) \cdot \text{MAKELOCK}[][l], \text{unlock}(\ast) \cdot \text{MAKELOCK}[][l], \text{MAKELOCK}[][l], \text{BOW}[(o_1, 0), (ret_1, 0), (o_2, 0)], \text{BOW}[(o_2, 0), (ret_2, 0), (o_1, 0)], \overline{this(l)} \cdot \text{unlock}(\ast) \cdot \text{BOWBACK}[\text{bower}, ret_1, \overline{this}] \mid \text{ret}_1(\ast) \cdot \overline{unlock}(\ast) \cdot \text{rel}(\ast)), \overline{l(unlock)} \cdot \text{BOWBACK}[\text{bower}, ret_1, \overline{this}] \mid \text{ret}_1(\ast) \cdot \overline{unlock}(\ast) \cdot \text{rel}(\ast)) \}, \text{BOWBACK}[\text{bower}, ret_1, \overline{this}], \text{ret}_1(\ast) \cdot \overline{unlock}(\ast) \cdot \overline{rel}(\ast), \overline{unlock}(\ast) \cdot \overline{rel}(\ast), \overline{ret}(\ast), (ret_1, 0)(\ast), (ret_2, 0)(\ast) \}
\]

Allowing for all substitutions \(\Sigma(P_{ia})\), we can then build \(\text{Ctrl}(P_{ia}, b)\). It has a place for every possible sequential process, and for each one of them, for all different possible instances \((a, k) \cup \mathcal{F}(P_{ia}), a \in \mathcal{R}(P)\) and bounded by \(k \leq b\).

In Figure 3.1, we show portion of the Ctrl net for an invocation of bow. Each place would need to have an edge to all transitions corresponding to other substitution for all instances and free names in \(P_{ia}\).
Figure 3.1: Portion of the \texttt{Ctrl} net for an invocation of \textit{BOW}
$com(t)$ references the reference counting actions from Ref. From the figure are still lacking the edges that Synchronize enabled reactions. These are all extra edges between places that are labelled with a process prefixed by a send operation over a channel that is an instance $(a, k)$, and transitions that are in the postset of those places labelled with sequential processes prefixed with a receive operation on the same $(a, k)$. 
Conclusions and Further Research

Life is the art of drawing sufficient conclusions from insufficient premises.

4.1 Conclusions

This thesis has taken the theory of name-boundedness for \( \pi \)-calculus processes developed by Hüchting, Majumdar and Meyer, and worked on giving a corresponding translation from the Java programming language.

The resulting agents and processes can be then further transformed in Petri Nets, and fed to a model checker for the static verification of important properties for concurrent programs.

We have attempted to provide, with our translation, a solution which represent a good subset of the Java language, and that can be extended to include more features without too much hassle, by mimicking some of the internal workings of the JVM.

We have also seen how some features, in particular recursive locks, are hard to represent in our restricted version of \( \pi \)-calculus without alterations to the original program.

On the other hand, things such as preserving polymorphism, maintaining the program state, and resolving the scope of names, were handled successfully in the majority of real-world cases.

An important contribution to keeping the resulting translation name-bounded, comes from the disposal of names associated to objects allocated on the heap. This was implemented by reference counting.

4.2 Ideas for Further Research

In this section, we give some pointers which might be explored by others, in order to elicit new research paths. These points arose while writing this thesis, but were only tangential to its contents, and so were left for others to pick up.
It could be nice to extend the name-bounded theory to higher-order $\pi$-calculus, so that also agents can be sent and received on channels. This is interesting because it would make it easier to model things such as Java RMI, exchanging code among processes.

A different and yet related possibility, is to assign a free name for each agent. Each agent definition is then transformed into a running process, guarded on a receive operation upon this name. Since channels can also be passed as parameters of other send and receive operations, this would allow to effectively emulate passing function names as parameters to other functions, without adding to the name-bound.

When building the Karp-Miller tree for reachable $\pi$-calculus processes, an apparent optimization would be to share nodes and paths while unfolding the tree. Some care has to be taken, though, because subtrees with a common root cannot just trivially be shared: their unfolding depends also on the nodes going from the root $P_{ia}$ to the root of the subtree.

It could prove also interesting to define a number that measures the complexity of a program in terms of the KM-tree’s depth, or on “how far” a name propagates throughout the program. This could complement McCabe’s cyclomatic complexity number in helping understand how much a program is hard to debug (some sort of “debugging complexity number”), because variables/names cross component boundaries or get distributed to other parts of the system rendering it harder to understand to an external observer.

It might prove worthy exploring different definitions of structural congruence, and their impact on the name-bounded theory. E.g. why not letting also $\nu a.b(c).\pi(c) \equiv b(c).\nu a.\pi(c)$? This is a transformation that compilers do quite often; for instance, in C++, the stack frame in some compilers is pushed only once for block by the sum of all variable sizes, even if they are declared at different points inside the block instead than at its beginning (as it is customary in C).
License

Attribution-ShareAlike 3.0 Unported

THE WORK (AS DEFINED BELOW) IS PROVIDED UNDER THE TERMS OF THIS CREATIVE COMMONS PUBLIC LICENSE (“CCPL” OR “LICENSE”). THE WORK IS PROTECTED BY COPYRIGHT AND/OR OTHER APPLICABLE LAW. ANY USE OF THE WORK OTHER THAN AS AUTHORIZED UNDER THIS LICENSE OR COPYRIGHT LAW IS PROHIBITED.

BY EXERCISING ANY RIGHTS TO THE WORK PROVIDED HERE, YOU ACCEPT AND AGREE TO BE BOUND BY THE TERMS OF THIS LICENSE. TO THE EXTENT THIS LICENSE MAY BE CONSIDERED TO BE A CONTRACT, THE LICENSOR GRANTS YOU THE RIGHTS CONTAINED HERE IN CONSIDERATION OF YOUR ACCEPTANCE OF SUCH TERMS AND CONDITIONS.

1 Definitions

a. “Adaptation” means a work based upon the Work, or upon the Work and other pre-existing works, such as a translation, adaptation, derivative work, arrangement of music or other alterations of a literary or artistic work, or phonogram or performance and includes cinematographic adaptations or any other form in which the Work may be recast, transformed, or adapted including in any form recognizably derived from the original, except that a work that constitutes a Collection will not be considered an Adaptation for the purpose of this License. For the avoidance of doubt, where the Work is a musical work, performance or phonogram, the synchronization of the Work in timed-relation with a moving image (“synching”) will be considered an Adaptation for the purpose of this License.
b. “Collection” means a collection of literary or artistic works, such as encyclopedias and anthologies, or performances, phonograms or broadcasts, or other works or subject matter other than works listed in Section 1(f) below, which, by reason of the selection and arrangement of their contents, constitute intellectual creations, in which the Work is included in its entirety in unmodified form along with one or more other contributions, each constituting separate and independent works in themselves, which together are assembled into a collective whole. A work that constitutes a Collection will not be considered an Adaptation (as defined above) for the purposes of this License.

c. “Creative Commons Compatible License” means a license that is listed at http://creativecommons.org/compatiblelicenses that has been approved by Creative Commons as being essentially equivalent to this License, including, at a minimum, because that license: (i) contains terms that have the same purpose, meaning and effect as the License Elements of this License; and, (ii) explicitly permits the relicensing of adaptations of works made available under that license under this License or a Creative Commons jurisdiction license with the same License Elements as this License.

d. “Distribute” means to make available to the public the original and copies of the Work or Adaptation, as appropriate, through sale or other transfer of ownership.

e. “License Elements” means the following high-level license attributes as selected by Licensor and indicated in the title of this License: Attribution, Share-Alike.

f. “Licensor” means the individual, individuals, entity or entities that offer(s) the Work under the terms of this License.

g. “Original Author” means, in the case of a literary or artistic work, the individual, individuals, entity or entities who created the Work or if no individual or entity can be identified, the publisher; and in addition (i) in the case of a performance the actors, singers, musicians, dancers, and other persons who act, sing, deliver, declaim, play in, interpret or otherwise perform literary or artistic works or expressions of folklore; (ii) in the case of a phonogram the producer being the person or legal entity who first fixes the sounds of a performance or other sounds; and, (iii) in the case of broadcasts, the organization that transmits the broadcast.

h. “Work” means the literary and/or artistic work offered under the terms of this License including without limitation any production in the literary, scientific and artistic domain, whatever may be the mode or form of its expression including digital form, such as a book, pamphlet and other writing; a lecture,
address, sermon or other work of the same nature; a dramatic or dramatico-
musical work; a choreographic work or entertainment in dumb show; a musical
composition with or without words; a cinematographic work to which are as-
similated works expressed by a process analogous to cinematography; a work of
drawing, painting, architecture, sculpture, engraving or lithography; a photo-
graphic work to which are assimilated works expressed by a process analogous
to photography; a work of applied art; an illustration, map, plan, sketch or
three-dimensional work relative to geography, topography, architecture or sci-
ence; a performance; a broadcast; a phonogram; a compilation of data to the
extent it is protected as a copyrightable work; or a work performed by a vari-
ety or circus performer to the extent it is not otherwise considered a literary
or artistic work.

i. “You” means an individual or entity exercising rights under this License who
has not previously violated the terms of this License with respect to the Work,
or who has received express permission from the Licensor to exercise rights
under this License despite a previous violation.

j. “Publicly Perform” means to perform public recitations of the Work and to
communicate to the public those public recitations, by any means or process,
including by wire or wireless means or public digital performances; to make
available to the public Works in such a way that members of the public may
access these Works from a place and at a place individually chosen by them; to
perform the Work to the public by any means or process and the communication
to the public of the performances of the Work, including by public digital
performance; to broadcast and rebroadcast the Work by any means including
signs, sounds or images.

k. “Reproduce” means to make copies of the Work by any means including
without limitation by sound or visual recordings and the right of fixation and
reproducing fixations of the Work, including storage of a protected performance
or phonogram in digital form or other electronic medium.

2 Fair Dealing Rights

Nothing in this License is intended to reduce, limit, or restrict any uses free from
copyright or rights arising from limitations or exceptions that are provided for in
connection with the copyright protection under copyright law or other applicable
laws.
3 License Grant

Subject to the terms and conditions of this License, Licensor hereby grants You a worldwide, royalty-free, non-exclusive, perpetual (for the duration of the applicable copyright) license to exercise the rights in the Work as stated below:

a. to Reproduce the Work, to incorporate the Work into one or more Collections, and to Reproduce the Work as incorporated in the Collections;

b. to create and Reproduce Adaptations provided that any such Adaptation, including any translation in any medium, takes reasonable steps to clearly label, demarcate or otherwise identify that changes were made to the original Work. For example, a translation could be marked “The original work was translated from English to Spanish,” or a modification could indicate “The original work has been modified.”;

c. to Distribute and Publicly Perform the Work including as incorporated in Collections; and,

d. to Distribute and Publicly Perform Adaptations.

e. For the avoidance of doubt:

   i. **Non-waivable Compulsory License Schemes.** In those jurisdictions in which the right to collect royalties through any statutory or compulsory licensing scheme cannot be waived, the Licensor reserves the exclusive right to collect such royalties for any exercise by You of the rights granted under this License;

   ii. **Waivable Compulsory License Schemes.** In those jurisdictions in which the right to collect royalties through any statutory or compulsory licensing scheme can be waived, the Licensor waives the exclusive right to collect such royalties for any exercise by You of the rights granted under this License; and,

   iii. **Voluntary License Schemes.** The Licensor waives the right to collect royalties, whether individually or, in the event that the Licensor is a member of a collecting society that administers voluntary licensing schemes, via that society, from any exercise by You of the rights granted under this License.

The above rights may be exercised in all media and formats whether now known or hereafter devised. The above rights include the right to make such modifications as are technically necessary to exercise the rights in other media and formats. Subject to Section 8(f), all rights not expressly granted by Licensor are hereby reserved.
4 Restrictions

The license granted in Section 3 above is expressly made subject to and limited by the following restrictions:

a. You may Distribute or Publicly Perform the Work only under the terms of this License. You must include a copy of, or the Uniform Resource Identifier (URI) for, this License with every copy of the Work You Distribute or Publicly Perform. You may not offer or impose any terms on the Work that restrict the terms of this License or the ability of the recipient of the Work to exercise the rights granted to that recipient under the terms of the License. You may not sublicense the Work. You must keep intact all notices that refer to this License and to the disclaimer of warranties with every copy of the Work You Distribute or Publicly Perform. When You Distribute or Publicly Perform the Work, You may not impose any effective technological measures on the Work that restrict the ability of a recipient of the Work from You to exercise the rights granted to that recipient under the terms of the License. This Section 4(a) applies to the Work as incorporated in a Collection, but this does not require the Collection apart from the Work itself to be made subject to the terms of this License. If You create a Collection, upon notice from any Licensor You must, to the extent practicable, remove from the Collection any credit as required by Section 4(c), as requested. If You create an Adaptation, upon notice from any Licensor You must, to the extent practicable, remove from the Adaptation any credit as required by Section 4(c), as requested.

b. You may Distribute or Publicly Perform an Adaptation only under the terms of: (i) this License; (ii) a later version of this License with the same License Elements as this License; (iii) a Creative Commons jurisdiction license (either this or a later license version) that contains the same License Elements as this License (e.g., Attribution-ShareAlike 3.0 US)); (iv) a Creative Commons Compatible License. If you license the Adaptation under one of the licenses mentioned in (biv), you must comply with the terms of that license. If you license the Adaptation under the terms of any of the licenses mentioned in (bi), (bii) or (biii) (the ”Applicable License”), you must comply with the terms of the Applicable License generally and the following provisions: (I) You must include a copy of, or the URI for, the Applicable License with every copy of each Adaptation You Distribute or Publicly Perform; (II) You may not offer or impose any terms on the Adaptation that restrict the terms of the Applicable License or the ability of the recipient of the Adaptation to exercise the rights granted to that recipient under the terms of the Applicable License; (III) You must keep intact all notices that refer to the Applicable License and to the disclaimer of warranties with every copy of the Work as included in the Adaptation You Distribute or Publicly Perform; (IV) when
You Distribute or Publicly Perform the Adaptation, You may not impose any effective technological measures on the Adaptation that restrict the ability of a recipient of the Adaptation from You to exercise the rights granted to that recipient under the terms of the Applicable License. This Section 4(b) applies to the Adaptation as incorporated in a Collection, but this does not require the Collection apart from the Adaptation itself to be made subject to the terms of the Applicable License.

c. If You Distribute, or Publicly Perform the Work or any Adaptations or Collections, You must, unless a request has been made pursuant to Section 4(a), keep intact all copyright notices for the Work and provide, reasonable to the medium or means You are utilizing: (i) the name of the Original Author (or pseudonym, if applicable) if supplied, and/or if the Original Author and/or Licensor designate another party or parties (e.g., a sponsor institute, publishing entity, journal) for attribution ("Attribution Parties") in Licensor’s copyright notice, terms of service or by other reasonable means, the name of such party or parties; (ii) the title of the Work if supplied; (iii) to the extent reasonably practicable, the URI, if any, that Licensor specifies to be associated with the Work, unless such URI does not refer to the copyright notice or licensing information for the Work; and (iv), consistent with Section 3(b), in the case of an Adaptation, a credit identifying the use of the Work in the Adaptation (e.g., "French translation of the Work by Original Author," or "Screenplay based on original Work by Original Author"). The credit required by this Section 4(c) may be implemented in any reasonable manner; provided, however, that in the case of a Adaptation or Collection, at a minimum such credit will appear, if a credit for all contributing authors of the Adaptation or Collection appears, then as part of these credits and in a manner at least as prominent as the credits for the other contributing authors. For the avoidance of doubt, You may only use the credit required by this Section for the purpose of attribution in the manner set out above and, by exercising Your rights under this License, You may not implicitly or explicitly assert or imply any connection with, sponsorship or endorsement by the Original Author, Licensor and/or Attribution Parties, as appropriate, of You or Your use of the Work, without the separate, express prior written permission of the Original Author, Licensor and/or Attribution Parties.

d. Except as otherwise agreed in writing by the Licensor or as may be otherwise permitted by applicable law, if You Reproduce, Distribute or Publicly Perform the Work either by itself or as part of any Adaptations or Collections, You must not distort, mutilate, modify or take other derogatory action in relation to the Work which would be prejudicial to the Original Author’s honor or reputation. Licensor agrees that in those jurisdictions (e.g. Japan), in which any exercise of the right granted in Section 3(b) of this License (the right to make
Adaptations) would be deemed to be a distortion, mutilation, modification or other derogatory action prejudicial to the Original Author’s honor and reputation, the Licensor will waive or not assert, as appropriate, this Section, to the fullest extent permitted by the applicable national law, to enable You to reasonably exercise Your right under Section 3(b) of this License (right to make Adaptations) but not otherwise.

5 Representations, Warranties andDisclaimer

UNLESS OTHERWISE MUTUALLY AGREED TO BY THE PARTIES IN WRITING, LICENSOR OFFERS THE WORK AS-IS AND MAKES NO REPRESENTATIONS OR WARRANTIES OF ANY KIND CONCERNING THE WORK, EXPRESS, IMPLIED, STATUTORY OR OTHERWISE, INCLUDING, WITHOUT LIMITATION, WARRANTIES OF TITLE, MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, NONINFRINGEMENT, OR THE ABSENCE OF LATENT OR OTHER DEFECTS, ACCURACY, OR THE PRESENCE OF ABSENCE OF ERRORS, WHETHER OR NOT DISCOVERABLE. SOME JURISDICTIONS DO NOT ALLOW THE EXCLUSION OF IMPLIED WARRANTIES, SO SUCH EXCLUSION MAY NOT APPLY TO YOU.

6 Limitation on Liability

EXCEPT TO THE EXTENT REQUIRED BY APPLICABLE LAW, IN NO EVENT WILL LICENSOR BE LIABLE TO YOU ON ANY LEGAL THEORY FOR ANY SPECIAL, INCIDENTAL, CONSEQUENTIAL, PUNITIVE OR EXEMPLARY DAMAGES ARISING OUT OF THIS LICENSE OR THE USE OF THE WORK, EVEN IF LICENSOR HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

7 Termination

a. This License and the rights granted hereunder will terminate automatically upon any breach by You of the terms of this License. Individuals or entities who have received Adaptations or Collections from You under this License, however, will not have their licenses terminated provided such individuals or entities remain in full compliance with those licenses. Sections 1, 2, 5, 6, 7, and 8 will survive any termination of this License.

b. Subject to the above terms and conditions, the license granted here is perpetual (for the duration of the applicable copyright in the Work). Notwithstanding the above, Licensor reserves the right to release the Work under different license terms or to stop distributing the Work at any time; provided, however that any
such election will not serve to withdraw this License (or any other license that has been, or is required to be, granted under the terms of this License), and this License will continue in full force and effect unless terminated as stated above.

8 Miscellaneous

a. Each time You Distribute or Publicly Perform the Work or a Collection, the Licensor offers to the recipient a license to the Work on the same terms and conditions as the license granted to You under this License.

b. Each time You Distribute or Publicly Perform an Adaptation, Licensor offers to the recipient a license to the original Work on the same terms and conditions as the license granted to You under this License.

c. If any provision of this License is invalid or unenforceable under applicable law, it shall not affect the validity or enforceability of the remainder of the terms of this License, and without further action by the parties to this agreement, such provision shall be reformed to the minimum extent necessary to make such provision valid and enforceable.

d. No term or provision of this License shall be deemed waived and no breach consented to unless such waiver or consent shall be in writing and signed by the party to be charged with such waiver or consent.

e. This License constitutes the entire agreement between the parties with respect to the Work licensed here. There are no understandings, agreements or representations with respect to the Work not specified here. Licensor shall not be bound by any additional provisions that may appear in any communication from You. This License may not be modified without the mutual written agreement of the Licensor and You.

f. The rights granted under, and the subject matter referenced, in this License were drafted utilizing the terminology of the Berne Convention for the Protection of Literary and Artistic Works (as amended on September 28, 1979), the Rome Convention of 1961, the WIPO Copyright Treaty of 1996, the WIPO Performances and Phonograms Treaty of 1996 and the Universal Copyright Convention (as revised on July 24, 1971). These rights and subject matter take effect in the relevant jurisdiction in which the License terms are sought to be enforced according to the corresponding provisions of the implementation of those treaty provisions in the applicable national law. If the standard suite of rights granted under applicable copyright law includes additional rights not granted under this License, such additional rights are deemed to be included...
in the License; this License is not intended to restrict the license of any rights under applicable law.

**Creative Commons Notice**

Creative Commons is not a party to this License, and makes no warranty whatsoever in connection with the Work. Creative Commons will not be liable to You or any party on any legal theory for any damages whatsoever, including without limitation any general, special, incidental or consequential damages arising in connection to this license. Notwithstanding the foregoing two (2) sentences, if Creative Commons has expressly identified itself as the Licensor hereunder, it shall have all rights and obligations of Licensor.

Except for the limited purpose of indicating to the public that the Work is licensed under the CCPL, Creative Commons does not authorize the use by either party of the trademark "Creative Commons" or any related trademark or logo of Creative Commons without the prior written consent of Creative Commons. Any permitted use will be in compliance with Creative Commons’ then-current trademark usage guidelines, as may be published on its website or otherwise made available upon request from time to time. For the avoidance of doubt, this trademark restriction does not form part of the License.

Creative Commons may be contacted at [http://creativecommons.org/](http://creativecommons.org/).
This bibliography has the goal of presenting a comprehensive list of sources that can be used to check statements and facts presented in our scientific work. As such, one of its prerogatives, apart from completeness and correctness, is discerning among stronger, peer-reviewed publications that can be trusted within an appreciable degree of certainty, and less objective, but still important, opinions and resources: these are mostly found on the web. Besides, we strive to give credit whenever it’s due.

Thus, we have separated between publications and other resources: not only by separating them into two sections of the bibliography, but choosing two different citation styles as well. In the text, it should then be apparent to the reader what rule-of-the-thumb to apply, in order to establish the right amount of trust they should pose in a statement, at least without performing a deeper checking.

**Publications**


Web Resources


Index

action, 22
active restrictions, number of, 9
\textit{AllGlobUnlock}s, 20
\textit{Allocate}$_K$, 32
\textit{Allocate}$_{\text{nullClass}}$, 43
allocation, 31, 32
\textit{Assign}$_{\text{obj}}$, 45
assignment, 45

block
  scope, 45

channel
  \textit{refcount}, 25
  system exception, 50
choice, operator, 6
coverability set, 10
do-while statement, 48
embedding
  among processes, 10
entry point, 51
exception handling, 49

\textit{FieldOps}$_K$, 33
for statement, 48
\textit{FreeRef}, 30

garbage collection, 23
\textit{GcCount}, 21
  definition, 24
example run, 28
\textit{GcCount}_\textit{Add}, 27
\textit{GcCount}_\textit{Dmx}, 26
\textit{GcCount}_\textit{Remove}, 27
\textit{GetFieldLiteral}, 35
\textit{GetFieldObj}, 35
global bound
  loops, 47
global lock, 20
\textit{GlobalUnlock}$_L$, 20

identity-aware process, 10
\textit{if-else} statement, 46
\textit{InitClass}$_K$, 32
\textit{InitClass}$_{\text{nullClass}}$, 43
\textit{#Invoke}, 37

limit process, 11
literature review, 13
  methodology, 14
  queries, 15
  review questions, 13
\textit{LockOver}, 41
loop global bound, 47
\$\textit{loop}_K$, 47

macro, 19
\textit{MakeLock}, 43
\textit{MakeNull}, 44
\textit{MakeObj}, 22
\textit{MakeReentrant}, 41
\textit{MakeRef}, 30
method
implementation, 44
METHODS\(_K\), 36
METHODS\(_{\text{NULLCLASS}}\), 44
name, 5
name boundedness, 9
normal form, 7
null, 43
OBJLIFE, 23
parallel, operator, 6
petri net, 13
\(\pi\)-calculus
   grammar, 5
program representation, 51
#PROGRAMCONFIG, 51
reachability set, 10
reaction relation, 9
receive, operation, 6
recursive lock, 37
reentrant lock, 37
reference counting, 23
RELEASEOBJFIELDS\(_K\), 35
research questions, see also literature review
send, operation, 6
SETFIELD\(_{\text{LITERAL}}\), 35
SETFIELD\(_{\text{OBJ}}\), 36
structural congruence, 6
synchronized, block, 50
SYNCHRONIZED\(_{\text{BLOCK}}\), 50
systematic literature review, see literature review
threads
   creating, 50
THROW, 49
transition system, 9
unit value, 19
while statement, 46