Refining Santa: An Exercise in Efficient Synchronization Extended Abstract

Emil Sekerinski Shucai Yao
Department of Computing and Software
McMaster University
Hamilton, Ontario, Canada
emil@mcmaster.ca yaos4@mcmaster.ca

The Santa Claus Problem is an intricate exercise for concurrent programming. This paper outlines the refinement steps to develop a highly efficient implementation with concurrent objects, starting from a very simple specification. The efficiency of the implementation is compared to three other languages.

1 Introduction

In 1994, Trono proposed the Santa Claus Problem as an exercise in concurrent programming [16]:

> Santa Claus sleeps in his shop up at the North Pole, and can only be wakened by either all nine reindeer being back from their year long vacation on a tropical island, or by some elves who are having some difficulties making the toys. One elf's problem is never serious enough to wake up Santa (otherwise, he may never get any sleep), so, the elves visit Santa in a group of three. When three elves are having their problems solved, any other elves wishing to visit Santa must wait for those elves to return. If Santa wakes up to find three elves waiting at his shop's door, along with the last reindeer having come back from the tropics, Santa has decided that the elves can wait until after Christmas, because it is more important to get his sleigh ready as soon as possible. (It is assumed that the reindeer don't want to leave the tropics, and therefore they stay there until the last possible moment.) The penalty for the last reindeer to arrive is that it must get Santa while the others wait in a warming hut before being harnessed to the sleigh.

Trono's original solution uses ten semaphores. The problem is indeed intricate: as Ben-Ari argues, Trono's solution assumes that a signalled process executes immediately: otherwise, when all reindeer are signalled to proceed to the sleigh, some reindeer may still not be harnessed while others have already finished delivering the toys [2]. A more robust solution would need additional semaphores for barrier synchronization [1]. Ben-Ari argues that the rendezvous construct of Ada is particularly suitable for this problem and compares a solution in Ada with one in Java using monitors. Downey proposes a solution of a simplified problem employing only four semaphores, but makes the assumption that a signalling process does not continue [9]; under some schedulers, e.g. the semaphore implementation of Python, the first elf runs forever.

The Santa Claus Problem follows a line of whimsically named concurrency problems (see [9] for a beautiful collection of those) that all are representative for specific aspects: here, these are *priority* (the reindeer have priority over elves), multi-party synchronization (all reindeer have to be present to engage with Santa and Santa engages either with reindeer or elves), barriers (all reindeer have to be harnessed, then they jointly ride with Santa, then Santa dismisses them), and grouping (Santa consults

Submitted to:
REFINE 2018

elves one by one, but only if a group of three is present). The Santa Claus Problem has been used to illustrate concurrency constructs, e.g. [3, 6, 7, 8, 13] and for comparing concurrency constructs [10]. Peyton Jones gives a solution in Haskell using software transactional memory [14]. Welch and Pedersen present a process-oriented solution using Occam and discuss model-checking a CSP formulation of the problem [17]. The reader is invited to compare these designs with the one proposed here.

This paper develops a solution using concurrent objects by a series of refinement steps. The thrust is to start the development with a specification that is as simple as possible, to add details about Santa, the reindeer, the elves, and their interaction in refinement steps, and to arrive at an implementation that is comparable to other efficient implementations. This work is part of an ongoing research program in developing a highly efficient implementation [12, 18] of concurrent objects together with an accompanying verification and refinement theory [15].

The refinement steps are presented *rigorously*, but are not formally proven to be correct. Our goal is to argue for the potential of refinement with concurrent objects. Once the implementation of our language is finalized, we plan to revisit the theory and complete the proofs.

The next section gives a brief overview of concurrent objects and their refinement. This is followed by the development of a solution to the Santa Claus Problem, the timing results comparing four implementations, and a discussion.

2 Concurrent Objects

Concurrent objects here consists of fields, methods, and actions [4, 5, 11, 15]. Methods must be called to execute but an action can execute on its own whenever its guards is true. Only one method or action can execute at a time in one object, but all objects can execute concurrently. Objects communicate through method calls; no separate mechanism is needed. For synchronization of objects, methods may also have a guard, which can block the caller. Consider class *Santa*:

class *Santa*
  var *s*: {*Sleeping, Working*} = *Sleeping*
  method *wakeup*()
    s = *Sleeping* → *s* := *Working*
  action
    s = *Working* → *s* := *Sleeping*

When object *st* is created by *st* := **new** *Santa*, the method *wakeup* can be called, *st.wakeup*(). The call blocks if $s \neq Sleeping$ and sets field *s* to *Working* otherwise. The single action of the object is executed on its own when its guard is true, $s = Working$, and then sets field *s* to Sleeping. Thus this represents a Santa who needs to be woken up externally, but will go to sleep on his own.

The guard of methods and actions of an object can depend only on fields of that object; the guard cannot refer to fields of other objects or contain calls. This restriction is meant to allow for an efficient implementation: all objects can evaluate their guards concurrently without interference; a guard can change its value by execution with an object, hence guards only need to be reevaluated after a method or action in that object executes.

All methods and actions are executed atomically, up to method calls. For example, if *S* is a statement without calls, the sequence *st.wakeup*(); *S* ; *st.wakeup*() executes the first call *st.wakeup*() atomically, then *S* atomically, then the second call *st.wakeup*() atomically. Using angular brackets to denote atomic regions, this is equivalent to ⟨ *st.wakeup*() ⟩; ⟨ *S* ⟩ ; ⟨ *st.wakeup*() ⟩. Both calls to *wakeup* may block and delay execution until the guard holds, i.e. Santa is sleeping again.

3 Refining Santa

In the refinement steps, subscripts are used to distinguish the different versions when needed. For example, the field *santa* of class $NorthPole_0$ below is referred to as $santa_0$.

Specification: Santa's Cycle

The activity at the North Pole centers around Santa. In the simplest form, Santa either sleeps or works. This is represented by a class with one field for Santa's state and two actions that switch between these two states, whenever Santa feels like doing so.

class $NorthPole_0$
  var *santa*: $\{Sleeping, Working\} = Sleeping$
  action $santa = Sleeping \rightarrow santa := Working$
  action $santa = Working \rightarrow santa := Sleeping$

A single $NorthPole_0$ is created:

np_0 := **new** $NorthPole_0$

Refinement: Splitting Santa's Work

Santa's work consists of either delivering toys or helping the elves: when Santa wakes up, he may either go to state *Delivering* or *Helping*

class $NorthPole_1$
  var *santa*: $\{Sleeping, Delivering, Helping\} = Sleeping$
  action $santa = Sleeping \rightarrow santa := Delivering$
  action $santa = Sleeping \rightarrow santa := Helping$
  action $santa = Delivering \rightarrow santa := Sleeping$
  action $santa = Helping \rightarrow santa := Sleeping$

We assume that a single object np_1 of class $NorthPole_1$ is created. As the coupling invariant between $NorthPole_0$ and $NorthPole_1$we take:

$$R_1 \mathrel{\widehat{=}} santa_0 = Working \equiv santa_1 \in \{Delivering, Helping\}$$

For $NorthPole_0$ to be refined by $NorthPole_1$ we need to show

$$santa_0 = Sleeping \rightarrow santa_0 := Working \sqsubseteq_{R_1} santa_1 = Sleeping \rightarrow santa_1 := Delivering$$
$$santa_0 = Working \rightarrow santa_0 := Sleeping \sqsubseteq_{R_1} santa_1 = Delivering \rightarrow santa_1 := Sleeping$$

and similarly for the analogous actions involving *Helping*.

Refinement: Introducing Reindeer and Elves

For Santa to deliver the toys, he needs reindeer, and to help elves, he obviously needs elves. Abstractly, reindeer are either back from vacation or not, represented by boolean variable b. Elves are either puzzled or not, represented by boolean variable p. The action $b := true$; *deliver*() represents the reindeer returning and then calling *deliver*() to deliver the toys together with Santa. That call will block if Santa is not in state *Delivering*, i.e. the action may get stuck in the middle, and similarly for elves. The action $santa = Sleeping \wedge b \rightarrow santa := Delivering$ represents the reindeer waking up Santa on their return. Note that

the corresponding action of elves has $\neg b$ as part of the guard, meaning that priority is given to reindeer in case both vie for Santa's attention.

class *NorthPole*$_2$
  var *santa*: {*Sleeping, Delivering, Helping*} = *Sleeping*
  var *b*, *p*: *boolean* = *false*, *false*
  method *deliver*()
    santa = *Delivering* → *santa*, *b* := *Sleeping*, *false*
  method *help*()
    santa = *Helping* → *santa*, *p* := *Sleeping*, *false*
  action *b* := *true* ; *deliver*()
  action *p* := *true* ; *help*()
  action *santa* = *Sleeping* ∧ *b* → *santa* := *Delivering*
  action *santa* = *Sleeping* ∧ *p* ∧¬ *b*→ *santa* := *Helping*

We assume that a single object np_2 of class *NorthPole*$_2$ is created. This refinement step is a superposition: variable *santa* is unchanged, variables *b* and *p* are added. As the coupling invariant we take:

$$R_2 \mathrel{\widehat{=}} (santa = Delivering \Rightarrow b) \wedge (santa = Helping \Rightarrow p)$$

That is, Santa can deliver only if the reindeer are back and can help the elves only if the elves are puzzled. Each of the four actions of *NorthPole*$_2$ has to refine the corresponding action of *NorthPole* $_1$. This refinement step involves "splitting atomicity". A general rule for the introducing sequential composition of atomic statements is, for any statements S, T, U and relation R:

$$\langle S \rangle \sqsubseteq_R \langle T \rangle ; \langle U \rangle \equiv skip \sqsubseteq_R \langle T \rangle \wedge \langle S \rangle \sqsubseteq_R \langle U \rangle$$

To show that

santa = *Delivering* → *santa* := *Sleeping* $\sqsubseteq_{R_2}$
  b := *true* ; ⟨ *santa* = *Delivering* → *santa*, *b* := *Sleeping*, *false* ⟩

holds, this rule is applied, resulting in

1. $skip \sqsubseteq_{R_2} b := true$
2. $santa = Delivering \rightarrow santa := Sleeping \sqsubseteq_{R_2} santa = Delivering \rightarrow santa, b := Sleeping, false$

which is easy to see. Refinement of the remaining three actions is shown similarly.

Refinement: Santa's Interactions with Reindeer and Elves

Santa can't simply deliver the toys when the reindeer are back from vacation: they first need to be harnessed before Santa can ride the sleigh. Likewise, the elves first have to enter Santa's shop before Santa can enlighten them. However, Santa rides the sleigh with all reindeer together, but consults the elves one by one. For this, Santa switches between the states of *Welcoming* and *Consulting* three times before going to state *Sleeping*. This formulation ensures the "safety" of Santa.

class *NorthPole*$_3$
  var *santa*: {*Sleeping, Harnessing, Riding, Welcoming, Consulting*} = *Sleeping*
  var *b*: *boolean* = *false*
  var *p*: 0 .. 3 = 0
  method *harness*()

santa = *Harnessing* → *santa* := *Riding*
method *pull*()
santa = *Riding* → *santa*, *b* := *Sleeping*, *false*
method *enter*()
santa = *Welcoming* → *santa* := *Consulting*
method *consult*()
santa = *Consulting* →
p := *p* − 1 ; **if** *p* = 0 **then** *santa* := *Sleeping* **else** *santa* := *Welcoming*
action *b* := *true* ; *harness*() ; *pull*()
action *p* := 3 ; **for** *e* := 1 **to** 3 **do** (*enter*() ; *consult*())
action *santa* = *Sleeping* ∧ *b* → *santa* := *Harnessing*
action *santa* = *Sleeping* ∧ *p* = 3 ∧¬ *b*→ *santa* := *Welcoming*

We assume that a single object np_3 of class $NorthPole_3$ is created. In this step, the values of variable *santa* are extended, variable *b* is kept with the same meaning, variable *p* is generalized from boolean to subrange type, and variable *c* is added. As the coupling invariant we take:

$$R_3 \mathrel{\widehat{=}} (santa_3 = Delivering \equiv santa_4 \in \{Harnessing, Riding\}) \wedge (santa_3 = Helping \equiv santa_4 \in \{Welcoming, Consulting\}) \wedge p_2 \equiv p_3 = 3$$

Refinement: Separating Santa, Sleigh, Shop

So far, only one activity at the North Pole can happen at any time, as all actions are part of the single North Pole object. This step splits the North Pole into three concurrent objects, *Santa*, *Sleigh*, and *Shop*.

class $Santa_4$
var *s*: {*Sleeping, Harnessing, Riding, Welcoming, Consulting*} = *Sleeping*
var *b*: *boolean* = *false*
var *p*: 0 .. 3 = 0
method *back*()
b := *true*
method *harness*()
s = *Harnessing* → *s* := *Riding*
method *pull*()
s = *Riding* → *s*, *b* := *Sleeping*, *false*
method *puzzled*()
p := 3
method *enter*()
s = *Welcoming* → *s* := *Consulting*
method *consult*()
s = *Consulting* →
p := *p* − 1 ; **if** *p* = 0 **then** *s* := *Sleeping* **else** *s* := *Welcoming*
action *s* = *Sleeping* ∧ *b* → *s* := *Harnessing*
action *s* = *Sleeping* ∧ *p* = 3 ∧¬ *b*→ *s* := *Welcoming*

class $Sleigh_4$(*st*: *Santa*)

action *st.back*() ; *st.harness*() ; *st.pull*()

class $Shop_4$(*st*: *Santa*)
action *st.puzzled*() ; **for** *e* := 1 **to** 3 **do** (*st.enter*() ; *st.consult*())

Santa is connected to the sleigh and shop in the initialization:

st := **new** *Santa* ; **new** *Sleigh*(*st*) ; **new** *Shop*(*st*)

This refinement step moves the state of the single $NorthPole_3$ to the single $Santa_4$ object. As the coupling invariant we therefore take:

$R_4 \mathrel{\widehat{=}} np.santa = st.s \wedge np.b = st.b \wedge np.p = st.p$

Refinement: Creating Reindeer and Elves

So far, reindeer and elves are accounted for, but don't have a life of their own. Class *Santa* is unchanged, class *Sleigh* is split into *Sleigh* and *Reindeer*, and class *Shop* is split into *Shop* and *Elf*. The role of *Sleigh* is 3-stage barrier synchronization, i.e. waiting for all reindeer to be back before signalling that to Santa, waiting for all reindeer to be harnessed before signalling that to Santa, and waiting for all reindeer to pull before signalling that to Santa. Barrier synchronization is enforced by an integer counter for each stage with how many arrivals at that stage are still expected. The counters are initialized and reset such that repeated 3-stage synchronization is possible. Each reindeer now cyclically goes through getting back, harnessing, and pulling. Likewise, each elf cyclically goes through being puzzled, entering the shop, and consulting Santa. Note that *puzzled* of *Santa* is only called once for each group of three elves, *enter* is called three times, and *consult* is called three times but doesn't need a guard because it is only called by an elf after calling *enter*.

class $Sleigh_5$(*st*: *Santa*)
var *s*: {*Idle, Back, Harness, Pull*} = *Back*
var *c*: 0 .. 9 = 9
method *back*()
s = *Back* → *c* := *c* − 1 ; **if** *c* = 0 **then** (*s* := *Idle* ; *st.back*() ; *s*, *c* := *Harness*, 9)
method *harness*()
s = *Harness* → *c* := *c* − 1 ; **if** *c* = 0 **then** (*s* := *Idle* ; *st.harness*() ; *s*, *c* := *Pull*, 9)
method *pull*()
s = *Pull* → *c* := *c* − 1 ; **if** *c* = 0 **then** (*s* := *Idle* ; *st.pull*() ; *s*, *c* := *Back*, 9)

class $Reindeer_5$(*sl*: *Sleigh*)
action *sl.back*() ; *sl.harness*() ; *sl.pull*()

class $Shop_5$(*st*: *Santa*)
var *s*: {*Idle, Puzzled, Help*} = *Puzzled*
var *c*: 0 .. 3 = 3
method *puzzled*()
s = *Puzzled* → *c* := *c* − 1 ; **if** *c* = 0 **then** (*s* := *Idle* ; *st.puzzled*() ; *s*, *c* := *Help*, 3)
method *enter*()
s = *Help* → *st.enter*()

Repetitions of Santa	Lime (guards)	C (semaphores)	Go (channels)	Java (monitors)
10,000	0.03 / 0.03 / 0.00	0.87 / 0.26 / 1.18	0.08 / 0.12 / 0.01	6.38 / 2.48 / 5.30
100,000	0.21 / 0.21 / 0.00	8.82 / 2.50 / 12.0	0.77 / 1.18 / 0.06	60.3 / 21.6 / 52.0
1,000,000	2.03 / 2.03 / 0.01	93.0 / 24.8 / 123	7.51 / 11.6 / 0.55	$\approx$ 534 / 159 / 509

Table 1: Execution time in sec on AMD Ryzen Threadripper 1950X 16 core (32 threads) processor with 32 GB memory under Ubuntu 16.04. The compilers used are gcc 5.4.0, Java 9.0.4, Go 1.8.3 linux/amd64. The times are reported as the average real / user / system times of 20 runs. Only a single run was used for Java with 1,000,000 repetitions of Santa.

method *consult*()
st.consult() ; $c := c - 1$; **if** $c = 0$ **then** s, $c :=$ *Puzzled*, 3

class Elf_5(*sh*: *Shop*)
action *sh.puzzled*() ; *sh.enter*() ; *sh.consult*()

Reindeer and elves are connected to the sleigh and shop by:

sl := **new** *Sleigh*(*st*) ; *sh* := **new** *Shop*(*st*) ;
for $i := 1$ **to** 9 **do new** *Reindeer*(*sl*)
for $i := 1$ **to** 10 **do new** *Elf*(*sh*)

We claim that $Sleigh_4$ is refined by $Sleigh_5$ together with $Reindeer_5$: nine $Reindeer_5$ objects are created, each with an action. As the $Sleigh_5$ object does not have actions, the refinement condition is that the sole action of $Sleigh_4$,

st.back() ; *st.harness*() ; *st.pull*()

is refined by the nondeterministic choice of all $Reindeer_5$ actions,

$[]\ r \in Reindeer \bullet$ *r.sl.back*() ; *r.sl.harness*() ; *r.sl.pull*()

where $\#Reindeer = 9$. Likewise $Shop_4$ is refined by $Shop_5$ together with Elf_5.

4 Results

We have implemented an experimental compiler for Lime, a language that closely follows the above theory of concurrent objects. Appendix A contains the Lime implementation of the Santa Claus Problem. The key contributions of the compiler are the management of dynamically growing stacks, the efficient evaluation of method and action guards, a mapping of actions to coroutines, and a distribution of coroutines onto processor cores. The details are in [18].

The Lime implementation is compared to implementations in C using semaphores of the Pthreads library, in Go using channels, and in Java using monitors, see Appendix A. Table 1 shows the running times for Santa with 9 reindeer and 20 elves. Santa's division of work is that for 10,000 rounds until retirement, he rides the sleigh 2,000 times and helps 8,000 times groups of three elves, or for 20 elves, each elf on average 1,200 times. For 100,000 and 1,000,000 rounds until Santa's retirement the ratio is the same. Some observations are in order:

- The Java implementation uses a single monitor for all synchronization. While it would be natural to have Santa, reindeer, and elf processes as well as sleigh, shop, and Santa monitors (synchronizing reindeer, elves, and the sleigh / shop, respectively), this leads to the nested monitor call problem, for example when elves are calling the shop and the shop calls Santa. Ben-Ari's and our implementation use therefore a single monitor with the functionality of sleigh, shop, and Santa monitors. This limits concurrency, e.g. reindeer and elves cannot assemble independently. Java necessitates that each monitor method contains a *notifyAll*() for waking up all threads, most of which will immediately sleep again. The timing results confirm that this is wasteful; in particular the ratio between user and system times make the synchronization effort evident.

- The C implementation uses operating systems threads, which require more cycles when switching than lightweight threads as used by Lime, Go, and Java. Compared to Java with monitors, only the "right" threads are woken up, but the ratio of user to system time tells that switching operating systems threads is expensive.

- The Go implementation uses CSP-like synchronous channels, which are particularly suitable for barrier synchronization with Santa; by comparison, of the semaphore $P()$ and $V()$ operations, only one blocks, meaning that two semaphores are needed for each synchronization point. The goroutines (lightweight threads) of Go are mapped to coroutines, like in Lime, and distributed over cores (like in Lime), leading to good performance. Go does not support priorities when receiving or sending over channels, so to give reindeer priority over elves, a workaround is needed.

- The Lime runtime system is designed for very quickly switching between actions when a guard blocks. Since the bodies of methods and actions in the Santa Claus Problem are short, this pays off. Interestingly, the real time is the user time, suggesting that only one core was active. The Lime runtime system is also designed for distributing a very large number of concurrent objects among cores. As there are relatively few objects here and the bodies of methods are so short that work stealing is not effective, the Lime runtime system is not able to utilize more than one core.

The Haskell implementation of Peyton Jones was not included as its proper functioning depends on the presence of delay statements. Trono's implementation does not run reliably under Pthreads and has more relaxed synchronization constraints than the Lime version, so is not included in the comparison either.

5 Discussion

In ongoing work, we observed on a number of concurrency examples, that Lime compares favourably to all other languages that we compared with [18], which made us wonder if that would be the case for the Santa Claus Problem as well. It took us by surprise that Lime is almost four times faster than Go, about 45 times faster than C, and more than 250 time faster than Java. This line of work provides evidence that the evaluation of guards in methods and actions, compared to synchronizing with semaphores and monitors or sending over channels, is not intrinsically less efficient; the overall efficiency depends more on the techniques used for mapping actions to coroutines and quickly switching between them. This is encouraging for the use of verification and refinement techniques that rely on guards, as these can an applied to highly efficient implementations.

References

[1] Gregory R. Andrews (1991): *Concurrent Programming: Principles and Practice*. Benjamin/Cummings Publishing Company.

[2] Mordechai Ben-Ari (1998): *How to solve the Santa Claus problem*. *Concurrency - Practice and Experience* 10(6), pp. 485–496, doi:10.1002/(SICI)1096-9128(199805)10:6<485::AID-CPE329>3.0.CO;2-2.

[3] Nick Benton (2003): *Jingle bells: Solving the Santa Claus problem in Polyphonic C#*. Technical Report, Microsoft Research. Available at `https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/santa.pdf`.

[4] Marcello M. Bonsangue, Joost N. Kok & Kaisa Sere (1999): *Developing Object-based Distributed Systems*. In P. Ciancarini, A. Fantechi & R. Gorrieri, editors: *3rd IFIP International Conference on Formal Methods for Open Object-based Distributed Systems (FMOODS'99)*, Kluwer, pp. 19–34.

[5] Martin Büchi & Emil Sekerinski (2000): *A Foundation for Refining Concurrent Objects*. *Fundamenta Informaticae* 44(1,2), pp. 25–61. Available at `http://content.iospress.com/articles/fundamenta-informaticae/fi44-1-2-02`.

[6] Peter A. Buhr (2016): *High-Level Concurrency Constructs*. In: *Understanding Control Flow: Concurrent Programming Using μC++*, Springer International Publishing, Cham, pp. 425–522, doi:10.1007/978-3-319-25703-7_9.

[7] Nick Cameron, Ferruccio Damiani, Sophia Drossopoulou, Elena Giachino & Paola Giannini (2006): *Solving the Santa Claus problem using state classes*. Technical Report, Dip. di inf., Univ. di Torino. Available at `http://www.di.unito.it/~damiani/papers/scp.pdf`.

[8] Steingrim Dovland (2006): *Liberating Coroutines: Combining Sequential and Parallel Execution*. Master's thesis, University of Oslo, Department of Informatics. Available at `http://urn.nb.no/URN:NBN:no-11637`.

[9] Allen B. Downey (2016): *Little Book of Semaphores*. Green Tea Press. Available at `http://greenteapress.com/semaphores`.

[10] Jason Hurt & Jan B. Pedersen (2008): *Solving the Santa Claus Problem: a Comparison of Various Concurrent Programming Techniques*. In Peter H. Welch, Susan Stepney, Fiona A.C. Polack, Frederick R.M. Barnes, Alistair A. McEwan & Adam T. Sampson Gardiner S. Stiles, Jan F. Broenink, editors: *Communicating Process Architectures 2008*, IOS Press, pp. 381–396, doi:10.3233/978-1-58603-907-3-381.

[11] Jayadev Misra (2002): *A Simple, Object-Based View of Multiprogramming*. *Formal Methods in System Design* 20(1), pp. 23–45, doi:10.1023/A:1012904412467.

[12] Joshua Moore-Oliva, Emil Sekerinski & Shucai Yao (2014): *A Comparison of Scalable Multi-Threaded Stack Mechanisms*. Technical Report CAS-14-07-ES, McMaster University, Department of Computing and Software. Available at `http://www.cas.mcmaster.ca/cas/0reports/CAS-14-07-ES.pdf`.

[13] Piotr Nienaltowski (2007): *Practical framework for contract-based concurrent object-oriented programming*. Ph.D. thesis, ETH Zürich, doi:10.3929/ethz-a-005363875.

[14] Simon Peyton Jones (2007): *Beautiful concurrency*. In A. Oram & G. Wilson, editors: *Beautiful Code: Leading Programmers Explain How They Think*, O'Reilly, pp. 385–406. Available at `https://www.schoolofhaskell.com/school/advanced-haskell/beautiful-concurrency`.

[15] Emil Sekerinski (2005): *Verification and Refinement with Fine-Grained Action-Based Concurrent Objects*. *Theoretical Computer Science* 331(2–3), pp. 429–455, doi:10.1016/j.tcs.2004.09.024.

[16] John A. Trono (1994): *A new exercise in concurrency*. *ACM SIGCSE Bulletin* 26(3), pp. 8–10, doi:10.1145/187387.187391.

[17] Peter H. Welch & Jan B. Pedersen (2010): *Santa Claus: Formal Analysis of a Process-oriented Solution*. *ACM Trans. Program. Lang. Syst.* 32(4), pp. 14:1–14:37, doi:10.1145/1734206.1734211.

[18] Shucai Yao (2018, Draft): *An Efficient Implementation of Guard-based Synchronization for an Object-Oriented Programming Language*. Ph.D. thesis, McMaster University.

Appendix A

These implementations are used in the comparison of timing results.

Listing 1: Implementation with Lime

```
class Santa
    var s: {Sleeping, Harnessing, Riding, Welcoming, Consulting}
    var b: boolean
    var p: int
    init ()
        s, b, p := Sleeping, false, 0
    method back()
        b := true
    method harness()
        when s = Harnessing do
            s := Riding
    method pull()
        when s = Riding do
            s, b := Sleeping, false
    method puzzled()
        p := 3
    method enter()
        when s = Welcoming do
            s := Consulting
    method consult()
        when s = Consulting do
            p := p − 1
            if p = 0 then
                s := Sleeping
            else
                s := Welcoming
    action action1
        when s = Sleeping and b do
            s := Harnessing
    action action2
        when s = Sleeping and p = 3 and not b do
            s := Welcoming

class Sleigh
    var s: {Idle, Back, Harness, Pull}
    var c: int
    var st: Santa
    init (santa: Santa)
        s, c, st := Back, 9, santa
    method back()
        when s = Back do
            c := c − 1
            if c = 0 then
                s := Idle
                st.back()
                s, c := Harness, 9
    method harness()
```

```
            when s = Harness do
                c := c − 1
                if c = 0 then
                    s := Idle
                    st . harness()
                    s, c := Pull, 9
        method pull()
            when s = Pull do
                c := c − 1
                if c = 0 then
                    s := Idle
                    st . pull ()
                    s, c := Back, 9

class Reindeer
    var sl : Sleigh
    init (sleigh : Sleigh )
        sl := sleigh
    action action1
        sl . back()
        sl . harness()
        sl . pull ()

class Shop
    var s: {Idle, Puzzled, Help}
    var c: int
    init (santa: Santa)
        s, c, st := Puzzled, 3, santa
    method puzzled()
        when s = Puzzled do
            c := c − 1
            if c = 0 then
                s := Idle
                st . puzzled ()
                s, c := Help, 3
    method enter()
        when s = Help do
            st . enter ()
    method consult()
        st . consult ()
        c := c − 1
        if c = 0 then
            s, c := Puzzled, 3

class Elf
    var sh: Shop
    init (shop: Shop)
        sh := shop
    action action1
        sh. puzzled ()
        sh. enter ()
        sh. consult ()

class Start
    var st : Santa
    var sl : Sleigh
```

```
    var sh: Shop
    init ()
        st := new Santa()
        sl := new Sleigh(st)
        sh := new Shop(st)
        for i := 1 to 9 do new Reindeer(sl)
        for i := 1 to 20 do new Elf(sh)
```

Listing 2: Implementation with C

```
#include <stdbool.h>
#include <pthread.h>
#include <semaphore.h>
#define P(sem) (sem_wait(&(sem))) /* uses P and V for the wait and ... */
#define V(sem) (sem_post(&(sem))) /* ... signal semaphore operations */

sem_t wakeup, wakeupReindeer, wakeupElves;
sem_t harness, harnessDone;
sem_t pull, pullDone;
sem_t enter, enterDone;
sem_t consult, consultDone;
sem_t reindeerBack, reindeerBackDone;
sem_t reindeerHarness, reindeerHarnessDone;
sem_t reindeerPull, reindeerPullDone;
sem_t elfPuzzled, elfPuzzledDone;
sem_t elfEnter, elfEnterDone;
sem_t elfConsult, elfConsultDone;

bool b;

void *Santa(void *arg) {
    for (int t = 0; t < 10000; t++) { // Sleeping
        P(wakeup); // woken up by Sleigh or Shop
        if (b) { // Delivering
            b = false; V(wakeupReindeer);
            P(harness); V(harnessDone);
            P(pull); V(pullDone);
        } else { // Helping
            V(wakeupElves);
            for (int i = 0; i < 3; i++) {
                P(enter); V(enterDone);
                P(consult); V(consultDone);
            }
        }
    }
}
void *Sleigh(void *arg) {
    for (;;) {
        for (int i = 0; i < 9; i++) V(reindeerBack);
        for (int i = 0; i < 9; i++) P(reindeerBackDone);
        b = true; V(wakeup); P(wakeupReindeer);
        for (int i = 0; i < 9; i++) V(reindeerHarness);
        for (int i = 0; i < 9; i++) P(reindeerHarnessDone);
        V(harness); P(harnessDone);
        for (int i = 0; i < 9; i++) V(reindeerPull);
        for (int i = 0; i < 9; i++) P(reindeerPullDone);
        V(pull); P(pullDone);
```

```
    }
}
void *Reindeer(void *arg) {
    for (int t = 0; t < 2000; t++) {
        P(reindeerBack); V(reindeerBackDone);
        P(reindeerHarness); V(reindeerHarnessDone);
        P(reindeerPull ); V(reindeerPullDone);
    }
}
void *Shop(void *arg) {
    for (;;) {
        for (int i = 0; i < 3; i++) V(elfPuzzled );
        for (int i = 0; i < 3; i++) P(elfPuzzledDone);
        V(wakeup); P(wakeupElves);
        for (int i = 0; i < 3; i++) {
            V(elfEnter ); P(elfEnterDone);
            V(enter ); P(enterDone);
            V(elfConsult ); P(elfConsultDone);
            V(consult ); P(consultDone);
        }
    }
}
void *Elf( void *arg) {
    for (;;) {
        P(elfPuzzled ); V(elfPuzzledDone);
        P(elfEnter ); V(elfEnterDone);
        P(elfConsult ); V(elfConsultDone);
    }
}
void main() {
    sem_init (&wakeup, 0, 0); sem_init (&wakeupReindeer, 0, 0); sem_init (&wakeupElves, 0, 0);
    sem_init (&harness, 0, 0); sem_init (&harnessDone, 0, 0);
    sem_init (&pull, 0, 0); sem_init (&pullDone, 0, 0);
    sem_init (&enter, 0, 0); sem_init (&enterDone, 0, 0);
    sem_init (&consult, 0, 0); sem_init (&consultDone, 0, 0);
    sem_init (&reindeerBack, 0, 0); sem_init (&reindeerBackDone, 0, 0);
    sem_init (&reindeerHarness, 0, 0); sem_init (&reindeerHarnessDone, 0, 0);
    sem_init (&reindeerPull , 0, 0); sem_init (&reindeerPullDone, 0, 0);
    sem_init (&elfPuzzled , 0, 0); sem_init (&elfPuzzledDone, 0, 0);
    sem_init (&elfEnter , 0, 0); sem_init (&elfEnterDone, 0, 0);
    sem_init (&elfConsult , 0, 0); sem_init (&elfConsultDone, 0, 0);
    pthread_t tid ;
    for (int i = 0; i < 9; i++) pthread_create (&tid, NULL, Reindeer, NULL);
    for (int i = 0; i < 20; i++) pthread_create (&tid, NULL, Elf, NULL);
    pthread_create (&tid, NULL, Sleigh, NULL); pthread_create (&tid, NULL, Shop, NULL);
    pthread_create (&tid, NULL, Santa, NULL); pthread_join (tid , NULL);
}
```

Listing 3: Implementation with Go

```
package main

var reindeerBack, reindeerHarness, reindeerPull chan bool
var back, harness, pull chan bool
var elfPuzzled , elfEnter , elfConsult chan bool
var puzzled, enter , consult chan bool
var done chan bool
```

```
func Santa() {
    b, p := false, false // reindeer back, elves puzzled
    for t := 0; t < 10000; t++ { // invariant: !b
        if !p { // neither reindeer back nor elves puzzled
            select { // wait for either one
            case <- back: b = true
            case <- puzzled: p = true
            }
        }
        if p { // elves puzzled
            select { // check if reindeer back as well
            case <- back: b = true
            default:
            }
        }
        // either b or p is true, pick one
        if b { // prefer reindeer
            <- harness; <- pull; b = false
        } else { // otherwise elves
            for i := 0; i < 3; i++ {
                <- enter; <- consult
            }
            p = false
        }
    }
    done <- true
}
func Sleigh() {
    for {
        for i := 0; i < 9; i++ {<- reindeerBack}
        back <- true
        for i := 0; i < 9; i++ {<- reindeerHarness}
        harness <- true
        for i := 0; i < 9; i++ {<- reindeerPull}
        pull <- true
    }
}
func Shop() {
    for {
        for i := 0; i < 3; i++ {<- elfPuzzled}
        puzzled <- true
        for i := 0; i < 3; i++ {
            <- elfEnter; enter <- true; <- elfConsult; consult <- true
        }
    }
}
func Reindeer() {
    for r := 0; r < 2000; r++ {
        reindeerBack <- true; reindeerHarness <- true; reindeerPull <- true
    }
}
func Elf() {
    for {
        elfPuzzled <- true; elfEnter <- true; elfConsult <- true
    }
```

```
}
func main() {
    reindeerBack, reindeerHarness, reindeerPull = make(chan bool), make(chan bool), make(chan bool)
    back, harness, pull = make(chan bool), make(chan bool), make(chan bool)
    elfPuzzled , elfEnter , elfConsult = make(chan bool), make(chan bool), make(chan bool)
    puzzled, enter, consult = make(chan bool), make(chan bool), make(chan bool)
    done = make(chan bool)
    go Santa() ; go Sleigh () ; go Shop()
    for i := 0; i < 9; i++ {go Reindeer() }
    for i := 0; i < 20; i++ {go Elf() }
    <− done
}
```

Listing 4: Implementation with Java

```
enum R {Relaxing, Back, Harnessing, Harnessed, Pulling , Done}
enum E {Working, Puzzled, Entering, Entered, Consulting, Enlightened}
enum Task {deliver, help}
class SantasShop {
    int rc = 9, ec = 3; // reindeer count, elf count
    R rs = R.Relaxing; // state of reindeer
    E es = E.Working; // state of elves

    synchronized void back() /* called by reindeer */ {
        while (rs != R.Relaxing) try {wait() ;} catch (Exception x) {}
        rc −= 1; if (rc == 0) {rs = R.Back; rc = 9;} notifyAll () ;
    }
    synchronized void harness() /* called by reindeer */ {
        while (rs != R.Harnessing) try {wait() ;} catch (Exception x) {}
        rc −= 1; if (rc == 0) {rs = R.Harnessed; rc = 9;} notifyAll () ;
    }
    synchronized void pull () /* called by reindeer */ {
        while (rs != R.Pulling ) try {wait() ;} catch (Exception x) {}
        rc −= 1; if (rc == 0) {rs = R.Done; rc = 9;} notifyAll () ;
    }

    synchronized void puzzled() /* called by elves */ {
        while (es != E.Working) try {wait() ;} catch (Exception x) {}
        ec −= 1; if (ec == 0) {es = E.Puzzled; ec = 3;} notifyAll () ;
    }
    synchronized void enter() /* called by elves */ {
        while (es != E.Entering) try {wait() ;} catch (Exception x) {}
        es = E.Entered; notifyAll () ;
    }
    synchronized void consult () /* called by elves */ {
        while (es != E.Consulting) try {wait() ;} catch (Exception x) {}
        es = E.Enlightened; notifyAll () ;
    }

    synchronized Task wakeup() /* called by Santa */ {
        while (rs != R.Back && es != E.Puzzled) try {wait() ;} catch (Exception x) {}
        if (rs == R.Back) {rs = R. Harnessing; notifyAll () ; return Task. deliver ;}
        else {es = E.Entering; notifyAll () ; return Task. help ;}
    }
    synchronized void hitch () /* called by Santa */ {
        while (rs != R.Harnessed) try {wait() ;} catch (Exception x) {}
        rs = R.Pulling ; notifyAll () ;
```

```
    }
    synchronized void ride() /* called by Santa */ {
        while (rs != R.Done) try {wait();} catch (Exception x) {}
        rs = R.Relaxing; notifyAll();
    }
    synchronized void welcome() /* called by Santa */ {
        while (es != E.Entered) try {wait();} catch (Exception x) {}
        es = E.Consulting; notifyAll();
    }
    synchronized void explain() /* called by Santa */ {
        while (es != E.Enlightened) try {wait();} catch (Exception x) {}
        ec -= 1; if (ec == 0) {es = E.Working; ec = 3;} else es = E.Entering;
        notifyAll();
    }

    public static void main(String[] args) {
        SantasShop shop = new SantasShop();
        new Santa(shop).start();
        for (int i = 0; i < 9; i++) new Reindeer(shop).start();
        for (int i = 0; i < 20; i++) {Thread e = new Elf(shop, i); e.setDaemon(true); e.start();}
    }
}
class Santa extends Thread {
    SantasShop shop;
    Santa(SantasShop ss) {shop = ss;}
    public void run() {
        for (int t = 0; t < 10000; t++) {
            Task task = shop.wakeup();
            if (task == Task.deliver) {
                shop.hitch(); shop.ride();
            } else {
                for (int i = 0; i < 3; i++) {shop.welcome(); shop.explain();}
            }
        }
    }
}
class Reindeer extends Thread {
    SantasShop shop;
    Reindeer(SantasShop ss) {shop = ss;}
    public void run() {
        for (int t = 0; t < 2000; t++) {shop.back(); shop.harness(); shop.pull();}
    }
}
class Elf extends Thread {
    SantasShop shop; int num;
    Elf(SantasShop ss, int n) {shop = ss; num = n;}
    public void run() {
        for (;;) {shop.puzzled(); shop.enter(); shop.consult();}
    }
}
```