Refining Santa: An Exercise in Efficient Synchronization

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 simple specification. The efficiency of the implementation is compared to those in 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 *batch processing* (Santa consults 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].

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 next section introduces concurrent objects with guard-based synchronization and discusses the assumptions about atomicity. A general refinement rule for concurrent objects is given and informally justified. This is followed by the presentation of the Santa Claus problem, the development of a solution in five refinement steps, the timing results comparing four implementations, and a discussion. The proofs of the refinement steps are sketched but not carried out in full detail: our goal is to argue that the chosen model of concurrent objects allows both highly efficient implementations and intuitive correctness reasoning.

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 *guard* 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 field *s* of *st* is not equal to *Sleeping* and sets *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 guards 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. In general, if the execution of a method

or action is suspended, another method or action may start to execute or continue execution. There can be arbitrarily many suspended method executions in an object. Once an action is chosen, that action will be executed until termination before another action can be initiated, hence at most one action execution can be suspended. There can only be as many concurrent executions as there are objects.

We assume that all fields of an object are *private* to the object, i.e. are accessed only by the methods and actions of the object. The sole purpose of classes is to create objects. In general, class *D* refines class *C* if a *D* object can be used instead of a *C* object. The following rule formalizes refinement with a *coupling relation* that relates the fields of *C* and *D*:

Rule 1 (Class Refinement) *Consider classes C, D with list f, g of fields, initialized to f_0, g_0, with methods $m_k()$ with bodies S_k, T_k, and with actions A_i, B_j. Class D may have new methods $n_l()$ with bodies U_l:*

$$
\begin{array}{ll}
\textbf{class } C & \textbf{class } D \\
\quad \textbf{var } f = f_0 & \quad \textbf{var } g = g_0 \\
\quad \textbf{method } m_k() & \quad \textbf{method } m_k() \\
\qquad S_k & \qquad T_k \\
\quad \textbf{action } A_i & \quad \textbf{method } n_l() \\
 & \qquad U_l \\
 & \quad \textbf{action } B_j
\end{array}
$$

Class C is refined by class D if for some relation R over f and g:

(**I**) $R f_0 g_0$

(**M**) $S_k \sqsubseteq_R T_k$ *for all k*

(**N**) $skip \sqsubseteq_R U_l$ *or* $A_i \sqsubseteq_R U_l$ *for all l and some i*

(**A**) $A_i \sqsubseteq_R B_j$ *for all j and some i*

Condition (**I**) requires that the field initializations have to establish the coupling relation *R*. Condition (**M**) requires that each method of *D* refines the corresponding method of *C* through *R*. Condition (**M**) requires that new methods of *D* either *stutter*, i.e. refine *skip* through *R*, or refine some action of *C* through *R*. Condition (**A**) requires that all actions of *D* refine some action of *C*. Note that not all actions of *C* have to be refined, i.e. *D* can restrict the behaviour of *C*.

For the refinement of statements through a relation, we give only a single rule and appeal to intuition otherwise:

Rule 2 (Guarded Assignment Refinement) *Let b, d be expressions over variables x, let c, e be expressions over variables y, and let R be a relation between x and y:*

$$ b \to x := d \sqsubseteq_R c \to y := e \quad \text{if} \quad R x y \wedge c \Rightarrow b \quad \text{and} \quad R x y \wedge c \Rightarrow R d e $$

In refinement steps, new classes may be introduced and objects of those classes may be created. Above rules are applied to ensure that the behaviour of existing objects is preserved.

3   Refining Santa

In the development below, subscripts are used to distinguish names across refinement steps.

Specification: Santa's Cycle

The activity at the North Pole centers around Santa. In the simplest form, Santa either sleeps or works. This is expressed 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** $Santa_0$
> > **var** s: $\{Sleeping, Working\}$ = $Sleeping$
> > **action** s = $Sleeping \rightarrow s$:= $Working$
> > **action** s = $Working \rightarrow s$:= $Sleeping$

A single object st of class $Santa_0$ is created:

> st := **new** $Santa_0$

Refinement 1: 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** $Santa_1$
> > **var** s: $\{Sleeping, Delivering, Helping\}$ = $Sleeping$
> > **action** s = $Sleeping \rightarrow s$:= $Delivering$
> > **action** s = $Sleeping \rightarrow s$:= $Helping$
> > **action** s = $Delivering \rightarrow s$:= $Sleeping$
> > **action** s = $Helping \rightarrow s$:= $Sleeping$

A single object st of class $Santa_1$ is created:

> st := **new** $Santa_1$

For applying the rule for Class Refinement, as the coupling relation between $Santa_0$ and $Santa_1$ we take:

$$R_1 \, s_0 \, s_1 \mathrel{\widehat{=}} s_0 = Working \equiv s_1 \in \{Delivering, Helping\}$$

Since there are no methods in $Santa_1$, refinement follows from the conditions for the initialization and the four actions of $Santa_1$:

(I) $R_1 \, Sleeping \, Sleeping$

(A_1) $s_0 = Sleeping \rightarrow s_0$:= $Working \sqsubseteq_{R_1} s_1 = Sleeping \rightarrow s_1$:= $Delivering$

(A_2) $s_0 = Sleeping \rightarrow s_0$:= $Working \sqsubseteq_{R_1} s_1 = Sleeping \rightarrow s_1$:= $Helping$

(A_3) $s_0 = Working \rightarrow s_0$:= $Sleeping \sqsubseteq_{R_1} s_1 = Delivering \rightarrow s_1$:= $Sleeping$

(A_4) $s_0 = Working \rightarrow s_0$:= $Sleeping \sqsubseteq_{R_1} s_1 = Helping \rightarrow s_1$:= $Sleeping$

Conditions **(A_1)** to **(A_4)** hold by the rule for Guarded Assignment Refinement.

Refinement 2: Introducing Santa's Sleigh

Santa's shop coordinates the elves and Santa's sleigh coordinates the reindeer. We first introduce and prioritize the sleigh, postponing the introduction of the reindeer, the shop, and the elves. Here, the sleigh is an active object: the sleigh signals to Santa that all reindeer are back, then Santa harnesses all reindeer, then the reindeer pull the sleigh until Santa releases all reindeer and sleeps again. The synchronization is expressed by the sleigh calling newly introduced methods of *Santa*. Class $Santa_2$ splits the *Delivering* state of $Santa_1$ into *Harnessing* and *Riding*; field b is true if the reindeer are back from vacationing:

> **class** $Santa_2$
>   **var** s: {*Sleeping, Harnessing, Riding, Helping*} = *Sleeping*
>   **var** b: *boolean* = *false*
>   **method** *back*()
> b := *true*
>   **method** *harness*()
> $s = Harnessing \rightarrow s$:= *Riding*
>   **method** *pull*()
> $s = Riding \rightarrow s, b$:= *Sleeping*, *false*
>   **action** $s = Sleeping \wedge b \rightarrow s$:= *Harnessing*
>   **action** $s = Sleeping \wedge \neg\, b \rightarrow s$:= *Helping*
>   **action** $s = Helping \rightarrow s$:= *Sleeping*
>
> **class** $Sleigh_2(st: Santa_2)$
>   **var** s: {*Back, Harnessing, Pulling*} = *Back*
>   **action** $s = Back \rightarrow s$:= *Harnessing* ; $st.back()$
>   **action** $s = Harnessing \rightarrow s$:= *Pulling* ; $st.harness()$
>   **action** $s = Pulling \rightarrow s$:= *Back* ; $st.pull()$

Object st of class $Santa_2$ and object sl of class $Sleigh_2$ are created; these objects can execute concurrently:

> st := **new** $Santa_2$; sl := **new** $Sleigh_2(st)$

The first action of $Sleigh_2$ calls $st.back()$, which executes immediately but under mutual exclusion with any other action of st. The calls $st.harness()$ and $st.harness()$ will block until the corresponding guard is true. For applying the rule for Class Refinement to show that $Santa_2$ refines $Santa_1$, we take as the coupling relation:

$$R_2\, s_1\, (s_2, b_2) \mathrel{\widehat{=}} (s_1 = Delivering \equiv s_2 \in \{Harnessing, Riding\}) \wedge (s_1 = Delivering \Rightarrow b_2)$$

Refinement follows as *back* and *harness* of $Santa_2$ stutter under R_2 and *pull* refines the action $s = Delivering \rightarrow s$:= *Sleeping* of $Santa_1$ under R_2. Field b of $Santa_2$ reduces the nondeterminism that is present among the actions of $Santa_1$. Formally, the conditions are:

  (I) $R_2\, Sleeping\, (Sleeping, false)$

(N$_1$) $skip \sqsubseteq_{R_2} b$:= *true*

(N$_2$) $skip \sqsubseteq_{R_2} s_2 = Harnessing \rightarrow s_2$:= *Riding*

(N$_3$) $s_1 = Delivering \rightarrow s_1$:= *Sleeping* $\sqsubseteq_{R_2} s_2 = Riding \rightarrow s_2, b_2$:= *Sleeping*, *false*

(A$_1$) $s_1 = Sleeping \rightarrow s_1$:= *Delivering* $\sqsubseteq_{R_2} s_2 = Sleeping \wedge b_2 \rightarrow s_2$:= *Harnessing*

(A$_2$) $s_1 = Sleeping \rightarrow s_1$:= *Helping* $\sqsubseteq_{R_2} s_2 = Sleeping \wedge \neg b_2 \rightarrow s_2$:= *Helping*

(**A₃**) $s_1 = Helping \rightarrow s_1 := Sleeping \sqsubseteq_{R_2} s_2 = Helping \rightarrow s_2 := Sleeping$

These follow from the rule for Guarded Assignment Refinement.

Refinement 3: Introducing Reindeer

This step leaves Santa unchanged, refines Santa's sleigh into a passive sleigh, and introduces active reindeer. The sleigh coordinates the reindeer by keeping a count, *c*, for the number of reindeer that need to come back, that need to be harnessed, and that need to be pulling. The reindeer cyclically call the *back*, *harness*, *pull* methods of the sleigh. Since reindeer are not further refined, this is simply expressed by a single action composing these calls in sequence rather than by three actions.

> **class** *Sleigh₃*(*st*: *Santa₂*)
>   **var** *s*: {*Back, Harnessing, Pulling*} = *Back*
>   **var** *c*: 0 .. 9 = 9
>   **method** *back*()
> $s = Back \rightarrow c := c - 1$; **if** $c = 0$ **then** ($s, c := Harnessing, 9$; *st.back*())
>   **method** *harness*()
> $s = Harnessing \rightarrow c := c - 1$; **if** $c = 0$ **then** ($s, c := Pulling, 9$; *st.harness*())
>   **method** *pull*()
> $s = Pulling \rightarrow c := c - 1$; **if** $c = 0$ **then** ($s, c := Back, 9$; *st.pull*())

> **class** *Reindeer₃*(*sl*: *Sleigh₃*)
>   **action** *sl.back*() ; *sl.harness*() ; *sl.pull*()

One sleigh and nine reindeer are created:

> *sl* := **new** *Sleigh₃* ; **for** *i* := 1 **to** 9 **do new** *Reindeer₃*(*sl*)

As a note, the refinement is also correct is more or fewer than nine reindeer are created: if there are more than nine reindeer, the first nine arriving will be harnessed; if there are fewer than nine reindeer, Santa can only occupy himself with the Elves and no presents will be delivered! The coupling relation between *Sleigh₂* and *Sleigh₃* includes the identity relation on *s* and restricts *c* to be between 1 and 9:

$$R_3 s_2 (s_3, c_3) \mathrel{\widehat{=}} s_2 = s_3 \wedge 1 \leq c_3 \leq 9$$

For brevity, we refer to the body of method *m* of class *C* as *C.m*. Refinement of *Sleigh₂* by *Sleigh₃* then follows from:

  (**I**) $R_2 Back (Back, 9)$

(**N₁**) $skip \sqsubseteq_{R_3} Sleigh_3.back$ or $s_2 = Back \rightarrow s_2 := Harnessing; st.back() \sqsubseteq_{R_3} Sleigh_3.back$

(**N₂**) $skip \sqsubseteq_{R_3} Sleigh_3.harness$ or $s_2 = Harnessing \rightarrow s_2 := Pulling; st.harness() \sqsubseteq_{R_3} Sleigh_3.harness$

(**N₃**) $skip \sqsubseteq_{R_3} Sleigh_3.pull$ or $s_2 = Pulling \rightarrow s_2 := Back; st.pull() \sqsubseteq_{R_3} Sleigh_3.pull$

To show (**N₁**), we distinguish the cases $c > 1$ and $c = 1$: if $c > 1$ initially, then *Sleigh₃.back* simplifies to $s = Back \rightarrow c := c - 1$, which refines *skip* under R_3; if $c = 1$ initially, then *Sleigh₃.back* simplifies to $s = Back \rightarrow s, c := Harnessing, 9; st.back()$, which refines $s = Back \rightarrow s := Harnessing; st.back()$ under R_3. The conditions (**N₂**) and (**N₃**) are shown similarly.

Refinement 4: Introducing Santa's Shop

Class *Santa*$_4$ splits the *Helping* state of *Santa*$_2$ into *Welcoming* and *Consulting*; field *p* is the number of puzzled elves. Santa will be woken up only by a group of three elves but then has to consult each individually. The shop is here an active object that represents the collective behaviour of elves: class *Shop*$_4$ maintains a count of the number of elves of the current group that still have to consult with Santa:

> **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* := *Welcoming* **else** *s* := *Sleeping*
>   **action** *s* = *Sleeping* ∧ *b* → *s* := *Harnessing*
>   **action** *s* = *Sleeping* ∧*p* = 3 ∧ ¬*b* → *s* := *Welcoming*

> **class** *Shop*$_4$(*st*: *Santa*$_4$)
>   **var** *s*: {*Puzzled, Entering, Consulting*} = *Puzzled*
>   **var** *c*: 0 .. 3 = 0
>   **action** *s* = *Puzzled* → *s*, *c* := *Entering*, 3 ; *st.puzzled*()
>   **action** *s* = *Entering* → *s* := *Consulting* ; *st.enter*()
>   **action** *s* = *Consulting* → *c* := *c* − 1 ; **if** *c* > 0 **then** *s* := *Entering* **else** *s*:= *Puzzled* ; *st.consult*()

One Santa and one shop are created:

> *st* := **new** *Santa*$_4$; *sh* := **new** *Shop*$_4$(*st*)

As the coupling relation between *Santa*$_2$ and *Santa*$_4$ we take:

$$R_4 (s_2, b_2) (s_4, b_4, p_4) \mathrel{\widehat{=}} (s_2 = s_4 \lor (s_2 = Helping \land s_4 \in \{Welcoming, Consulting\})) \land$$
$$b_2 = b_4$$

Class *Sleigh*$_4$ refines *Sleigh*$_2$ as the methods *back*, *harness*, *pull* refine themselves under R_4, new methods *puzzled*, *enter* stutter under R_4, new method *consult* stutters if *p* > 0 initially and refines $s_2 = Helping \to s_2 := Sleeping$ if *p* = 0 initially, and the two actions of *Sleigh*$_4$ refine actions of *Sleigh*$_2$. Formally, the conditions are:

(**I**) $R_4 (Sleeping, false) (Sleeping, false, 0)$

(**M$_1$**) *Sleigh*$_2$.*back* $\sqsubseteq_{R_4}$ *Sleigh*$_4$.*back*

(**M$_2$**) *Sleigh*$_2$.*harness* $\sqsubseteq_{R_4}$ *Sleigh*$_4$.*harness*

(**M₃**) *Sleigh₂.pull* $\sqsubseteq_{R_4}$ *Sleigh₄.pull*

(**N₁**) *skip* $\sqsubseteq_{R_4}$ *Sleigh₄.puzzled*

(**N₂**) *skip* $\sqsubseteq_{R_4}$ *Sleigh₄.enter*

(**N₃**) *skip* $\sqsubseteq_{R_4}$ *Sleigh₄.consult*   or $s_2 = Helping \rightarrow s_2 := Sleeping$ $\sqsubseteq_{R_4}$ *Sleigh₄.consult*

(**A₁**) $s_2 = Sleeping \wedge b_2 \rightarrow s_2 := Harnessing$ $\sqsubseteq_{R_4}$ $s_4 = Sleeping \wedge b_4 \rightarrow s_4 := Harnessing$

(**A₂**) $s_2 = Sleeping \wedge \neg b_2 \rightarrow s_2 := Helping$ $\sqsubseteq_{R_4}$ $s_4 = Sleeping \wedge p_4 = 3 \wedge \neg b_4 \rightarrow s_4 := Welcoming$

Refinement 5: Introducing Elves

This step leaves Santa, the sleigh, and the reindeer unchanged, refines the shop into a passive shop, and introduced active elves.

> **class** *Shop₅*(*st*: *Santa*)
>   **var** *s*: {*Puzzled, Entering, Consulting*} = *Puzzled*
>   **var** *c*: 0 .. 3 = 0
>   **method** *puzzled*()
> $s = Puzzled \rightarrow c := c + 1$; **if** $c = 3$ **then** ($s := Entering$; *st.puzzled*())
>   **method** *enter*()
> $s = Entering \rightarrow s := Consulting$; *st.enter*()
>   **method** *consult*()
> $s = Consulting \rightarrow c := c - 1$; **if** $c > 0$ **then** $s := Entering$ **else** $s := Puzzled$; *st.consult*()

> **class** *Elf₅*(*sh*: *Shop*)
>   **action** *sh.puzzled*() ; *sh.enter*() ; *sh.consult*()

One shop and 20 elves are created:

> $sh :=$ **new** *Shop₅* ; **for** $i := 1$ **to** 20 **do new** *Elf₅*(*sh*)

The coupling relation between *Shop₄* and *Shop₅* includes the identity relation on *s*. The count *c* is also identical except in state *Puzzled*, as in *Shop₅* the elves may increment *c* one by one but in *Shop₄* it is set to 3 at once:

$$R_5(s_4, c_4)(s_5, c_5) \;\widehat{=}\; s_4 = s_5 \wedge (c_4 = c_5 \vee (s_5 = Puzzled \wedge c_5 < 3))$$

Class *Shop₅* refines *Shop₄* as the new method *puzzled* stutters under R_5 if $c_5 < 2$ and refines the action $s_4 = Puzzled \rightarrow s_4, c_4 := Entering, 3; st.puzzled()$ if $c_5 = 2$, new method *enter* refines $s_4 = Entering \rightarrow s_4 := Consulting; st.enter()$, and new method *consult* refines $s_4 = Consulting \rightarrow c_4 := c_4 - 1; \textbf{if } c_4 > 0 \textbf{ then } s_4 := Entering \textbf{ else } s_4 := Puzzled; st.consult()$. Formally, the conditions are:

(**I**) $R_5(Puzzled, 0)(Puzzled, 0)$

(**N₁**) *skip* $\sqsubseteq_{R_5}$ *Shop₅.puzzled*   or $s_4 = Puzzled \rightarrow s_4, c_4 := Entering, 3; st.puzzled()$ $\sqsubseteq_{R_5}$ *Shop₅.puzzled*

(**N₂**) $s_4 = Entering \rightarrow s_4 := Consulting; st.enter()$ $\sqsubseteq_{R_5}$ *Shop₅.enter*

(**N₃**) $s_4 = Consulting \rightarrow c_4 := c_4 - 1; \textbf{if } c_4 > 0 \textbf{ then } s_4 := Entering \textbf{ else } s_4 := Puzzled; st.consult()$ $\sqsubseteq_{R_5}$
    Shop₅.consult

Summary of Refinement Steps

The final versions of classes *Santa*, *Sleigh*, *Reindeer*, *Shop*, and *Elf* are:

> **class** *Santa*
>   **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* := *Welcoming* **else** *s* := *Sleeping*
>   **action** *s* = *Sleeping* ∧ *b* → *s* := *Harnessing*
>   **action** *s* = *Sleeping* ∧ *p* = 3 ∧ ¬ *b* → *s* := *Welcoming*

> **class** *Sleigh*(*st*: *Santa*)
>   **var** *s*: {*Back, Harnessing, Pulling*} = *Back*
>   **var** *c*: 0 .. 9 = 9
>   **method** *back*()
>   *s* = *Back* → *c* := *c* − 1 ; **if** *c* = 0 **then** (*s*, *c* := *Harnessing*, 9 ; *st.back*())
>   **method** *harness*()
>   *s* = *Harnessing* → *c* := *c* − 1 ; **if** *c* = 0 **then** (*s*, *c* := *Pulling*, 9 ; *st.harness*())
>   **method** *pull*()
>   *s* = *Pulling* → *c* := *c* − 1 ; **if** *c* = 0 **then** (*s*, *c* := *Back*, 9 ; *st.pull*())

> **class** *Reindeer*(*sl*: *Sleigh*)
>   **action** *sl.back*() ; *sl.harness*() ; *sl.pull*()

> **class** *Shop*(*st*: *Santa*)
>   **var** *s*: {*Puzzled, Entering, Consulting*} = *Puzzled*
>   **var** *c*: 0 .. 3 = 0
>   **method** *puzzled*()
>   *s* = *Puzzled* → *c* := *c* + 1 ; **if** *c* = 3 **then** (*s* := *Entering*; *st.puzzled*())
>   **method** *enter*()
>   *s* = *Entering* → *s* := *Consulting* ; *st.enter*()
>   **method** *consult*()
>   *s* = *Consulting* → *c* := *c* − 1 ; **if** *c* > 0 **then** *s* := *Entering* **else** *s*:= *Puzzled* ; *st.consult*()

> **class** *Elf*(*sh*: *Shop*)

Repetitions of Santa	Lime (guards)	C (semaphores)	Go (channels)	Java (monitors)
10,000	0.04 / 0.04 / 0.00	0.87 / 0.26 / 1.18	0.08 / 0.12 / 0.01	6.38 / 2.48 / 5.30
100,000	0.30 / 0.30 / 0.00	8.82 / 2.50 / 12.0	0.77 / 1.18 / 0.06	60.3 / 21.6 / 52.0
1,000,000	2.91 / 2.90 / 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.

> **action** *sh.puzzled*() ; *sh.enter*() ; *sh.consult*()

The main program creates active objects for Santa, reindeer, and elves; these use the passive sleigh and shop objects for synchronization:

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

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 close to three times faster than Go, about 32 times faster than C, and more than 180 times faster than Java when measuring elapsed time. 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, doi:10.1007/978-0-387-35562-7_3.

[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 := Welcoming
                else
                    s := Sleeping
        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: {Back, Harnessing, Pulling}
    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, c := Harnessing, 9
                st.back()
    method harness()
        when s = Harnessing do
            c := c − 1
            if c = 0 then
                s, c := Pulling, 9
                st.harness()
    method pull()
        when s = Pulling do
            c := c − 1
            if c = 0 then
                s, c := Back, 9
                st.pull ()

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

sl . pull ()

class *Shop*
    var *s*: {*Puzzled, Entering, Consulting*}
    var *c*: **int**
    init (*santa*: *Santa*)
        s, *c*, *st* := *Puzzled*, 0, *santa*
    method *puzzled*()
        when *s* = *Puzzled* **do**
            c := *c* + 1
            if *c* = 3 **then**
                s := *Entering*
                st . puzzled ()
    method *enter*()
        when *s* = *Entering* **do**
            s := *Consulting*
            st . enter ()
    method *consult*()
        when *s* = *Consulting* **do**
            c := *c* − 1
            if *c* > 0 **then**
                s := *Entering*
            else
                s := *Puzzled*
            st . consult ()

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

```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 () ;}
    }
}
```