Spectra Example: Dining Philosophers

Ilia Shevrin

Tel Aviv University

Abstract. The Dining Philosophers is a well-known problem in concurrent design, used to illustrate the difficulties in developing concurrent algorithms [1]. This report presents a formal specification of the problem in the Spectra language [4]. We chose to model the solution as a system controlled arbitrator that must grant forks to the environment controlled hungry philosophers.

1 Introduction

The Dining Philosophers is a well-known problem in the field of concurrent algorithms design, originally formulated by Edsger Dijkstra [1]. The basic statement consists of a round table with n bowls of spaghetti. n forks are placed between each pair of bowls and every philosopher is seated beside one bowl. To eat, a philosopher must use both forks, but she may take them only if they are not being used by a neighboring philosopher. The philosophers are silent, i.e., no philosopher can announce to the table when does she intend to start or stop eating.

The problem is to describe a behavior for the philosophers such that no philosopher will starve, i.e., each will eat infinitely often, assuming an unlimited supply of spaghetti.

We present a Spectra [4] specification for the problem and a simulation of its solution. All related files are available from `http://smlab.cs.tau.ac.il/syntech/`.

2 The Specification

We model the solution as an arbitrator that takes requests from the philosophers and serves them both forks at once. When a philosopher finishes eating, she simply puts down the forks. The essence of the solution is the guarantee that no philosopher is going to starve, i.e., whenever a philosopher is hungry, she is eventually served.

We define a state enum for a fork (line 1 in List. 1.1). It consists of three states: `FREE`, which means that the fork is on the table, and `RIGHT` (`LEFT`), which means that the fork is taken by a philosopher sitting to its right (left). We set `N` to 5 (line 2), in order to conform with the number of philosophers in the original problem, but the specification is parametric and can be easily fixed to support any number of philosophers.

```
1 type State = {FREE, LEFT, RIGHT};
2 define N := 5;
3 env boolean[N] hungry;
4 sys State[N] forks;
5 predicate hasTwoForks(Int(0..(N-1)) i):
6   forks[i%N] = RIGHT & forks[(i+1)%N] = LEFT;
```

Listing 1.1. Spectra specification of the Dining Philosophers: defines and variable declarations

```
1 asm keepBeingHungry: G forall i in Int(0..(N-1)) .
2   (hungry[i] & !hasTwoForks(i)) -> next(hungry[i]);
```

Listing 1.2. Spectra specification of the Dining Philosophers: safety assumptions

The environment in our example controls the philosophers. It maintains an array of size `N` that keeps track which philosopher is hungry at the moment. This constitutes as asking permission from the arbitrator to eat. We model it using the variable `hungry` (line 3).

The system controls the arbitrator. Its job is to grant every hungry philosopher permission to eventually pick up both her right and left forks, once they become available, so that she will eat and fulfill her request. We model it using the variable `forks` (line 4).

We define a useful predicate to indicate when a philosopher holds both forks, so she may eat and fulfill her request in `hasTwoForks` (lines 5-6).

2.1 Assumptions

Every single philosopher, acting as a requesting agent, is assumed to keep asking permission for the forks as long as her hunger is not fulfilled, i.e., she does not withdraw her request (`keepBeingHungry` in listing 1.2). We also assume that a philosopher who holds both forks, will eventually stop being hungry (`stopsEating` in List. 1.3). This is modeled with the LTL formula $G(s \rightarrow Fp)$. As this formula is not in pure GR(1), its expression in Spectra is achieved via language constructs called patterns [2] (the pattern `pRespondsToS` in our example). Note the import statement at the beginning of the specification.

Observe that the liveness assumption `stopsEating` is parametric in the number of philosophers, and is in fact unrolled into `N` assumptions. This is an example of a Spectra's language construct that frees the writer of the specification from repeating a very similar formula `N` times and of having to edit it whenever `N` is changed.

2.2 Guarantees

The system, acting as a granting agent, should most importantly guarantee absence of starvation. It maintains liveness on account of each philosopher with

```
1 asm stopsEating{Int(0..(N-1)) i}:
2   pRespondsToS(hasTwoForks(i), !hungry[i]);
```

Listing 1.3. Spectra specification of the Dining Philosophers: liveness assumptions

```
1 gar eventuallyEats{Int(0..(N-1)) i}:
2   pRespondsToS(hungry[i], hasTwoForks(i));
```

Listing 1.4. Spectra specification of the Dining Philosophers: liveness guarantees

another instance of the response pattern $G(s \rightarrow Fp)$. Every hungry philosopher eventually gets a right and a left fork and can start eating, or in other words, no philosopher will be hungry forever (`eventuallyEats` in List. 1.5). Again, this is a parametric liveness guarantee.

Further, the system should not take away the forks from a hungry philosopher, as long as she is hungry (`alwaysKeepForksWhileEating` in line 1 in List. 1.4). It also guarantees to serve both forks at once or none at all (`alwaysGiveLeftAndRightTogether` in line 4). Moreover, we require that a fork switches hands only by being free on the table in a middle step (`mustPutDownFork` in line 8).

Finally, we require that the arbitrator will never give forks to non-hungry philosophers, i.e., it will give no redundant grants (`dontGiveForkIfNoHungry` in line 12).

3 Example Analyses

3.1 Realizability and Unrealizability Analysis

In addition to synthesizing a controller, one may apply different analyses to study the specification and explore the solution. The basic analysis one may apply is to check whether the specification is realizable. To check this use Spectra / Check Realizability. Our specification is indeed realizable.

Assume now that assumption `stopsEating` is missing, which, as one can check, renders the specification unrealizable. To better understand the reason for the unrealizability we use Spectra / Synthesize Concrete Counter-Strategy. Studying the state machine description Spectra outputs, we learn that the environment may keep philosopher i infinitely hungry, as long as the system serves her forks, and at the same time request spaghetti for philosopher $i-1$. Observe that in order to satisfy eventually `eventuallyEats`, the system must inevitably violate `alwaysKeepForksWhileEating`.

One might also be intersted in a minimal set of guarantees that account for specification unrealizability, i.e., an unrealizable core. This can be achieved via Spectra Add-ons / Cores / Find an Unrealizable Core. Unsurprisingly, the output is a core consisting of only two guarantees, `alwaysKeepForksWhileEating` and `eventuallyEats`.

```
gar alwaysKeepForksWhileEating: G forall i in Int(0..(N-1)) .
  (hasTwoForks(i) & hungry[i]) -> (next(hasTwoForks(i)));

gar alwaysGiveLeftAndRightTogether:
  G forall i in Int(0..(N-1)) .
    (forks[i] = RIGHT <-> forks[(i+1)%N] = LEFT);

gar mustPutDownFork: G forall i in Int(0..(N-1)) .
  (forks[i] = LEFT -> next(forks[i]) != RIGHT) &
    (forks[i] = RIGHT -> next(forks[i]) != LEFT);

gar dontGiveForkIfNoHungry: G forall i in Int(0..(N-1)) .
  (!hungry[i] ->
  (next(forks[i] != RIGHT) & next(forks[(i+1)%N] != LEFT)));
```

Listing 1.5. Spectra specification of the Dining Philosophers: safety guarantees

3.2 Vacuity Analysis

It is possible that the specification contains vacuous assumptions or guarantees, i.e., formulas that are logically implied from other formulas in the specification [5]. Some vacuities may be more difficult to spot than others. To check for system (resp. environment) vacuities use Spectra Add-ons / GR(1) Vacuity / Find System (resp. Environment) Module Vacuities. We discover that there are no vacuities in our specification.

Assume now that our specification contains two additional guarantees. First, one may define a mutual exclusion guarantee preventing any two adjacent philosophers from eating at the same time (`adjacentNeverEatTogether` in List. 1.6). Obviously, if a fork is currently assigned to an eating philosopher as the right fork, it cannot be assigned to another philosopher as a left fork at the same time (since its value can be `RIGHT` or `LEFT` (or `FREE`), but not both. Indeed, the vacuity analysis reports that this guarantee is vacuous. Further, in the analysis output we see that the vacuity core of this vacuous guarantee is the empty set, i.e., it is implied from the definition of the fork state enum, and not from other assumptions or guarnatees in the specifications. Such vacuities are called trivial vacuities (see [5]).

Second, one may define a parameteric justice guarantee to express that eventually every philosopher will not eat (`eventuallyNotEating` in List. 1.6). In this case, it is perhaps less easy to see whether these guarantees are vacuous. We run Spectra's vacuity analysis and find out that it is indeed vacuous. We further see that the vacuity core consists of two formulas, the assumption `stopsEating` and the guarantee `dontGiveForkIfNoHungry`. Intuitively, we explain the vacuity by noting that every philosopher must eventually fulfill her hunger, and following that, the system guarantees to stop serving her forks.

```
gar adjacentNeverEatTogether: G forall i in Int(0..(N-1)) .
  hasTwoForks(i) ->
    (!hasTwoForks(i+1) & !(hasTwoForks(i-1)));

gar eventuallyNotEating{Int(0..(N-1)) i}: GF !hasTwoForks(i);
```

Listing 1.6. Spectra specification of the Dining Philosophers: vacuous guarantees

3.3 Well-Separation Analysis

One may be interested in checking whether the specification is well separated [3]. To check this use Spectra Add-ons / Well-Separation / Diagonse Well-Separation of Environment.

Spectra provides two notions of well-separation, with and without taking system safeties into account. We observe that when we take system safeties into account, the specification is indeed well-separated, but in the stricter sense, when system safeties are ignored, the specification is actually non-well-separated. This means that the system must violate its own guarantees first in order to force the environment to violate its assumptions.

To find out more we use Spectra Add-ons / Well-Separation / Compute Non-Well-Separation Counter-Strategy. Studying the state machine description that Spectra outputs, we learn that the system may violate `alwaysKeepForksWhileEating` guarantee and take away the forks from some philosopher i while she is still eating. Then, on the one hand, the philosopher must keep being hungry according to `keepBeingHungry`, and on the other hand, must stop being hungry eventually according to `stopsEating` for some i.

Acknowledgements

We thank Keren Solodkin for implementing Spectra's support for parametric language constructs. This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 638049, SYNTECH).

References

1. Dijkstra, E.W.: Hierarchical ordering of sequential processes. Acta Inf. **1**, 115–138 (1971). https://doi.org/10.1007/BF00289519, https://doi.org/10.1007/BF00289519
2. Maoz, S., Ringert, J.O.: GR(1) synthesis for LTL specification patterns. In: ESEC/FSE. pp. 96–106. ACM (2015)
3. Maoz, S., Ringert, J.O.: On well-separation of GR(1) specifications. In: Zimmermann, T., Cleland-Huang, J., Su, Z. (eds.) Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016. pp. 362–372. ACM (2016), http://doi.acm.org/10.1145/2950290.2950300

4. Maoz, S., Ringert, J.O.: Spectra: A specification language for reactive systems. CoRR **abs/1904.06668** (2019), http://arxiv.org/abs/1904.06668
5. Maoz, S., Shalom, R.: Inherent vacuity for GR(1) specifications. In: ESEC/FSE. ACM (2020), to appear.

A Complete Specification

```
import "DwyerPatterns.spectra"
module DiningPhilosophers

type State = {FREE, LEFT, RIGHT};
define N := 5;
env boolean[N] hungry;
sys State[N] forks;
predicate hasTwoForks(Int(0..(N-1)) i):
  forks[i%N] = RIGHT & forks[(i+1)%N] = LEFT;

asm initialNoHungry: forall i in Int(0..(N-1)) . !hungry[i];

asm keepBeingHungry: G forall i in Int(0..(N-1)) . (hungry[i]
     & !hasTwoForks(i)) -> next(hungry[i]);

asm stopsEating{Int(0..(N-1)) i}: pRespondsToS(hasTwoForks(i)
    , !hungry[i]);

gar initialForksFree: forall i in Int(0..(N-1)) . forks[i] =
    FREE;

gar alwaysKeepForksWhileEating: G forall i in Int(0..(N-1)) .
     (hasTwoForks(i) & hungry[i]) -> (next(hasTwoForks(i)));

gar alwaysGiveLeftAndRightTogether: G forall i in Int(0..(N
    -1)) . (forks[i] = RIGHT <-> forks[(i+1)%N] = LEFT);

gar mustPutDownFork: G forall i in Int(0..(N-1)) . (forks[i]
    = LEFT -> next(forks[i]) != RIGHT) & (forks[i] = RIGHT ->
     next(forks[i]) != LEFT);

gar dontGiveForkIfNoHungry: G forall i in Int(0..(N-1)) . (!
    hungry[i] -> (next(forks[i] != RIGHT) & next(forks[(i+1)%
    N] != LEFT)));

gar eventuallyEats{Int(0..(N-1)) i}: pRespondsToS(hungry[i],
    hasTwoForks(i));
```

Listing 1.7. Complete Spectra specification of the Dining Philosophers