Journal of Laboratory Automation
2014, Vol. 19(6) 569–576

DOI: 10.1177/2211068214543373
jala.sagepub.com
SAGE

Laboratory Automation in a Functional Programming Language

Colin Runciman[1], Amanda Clare[2], and Rob Harkness[3]

Abstract

After some years of use in academic and research settings, functional languages are starting to enter the mainstream as an alternative to more conventional programming languages. This article explores one way to use Haskell, a functional programming language, in the development of control programs for laboratory automation systems. We give code for an example system, discuss some programming concepts that we need for this example, and demonstrate how the use of functional programming allows us to express and verify properties of the resulting code.

Keywords

programming, informatics and software, robotics and instrumentation, engineering, high-throughput screening, automated biology

Introduction

There are many different types of software applications in the field of laboratory automation. There are stand-alone applications for controlling a simple instrument such as a bulk-reagent dispenser. There are also larger software packages for controlling automated robotic systems with many instruments that are linked to data management systems.[1] These software packages are typically referred to as schedulers.

Currently, popular languages for laboratory automation applications include Java, C, C++, and C#. In our experience, the majority of such applications have been developed using these languages, along with other .NET variants such as Visual Basic.NET.[2]

These languages are commonly known as procedural or imperative languages. They describe the commands to use in a sequential manner, to achieve the intended functionality. They change state, such as the value of variables, along the way, and the values of these variables can dictate the flow of execution. Although languages such as Java and C++ use different syntax, the general principles of constructing code remain the same.

An imperative approach is the most common way to develop an application. However, different programming styles can be adopted, one of these being the functional approach. Functional languages, such as O'Caml and Haskell, are not as well known as C or C#, especially among programmers from a non–computer science background. However, since Microsoft released Visual Studio 2010 with the inclusion of F# and the increased adoption of Scala, there has been an increasing awareness of functional languages among commercial application developers. Originally a research project at Microsoft that based itself on O'Caml, F# has developed into a functional language that interacts with the Microsoft .NET library.[3] Indeed, anyone who has used the .NET Framework 3.5 is quite likely to be familiar with some functional language concepts without realizing it. For example, the design of LINQ queries within the .NET Framework was based on the use of anonymous functions within Haskell.

The remainder of this article provides a complete and executable program that implements a scheduler for laboratory automation. Along the way, we gently introduce the Haskell programming language and point out the properties that are declared in the code. We start by defining types, move on to define auxiliary functions, build up the scheduler, and finish with a section on automated testing of properties of interest.

The Scheduling Problem

Scheduling is an important component of a laboratory automation software package.[4,5] The benefit of using laboratory

[1]Dept. of Computer Science, University of York, York, UK
[2]Dept. of Computer Science, Aberystwyth University, Aberystwyth, UK
[3]PAA, Farnborough, UK

Received Feb 26, 2014.

Corresponding Author:
Rob Harkness, PAA, 6 Armstrong Mall, Southwood Business Park, Farnborough, GU14 0NR, UK.
Email: rob.harkness@paa-automation.com

automation is that multiple plates within an experiment or assay can be processed automatically. To improve efficiency, the schedule must allow multiple plates to run at the same time so the execution is interleaved, which then maximizes throughput.

One simple approach is to use an event-driven scheduling system. This works by assessing the state of the system on a continual cycle. After each event, the scheduler, or processing engine, determines a course of action for the system to run as quickly as possible without breaking constraints such as incubation times and the maximum number of plates allowed on each device. For example, if a plate is due out of an incubator, this task is given priority over adding a new plate into the system. The advantage of an event-driven system is that the assay can follow different processing paths based on events, such as acquired data or the failure of an instrument in the system.

However, with an event-driven scheduler, there is the possibility of encountering scheduling deadlocks. A simple example of a deadlock is the following:

- Plate 1, sitting on instrument A, needs to move to instrument B and
- Plate 2, sitting on instrument B, wants to move to instrument A.

It is not possible for either plate to move on to the next step within its respective workflow, so there is deadlock. Deadlock situations can involve more plates and instruments, but the basic problem is the same: it is not possible to unblock key resources to allow the workflow for each plate to be processed.

In the remainder of this article, we illustrate the use of functional programming as a style of programming that can help in defining control software for laboratory automation. This will bring out many of the distinctive aspects of functional programming as we develop the code. To make this illustration, we use a particular hardware setup that can be found in many laboratories. The system has an input stack, an output stack, and some number of washers and dispensers, as shown in **Figure 1**.

Here are some of the properties we want our laboratory scheduler to have:

1. each plate has a workflow;
2. each device, including the robot, requires a specified period to do its job;
3. each plate progresses through its workflow in a timely manner; and
4. the whole system is deadlock free.

Using a functional programming language allows us to write in a style that can express and verify such properties rather than just write code. Properties 1 and 2 can be expressed in types for plates and devices, statically checked by a compiler. Properties 3 and 4 can be expressed in property functions and checked by property-based testing tools such as QuickCheck[6] or SmallCheck.[7]

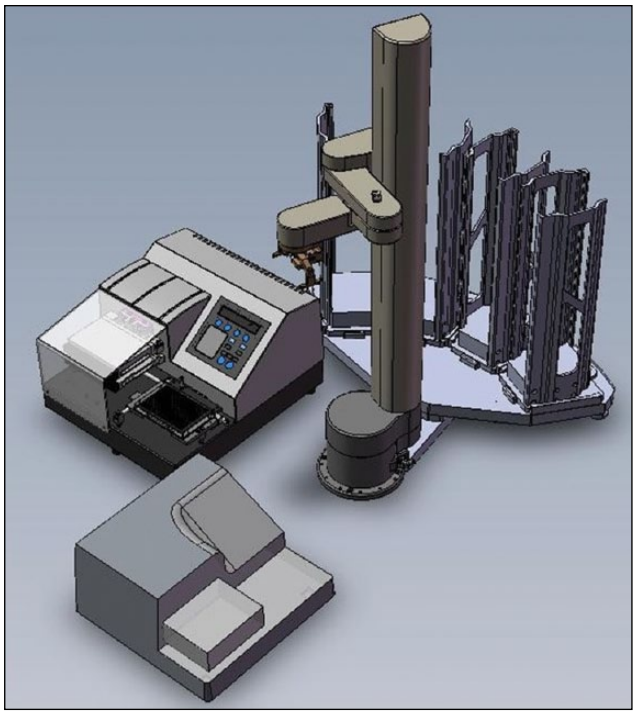

Figure 1. An example system with input stack, output stack, robot arm, washer, and dispenser. This is the simplest type of automated platform whereby more than one plate will be active on the system at the same time. With this platform, an optimal schedule will have the washer and the dispenser occupied simultaneously.

By formulating properties in this way, developers can capture general rules about the required behavior of a system, not just specific cases and fragments represented by unit tests. Computing power is harnessed to search the space of possible test inputs automatically, looking for cases in which one of the specified properties fails. The technique is also known as "lightweight verification" since it is the next best thing to a rigorous mathematical verification that all the formulated properties hold in all cases.

Method and Results

Devices and Workflows

First we must choose how to represent the kinds of devices found in a laboratory, such as washer and dispenser. This choice is reflected in the definition of our first data type.

One can think of a type as a description of a set of possible values, or equivalently a type is a property that any value may or may not have. Our first type is `DeviceKind`. It has four possible values for the four kinds of devices in

our laboratory. Only these values have the property that they are of type `DeviceKind`.

```
> data DeviceKind = Washer | Dispenser | InStack | OutStack
>                                   deriving (Eq, Show)
```

Informally, the vertical bar can be read "or": so a value of type `DeviceKind` is a `Washer` or a `Dispenser` or an `InStack` or an `OutStack`.

The deriving clause gives us two properties of the `DeviceKind` type: it belongs to the type-class Eq (so its values can be compared for equality), and it belongs to the type-class `Show` (so its values can be printed as strings). A type-class is similar to an interface in Java or C#, for which we must provide implementations of functions. In using the keyword *deriving*, we accept the default implementations for this type.

With the `DeviceKind` type defined, one simple representation of a laboratory workflow, sufficient for the purposes of this article, is a list of devices that a microtiter plate must go to in turn. Lists are a built-in data type in Haskell. The way that lists are defined guarantees that items in the same list are of the same type.

```
> type Workflow = [DeviceKind]
```

Here `Workflow` is defined as a synonym for a list of `DeviceKind`. We define the following example workflow for use in later tests.

```
> exampleWorkflow :: Workflow
> exampleWorkflow = [InStack, Washer, Dispenser,
>                    Washer, OutStack]
```

One simple but useful function on lists is `null`, which tests its list argument for emptiness. Here's how we can use it to define `nonEmpty`, a function that tests for a nonempty list.

```
> nonEmpty xs = not (null xs)
```

Function applications are frequent in functional programs so the notation needs to be light. In Haskell, we just write a function name and then each of the input arguments in turn. No extra symbols such as brackets or commas are needed. The brackets in `not (null xs)` merely indicate priority: without them, `not null xs` would apply the function `not` to the two arguments `null` and `xs`.

We shall often make use of the infix colon operator for constructing nonempty lists. The list `e:rest` contains a first element `e`, followed by a (possibly empty) list `'rest'` of other elements.

We need some representation of time. For example, we must represent the time needed for processing by each device and for transfer of plates between devices. For the purposes of this article, a time value is simply an integer representing a number of "ticks." Whether ticks are milliseconds, seconds, or something else need not concern us.

```
> type Time = Int
```

There may be more than one device in the laboratory of the same kind (for example, we may have two washers). So we also define a further type whose values represent specific devices. A specific `Device` is represented by a combination of a `DeviceKind` value, an integer to distinguish this device from others of the same kind, the length of time for this device to process a plate, and the length of time for a robot arm to move a plate between this device and a central safe location. For example, if we have two washers in our system, they might be represented by the values `Device Washer 1 3 2` and `Device Washer 2 3 3`.

```
> data Device = Device {devKind          :: DeviceKind,
>                       devNo            :: Int,
>                       devProcT         :: Time,
>                       devMoveT         :: Time}
>                                   deriving Eq
```

The above definition describes the fields of a `Device`, giving them names and types. It also provides automatic field accessor functions, which can be used to inspect the values or provide new values for the fields. As an example, the `devProcT` for a device `d` could be accessed with the expression `devProcT d` and a copy of a device `d` with a new devNo could be created by `d' = d {devNo = 4}`.

Rather than deriving an automated default for the printing of Device values (which would render them as, for example, "Device Washer 6 3 2"), we define our own custom instance, omitting the constructor name `Device` and also the timing details.

```
> instance Show Device where
>    show (Device d n p m) = show d ++ " " ++ show n
```

When we come to define scheduling, the workflow just specifies a `DeviceKind` but the scheduler must allocate a specific `Device`. We capture the `isA` relationship between devices and device-kinds in the following definition.

```
> isA :: Device -> DeviceKind -> Bool
> isA (Device sd n p m) d   =   sd == d
```

Functions are values too, and they have types. The types declare properties about the function, which can be statically checked by the compiler before the program is run. The first line describes the type of this function. The arrows can be read as logical implications. If the first argument is a value of type `Device`, then if the second is a value of type `DeviceKind`, the result is a value of type `Bool`, a predefined type with the two values, `True` and `False`. Although we can choose to declare the type of a function, in most cases, a Haskell compiler can automatically derive this information, so the programmer need not provide it. However, we might choose to provide a type declaration to check that our understanding of the function's properties agrees with that derived by the compiler or just to assist with code readability.

Note that the infix "==" is a function that tests values for equality, not to be confused with the single = symbol used to define a function. We can also choose to use infix notation when applying named functions: for example, we can write `sd `isA` d` rather than `isA sd d`, and the infix version makes the roles of `sd` and `d` clearer.

Plates and Locations

We are working toward a representation of the complete state of the laboratory. So far we have a representation for devices but not for the plates that are processed by these devices or for the robot arm that moves plates between them.

Our next step is to introduce a type to represent the possible locations of plates in the laboratory. A plate's location is either at a device or in transit (by means of the robotic arm) between two devices. Rather than a long-winded constructor name such as `InTransitByRoboticArm`, an infix constructor `:->` gives us a more convenient notation and makes the source and destination clearer.

```
> data Loc = At Device | Device :-> Device deriving (Eq, Show)
```

Example `Loc` values in their printed form include `At Washer 3` and `Washer 3 :-> Dispenser 1`.

Now we can define the data type for `Plates`.

```
> data Plate = Plate { plateNo      :: Int,
>                      plateLoc     :: Loc,
>                      plateSince   :: Time,
>                      plateFlow    :: Workflow }
>                           deriving (Eq)
```

The `plateNo` is a number uniquely identifying this plate: each plate is allocated a number as it enters the system. The `plateLoc` specifies the current location. The `plateSince` represents the time at which either the plate arrived (for `At` locations) or the transfer began (for `:->` locations). The plateFlow is the remaining list of the kinds of devices this plate must visit.

When we show a plate as a string, it is usually more convenient to omit the details of the remaining workflow for the plate.

```
> instance Show Plate where
>   show (Plate no loc since w) =
>    "plate "++show no++", "++show loc++" since "++show since
```

Two simple "helper" functions extract information from a `Plate` value.

```
> inTransfer :: Plate -> Bool
> inTransfer (Plate _ (_ :-> _) _ _) = True
> inTransfer _ = False

> plateDestination :: Plate -> Device
> plateDestination (Plate _ (_ :-> d) _ _) = d
```

Notice that we don't have to give names to every component in the `Plate` value. When a function does not need to refer to a component, we write _ in the argument pattern.

With representation types in hand for devices, time, and plates, we can complete the data model for the state of the laboratory automation system with the following data type declaration and associated Show instance.

```
> data SysState = SysState {sysPlates :: [Plate],
>                           sysDevs   :: [Device],
>                           sysTime   :: Time}
> instance Show SysState where
>   show (SysState ps ds t) =
>        "t = " ++ show t ++ ": " ++ show ps
```

The time in each `SysState` is the time at which that state exists. The plates list represents all the plates in the system at that time, including those in the `OutStack` for which the workflow has been completed. The devices list represents every device that is in use or available for use at that time.

Events and Scheduling Definition

We shall model the laboratory process as an event-driven system. By now it should be no surprise that we want to introduce a new data type, this time to model the four kinds of event that can occur.

```
> data InEvent = Tick | NewPlate | DeviceUp Device
>                | DeviceDown Device
>    deriving Show
```

A `Tick` event indicates the passage of time. A `NewPlate` event represents the introduction of a new plate into the system. A `DeviceUp` event represents the addition of a device, either by initial powering up and initialization or by the repair of a previously faulty device. A `DeviceDown` event represents the failure or removal of a device, which then becomes unavailable.

Now we can define the type of a Scheduler for the laboratory as follows:

```
> type Scheduler = InEvent -> SysState -> SysState
```

Auxiliary Functions

We shall work toward the definition of an appropriate function of this type. To prepare the way, we shall first define some auxiliary functions to compute information that any scheduler could be expected to need. We shall then define an example scheduler. Importantly, we can define and compare many different schedulers. One property they must all share is the `Scheduler` type, which makes it type-safe to plug in such code. We shall see some examples of the properties that can be analyzed and compared later.

First a scheduler must be able to determine whether a device is currently free.

```
> freeDevice :: SysState -> Device -> Bool
> freeDevice s (Device OutStack _ _ _) = True
> freeDevice (SysState ps ds _) d =
>          d `elem` ds
>          && null [p | p <- ps, plateLoc p == At d]
```

The first equation reflects our assumption that an `OutStack` cannot fail. Note that this assumption is not encoded in the type system: a `DeviceKindDown` event for an `OutStack` would pass the type-checker. We also assume that an `OutStack` has infinite capacity, so is always free. Devices other than the `OutStack` may fail and the `d `elem` ds` condition checks whether the device is up. The other devices are also assumed to have a capacity of a single plate, so they are only free if there is not already a plate at the device.

The expression `[p | p <- ps, plateLoc p == At d]` is a *list comprehension*. An informal reading of this particular comprehension would be "the list of all elements `p` with two qualifications: first `p` is an item of the list `ps`, and second `p` satisfies the condition `plateLoc p == At d`." The first kind of qualification is termed a *generator* and the second a *filter*, and in general a *comprehension* may have any number of qualifications of each kind. List comprehensions are a compact and powerful way to express many lists. First introduced in functional languages, they have since been adopted in many others, including Javascript, Python, and LINQ, within the .NET environment.

Since the robot arm has a special status and is not modeled in the same way as other devices, a scheduler also needs a function to check for the availability of the robot arm.

```
> robotArmFree :: SysState -> Bool
> robotArmFree (SysState ps _ _) =
>   null [p | p <- ps, inTransfer p]
```

This definition reflects a few assumptions. There is always exactly one robot arm. Unlike other devices that are affected by `DeviceUp` and `DeviceDown` events, the robot arm does not have to be initialized, and it cannot fail. It is free if it is not currently transferring a plate between devices.

Scheduling Functions

Having defined auxiliary functions to test whether devices are ready to participate in moves, we next consider the readiness of plates. The following function checks whether a plate is ready to be moved from its current location.

```
> ready :: Plate -> Time -> Bool
> ready p t = t >= plateSince p + timing (plateLoc p)
>     where
>     timing (At d)       = devProcT d
>     timing (d1 :-> d2)  = devMoveT d1 + devMoveT d2
```

A plate being processed by a device is ready if enough time has elapsed for the device to complete its process. A plate being moved by a robot arm is ready if enough time has elapsed for the required movements to and from the central safe position.

An `InEvent` determines a two-stage transition between current and next system states.

The first stage of this transition reflects the unavoidable consequences of the event: time advances, a new plate is added, a device goes down, or a device comes up (the `effectOf` function); in addition, if the time required for a robot arm transfer has elapsed, then the plate is delivered to the destination device (the `putPlateIfReady` function).

The second stage of the transition is then determined by the choices of a specific scheduler.

```
> consequences :: InEvent -> SysState -> SysState
> consequences ie s = putPlateIfReady (effectOf ie s)

> effectOf :: InEvent -> SysState -> SysState
> effectOf Tick s         = s {sysTime = sysTime s + 1}
> effectOf NewPlate s     = s {sysPlates = newPlate s : sysPlates s}
> effectOf (DeviceDown d) s = s {sysDevs = (sysDevs s) \\ [d]}
> effectOf (DeviceUp d) s  = s {sysDevs = (sysDevs s) `union` [d]}
```

The infix functions `\\` and `` `union` `` for list difference and list union are defined in the standard Haskell library Data.List.

```
> newPlate :: SysState -> Plate
> newPlate (SysState ps ds t) = Plate (length ps + 1) newloc t w
>    where
>      newloc   = At (head [d | d <- ds, d `isA` wd])
>      (wd:w)   = exampleWorkflow
```

The pattern `(wd:w)` on the left-hand side of the last equation indicates that we expect `exampleWorkflow` to be a list, with a first item `wd`, followed by a possible empty list of other items `w`.

The `putPlateIfReady` function is a little more complex. Its key component is a function relocated that uses auxiliary functions already defined (`inTransfer`, `ready`, `freeDevice`) to change the location of plates where appropriate.

```
> putPlateIfReady :: SysState -> SysState
> putPlateIfReady s@(SysState ps ds t) =
>   s {plates = [relocated p (plateDestination p) | p <- ps]}
>   where
>   relocated :: Plate -> Device -> Plate
>   relocated p dp =
>     if inTransfer p && ready p t && freeDevice s dp
>         then Plate (plateNo p) (At dp) t
>              (tail (plateFlow p))
>         else p
```

The `@`-character used in the definition of this function's SysState parameter allows us to inspect and use the individual components of the `SysState` (the `ps`, the `ds`, and the `t`) but also to refer to it in its entirety as the variable `s`.

The expression `tail (plateFlow p)` represents progress in the workflow for a relocated plate. The workflow of a plate is held in memory as a shared list, referenced by all plates undergoing the same workflow. No item in the workflow list is destroyed by the application of `tail`; all items remain available in memory for other plates. The `tail` function simply returns a pointer to the next portion of the list, which is an efficient operation.

The second stage of the transition involves a choice. If there are plates ready to be moved on from one device to another, a scheduler must choose a source device, a ready plate at that device, and an appropriate destination for it.

The following function lists all the possible options from which to choose.

```
> plateMoveChoices :: SysState -> [SysState]
> plateMoveChoices s@(SysState ps ds t) =
>   [ s {plates = p' : (ps \\ [p])}
>   | d <- devices s, not (d `isA` OutStack),
>     p <- ps, plateLoc p == At d, ready p t,
>     d' <- nextDeviceChoices s p,
>     let p' = p {plateLoc = d :-> d', plateSince = t} ]
```

The `plateMoveChoices` function examines all the devices `ds` in the system. For each device `d`, apart from the `OutStacks`, it determines the plates `ps` that are at `d` and ready to be moved on. For each such plate `p`, we call nextDeviceChoices to work out the possible devices `d'` to which `p` could be transferred next. There are many potential plate move choices that could be evaluated here. However, Haskell is a lazy language and will only evaluate as many as necessary to find a solution.[8]

The primed variable names `d'` and `p'` are appropriate for derived values. This convention for the naming of variables is also widely used in mathematics for the same purpose.

The function `nextDeviceChoices` works out the list of possible next devices for a plate. It selects from the device list those of the appropriate kind that are currently free.

```
> nextDeviceChoices :: SysState -> Plate -> [Device]
> nextDeviceChoices s p =
>   [d | d <- ds, d `isA` dk, freeDevice s d]
>   where
>   (dk:_) = plateFlow p
>   ds = devices s
```

The Scheduler

Now we are ready to define a specific simple Scheduler. It simply chooses the first of all the available options.

```
> scheduler :: Scheduler
> scheduler ie s =
>   if robotArmFree s' then head (plateMoveChoices s' ++ [s'])
>   else s'
>   where
>   s' = consequences ie s
```

A more complex scheduler might analyze the workflow and the current state more deeply. It may be necessary, for example, to prioritize moves from devices with plates that have been waiting for the longest time or to relocate plates that require urgent incubation to avoid temperature changes.

Although the functions `consequences` and `effectOf` also have the `Scheduler` type, they are too limited to be useful schedulers by themselves. They only deal with the unavoidable consequences of an event and make no further decisions.

The entire laboratory process can now be represented by a function from a sequence of events and an initial system state to a sequence of system states. We can think of state sequences as a representation of the behavior of the laboratory automation system.

```
> run :: SysState  -> [InEvent] -> [SysState]
> run s []         = [s]
> run s (ie:ies)   = s : run (scheduler ie s) ies
```

This function run is defined recursively. When run is applied to a state and a list of input events containing a first input event, the result is a list of states, beginning with the original state. The other states in the list are produced by the application of run to the remaining input events, but run must now use the new state that resulted from the scheduler's decisions after dealing with that first input event. The resulting list of `SysStates` is produced lazily and will only be extended as the input events occur.

The states produced may be logged and immediately discarded or may be retained for further processing. At the moment, the output is a plate-centered view, but this could be changed to produce different system views as required. For example, we could derive from the `SysState` list either a device view or an event-log view.

In the initial system state, no plates have yet been supplied as input, and no devices are initialized. The time is "zero."

```
> initialSysState = SysState { plates = [], devices = [], time = 0}
```

The following example shows the use of the `run` function, with the above `initialSysState` as one argument and a sequence of input events as the other. This sequence of input events begins with the devices being initialized and then consists of an infinite cycle of a new plate addition and then two time ticks.

```
> eg :: [SysState]
> eg = run initialSysState
>            ( [DeviceUp d | d <- initialdevices]
>               ++ cycle [NewPlate, Tick, Tick] )
```

Now we give an example list of initial devices. We have six washers and two dispensers, with varying process times and access times. The order in which the devices are listed here influences the order in which available choices are listed by `nextDeviceChoices` and `plateMoveChoices`. So for the scheduler that simply selects the first choice, it affects the plate moves that are made.

```
> initialdevices :: [Device]
> initialdevices = [Device InStack 1 0 1] ++
>                  [Device Washer n 4 n | n <- [1..6]] ++
>                  [Device Dispenser n 2 n | n <- [1..2]] ++
>                  [Device OutStack 1 0 1]
```

Now that we have defined a scheduler that can make choices and chosen some initial devices with particular timings, we want some way to inspect the consequences of making those choices.

Properties and Automated Testing

Many intended properties of the component functions of a Haskell program can themselves be defined as functions with Bool results. Such property functions are, by convention, given names starting with `prop_`. They are expected to return `True` for all possible choices of correctly typed input arguments. Libraries such as QuickCheck[6] and SmallCheck[7] support automatic *property-based testing*. They exploit Haskell's type system to generate many possible values for a property's input arguments, test the property's result in each case, and report any failing case.

To illustrate, recall two important properties a well-designed scheduler should have:

- each plate progresses through its workflow in a timely manner—no state occurs in which a plate has been at a device for too long;
- the whole system is deadlock free—no state occurs in which at least one plate has an unfinished workflow, but there is nothing the system can do to make progress.

Both properties concern undesirable states that might arise after *any possible sequence of events*. So lists of `InEvents` are suitable input arguments for these properties, and we define them as follows:

```
> prop_OverstayFree :: Time -> [InEvent] -> Bool
> prop_OverstayFree maxDelay ies =
>    null [ s | s <- run initialSysState ies,
>               hasOverstayedPlate maxDelay s]

> prop_OverstayFree :: [InEvent] -> Bool
> prop_DeadlockFree ies =
>    null [ s | s <- run initialSysState ies,
>               isDeadlocked s]
```

In each case, the comprehension expresses a list of undesirable states. These lists should be empty.

A simple definition of an overstayed plate is one that has been at a device or in transit for longer than `maxDelay` clock ticks.

```
> hasOverstayedPlate maxDelay (SysState ps ds t) =
>    nonEmpty [p | p <- ps, t - plateSince p >= maxDelay]
```

It is trickier to specify just what we mean by a deadlocked system. A state is deadlocked if there is at least one plate in the system with a workflow not yet completed, but for no such plate is there (1) more time needed at the current location, (2) a free destination (for a plate in transfer), or (3) a possible choice of next device (for a plate ready to leave its current device).

So we define `isDeadlocked` as follows:

```
> isDeadlocked s@(SysState ps ds t) =
>   nonEmpty activePlates &&
>   null [p | p <- activePlates,
>         not (ready p t) ||
>         inTransfer p && freeDevice s (plateDestination p) ||
>         not (inTransfer p) && nonEmpty (nextDeviceChoices s p)]
>   where
>   activePlates = [p | p <- ps, nonEmpty (plateFlow p)]
```

We can now ask QuickCheck to check the properties `prop_OverstayFree` and `prop_DeadlockFree` for any particular system configuration.

Discussion

The code we have presented provides a complete and executable scheduler for an example laboratory automation system. The scheduler has various properties that can be verified by the type system and by property-based testing. Along the way, we have also defined many Boolean-valued functions (such as `isA`, `ready`, `inTransfer`, and `freeDevice`). Beyond their use in the program itself, these functions provide a useful vocabulary when formulating testable properties.

Writing properties is not always easy and requires the programmer to think deeply about the system. Writing properties for QuickCheck gives us two advantages: automating the testing and making the programmer's understanding of the system more explicit. The second advantage is just as important as the first for ensuring the correctness of the code. When writing the `isDeadlock` property, we first began by stating simpler criteria but soon discovered that our initial criteria did not capture all of the possible deadlock scenarios, and this forced a better understanding of what might cause deadlock. With the initially simpler deadlock criteria, QuickCheck reported the success of the test, but the action log showed plates that were not moving through the system. Inspecting this log led us to understand further deadlock-causing scenarios and to improve the description of deadlock.

We can use this code not only as a scheduler but also as a simulator to test out various equipment configurations and test for desired properties. If we find that a particular equipment configuration creates a potential deadlock, for example, it is easy to try specifying faster or extra equipment and retest `prop_DeadlockFree`.

The system we have described here has deliberately been kept simple in order to explain the concepts of functional programming with a concrete example. However, one could easily model variations—for example, a plate capacity for each device, different maximum-delay periods for different devices, different workflows for different plates, or multiple plates per workflow as in a reformatting liquid handling process.

Functional programming is a style that encourages high-level thinking about the specification and desired properties of a system, rather than low-level sequential programming of actions to be performed. In return for specifying and declaring properties, the programmer benefits from the

guarantees of type safety and automated property-based testing. One of our main purposes in writing this article is to encourage the wider adoption of such practices in laboratory automation.

Acknowledgments

We acknowledge the helpful comments of the anonymous referees for the article.

Declaration of Conflicting Interests

The authors declared no potential conflicts of interest with respect to the research, authorship, and/or publication of this article.

Funding

The authors received no financial support for the research, authorship, and/or publication of this article. This work was carried out while the authors were employed by the University of York, the University of Aberystwyth, and Peak Analysis & Automation, UK. No funding for the work was received from any other organization.

Further Reading

- The Commercial Users of Functional Programming annual conference and website (http://cufp.org) hosts tutorials, talks, and Birds of a Feather sessions for practitioners (accessed October 2013).
- The Haskell in Industry website (http://www.haskell.org/haskellwiki/Haskell_in_industry) provides further case studies and support (accessed October 2013).
- Graham Hutton, *Programming in Haskell* (2007), Cambridge University Press.
- Get started with Haskell (installation and tutorial help): http://learnyouahaskell.com/introduction (accessed October 2013).

References

1. Delaney, N.; Echenique, J.; Marx, C. Clarity—An Open-Source Manager for Laboratory Automation. *J. Lab. Autom*. **2013**, *18*, 171–177.
2. Harkness, R.; Crook, M.; Povey, D. Programming Review of Visual Basic.NET for the Laboratory Automation Industry. *J. Lab. Autom*. **2007**, *12*, 25–32.
3. Syme, D.; Granicz, A.; Cisternino, A. *Expert F# 3.0;* Apress, 2012, New York.
4. Shäfer, R. Concepts for Dynamic Scheduling in the Laboratory. *J. Lab. Autom*. **2004**, *9*, 382–397.
5. Harkness, R. Novel Software Solutions for Automating Biochemical Assays. PhD Thesis, University of Surrey, Surrey, UK, 2010.
6. Claessen, K.; Hughes, J. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In *Proceedings of the International Conference on Functional Programming (ICFP)*. ACM: New York, 2000.
7. Runciman, C.; Naylor, M.; Lindblad, F. SmallCheck and Lazy SmallCheck: Automatic Testing for Small Values. In *Proceedings of the Haskell Symposium*. ACM: New York, 2008.
8. Hudak, P.; Hughes, J.; Peyton Jones, S.; Wadler, P. In *A History of Haskell: Being Lazy with Class*, Third ACM SIGPLAN History of Programming Languages Conference (HOPL-III), San Diego, CA, June 9–10, 2007.