Emulation of an unconventional model of computation in Java

Aidan Delaney and Thomas J. Naughton
Department of Computer Science, National University of Ireland, Maynooth

Abstract

This paper describes the emulation of an unconventional model of computation inspired by the field of optical computing. Our development employed a combination of eXtreme Programming, unit and integration testing with junit, and design patterns. In the final product we implemented a novel content-routing message passing system and have realised the first debugger for an optical computer programming language.

1 Introduction

Unconventional models of computation have been studied for many years. They differ to models such as the Turing machine and the λ-calculus, usually in their atomic operations or data representation. Examples include models for quantum computing, biomolecular computing, and for computing over the real numbers. A snapshot of current research can be found in [1]. This paper describes an emulator for an unconventional model inspired by optical physics. In this model all data is stored in infinite resolution images. The model has super-Turing capabilities and so cannot be fully emulated digitally [2]. Therefore, a programming methodology and programming tools were required that permitted necessary compromises to be made, as they presented themselves, during the development of an emulator.

We decided to use the eXtreme Programming (XP) paradigm. Design patterns were employed throughout. Java was chosen for our development as it offered modularity and extensibility. Java complements agile development methodologies (such as XP) by being object-oriented and platform-independent, by having standard support libraries, and by taking care of memory management. Design pattern implementation is well documented in Java (the Java AWT itself implements several patterns). We were also able to take advantage of the XML parsers and DOM implementations available for Java. Unit testing in Java is straightforward using junit and its spinoffs such as XMLunit. We found that it was also possible to configure junit for integration testing. A long-term development goal of ours is to provide a web-based optical computer emulator, and Java's extensibility in this regard is unparalleled.

In Sect. 2 we describe the model of computation. The emulator and novel content-routing message passing subsystem are explained in Sects. 3 and 4, respectively. The first (to our knowledge) debugger for an optical computing programming language is outlined in Sect. 5, and in Sect. 6 we describe unit and integration testing using junit.

2 Model of computation

The continuous-space model of computation is inspired by the theory of Fourier optics. The model was developed for the analysis of (analog) Fourier optical computing architectures and algorithms [3, 4, 5]. The model operates in discrete timesteps over a finite number of two-dimensional (2D) images. It can navigate, copy, and perform other optical operations on its images. A useful analogy would be to describe the model as a random access machine, without conditional branching and with registers that hold 2D functions. Universality, computational complexity gains, and super-Turing properties of the model have been demonstrated [2].

ld $c1$ $c2$ $r1$ $r2$ z_l z_u	: $c1, c2, r1, r2 \in \mathbb{N}$; $z_\mathrm{l}, z_\mathrm{u} \in \mathbb{Q}$; copy into **a** the rectangle of images defined by the coordinates $(c1, r1)$ and $(c2, r2)$. Two additional real-valued parameters $(z_\mathrm{l}, z_\mathrm{u})$ are used to filter the rectangle's contents by amplitude. By default, no filtering is performed, expressed as $(0/1, 1/1)$.
st $c1$ $c2$ $r1$ $r2$ z_l z_u	: copy the image in **a** into the rectangle defined by $(c1, r1)$ and $(c2, r2)$.
h	: perform a horizontal 1-D Fourier transform on the 2-D image in **a**.
v	: perform a vertical 1-D Fourier transform on the 2-D image in **a**.
$\cdot$	: multiply (point by point) the two images in **a** and **b**. Store result in **a**.
$+$	: perform a complex addition of **a** and **b**. Store result in **a**.
$*$	: replace **a** with its complex conjugate.
br $c1$ $r1$	: unconditionally branch to the instruction stored at $(c1, r1)$.
hlt	: halt.

Fig. 1: A summary of the optical computing programming language [2].

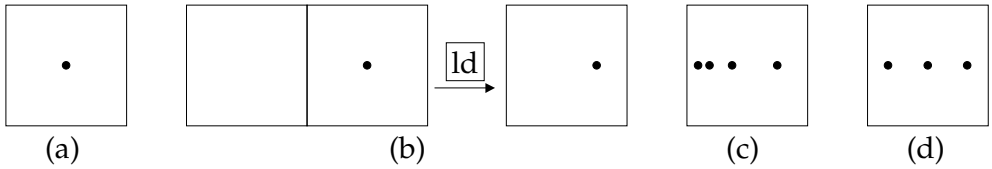

Fig. 2: Encoding numbers in images through the positioning of amplitude peaks. In the illustrations, the nonzero peaks are coloured black and the white areas denote zero amplitude. (a) The symbol '1' is encoded as a single peak in the centre of an image. (b) A 'stack' structure can be used to encode numbers in unary. We 'push' a symbol '1' onto an empty stack (the zero amplitude image) by loading both images simultaneously into **a** and rescaling to the dimensions of a single image. (c) The number 4 encoded in a stack. (d) The number 3 encoded in a 'list' structure.

The memory of the machine consists of a finite 2D grid of images that hold both a program and the input. Each grid element holds a 2D infinite resolution complex-valued image of the form $f : \mathbb{R}_0^1 \times \mathbb{R}_0^1 \mapsto \mathbb{C}$ where $\mathbb{R}_0^1 = \{x : x \in \mathbb{R} \wedge 0 \leq x \leq 1\}$. There is a program start location **sta** and a small number of 'well-known' addresses labeled **a**, **b**, **c**, and so on. The two most basic operations available to the programmer, **ld** and **st** (parameterised by two column addresses and two row addresses), copy rectangular subsets of the grid into and out of image **a**, respectively. Upon such loading and storing the image information is rescaled to the full extent of the target location. The complete set of atomic operations is given in Fig. 1.

There are many ways to encode finite, countable, and uncountable sets as images. We have designed our number encoding scheme around an image that contains a high amplitude at its centre and zero everywhere else. Such an image encodes the symbol '1'. An empty image (zero amplitude everywhere) encodes '0'. Images can be combined using a stepwise rescaling technique (an image 'stack') or with a single rescale operation (an image 'list') to encode nonnegative integers in unary and binary notations. These concepts are illustrated in Fig. 2.

3 Emulation of the model

Emulation of the model of computation is not straightforward: the set of infinite-resolution images is uncountable. We must define an encoding scheme for a countably infinite subset of this set in order to emulate (at least partially) the model discretely. In addition, we must develop algorithms for operations over this subset.

The core of the emulator is the `Processor` subsystem, which handles the memory. Other subsystems are concerned with user input, message passing, processor control, and input verifi-

list	::=	**l** orientation body
stack	::=	**sh** orientation body**:** | **sv** orientation body**:**
body	::=	{complex_number}^positive_int | {complex_number}^*

Fig. 3: A fragment of the regular grammar describing a subset of the infinite-resolution images. The '^*' symbols code for an infinite sequence of peaks of amplitude specified by the preceding complex number.

cation. The user interface `OutputViewer` subsystem prints processor output and errors. The other major subsystem, the `MessageCenter` communication subsystem, is explained in Sect. 4.

3.1 Memory

We decided to design an emulator that is independent of the image encoding. The core of the emulator, the `Processor` subsystem, executes all of the emulation operations. It stores the image grid in a 2D array, and uses an image iterator as an instruction pointer. Each image can itself contain a grid of images. This allows the emulator to load a rectangle of grid images into one image without explicitly needing to process the image data (thereby possibly introducing some loss of information due to rounding error). If an image stored over a rectangle of grid images, it is copied to each image in turn and an appropriate viewbox set. The viewbox concept was borrowed from the scalable vector graphic specification. In practice, no scaling of images actually takes place in our emulator; images are resized by manipulating viewboxes.

3.2 Encoding of images

The restricted set of images we decided to model is based on the encoding scheme in [2] that was sufficient to prove some computability and computational complexity properties of the model (see Fig. 2). A regular grammar (the core of which is shown in Fig. 3) was found to be sufficiently expressive. Each image contains either a word in the regular language or a finite-resolution pixelated bitmap (to also support all conventional raster images).

Extra encoding data was employed to emulate unary operations h, v, and $*$, and the binary operations $\cdot$ and $+$. It was used for the lazy evaluation of unary and binary operations over the encoded continuous images. For example, an image might be prepended with a symbol 'h' to indicate a horizontal Fourier transformation. This system is image representation-independent, be it a raster scheme, vector scheme, or (in our case) a regular grammar. Importantly, lazy evaluation of operations (possibly only when the final result is required by the user) means that defects due to rounding errors, etc., do not propagate through the computation and are only realised at a halt.

3.4 Cancellation calculus

A cancellation calculus was developed to reduce redundant encoding data at each step in the computation, and to implement image operations. This calculus also has the ability to simplify concatenated viewbox representations. The calculus is given in the form of (i) a list of simplifying transformation rules and (ii) algorithms for Fourier transformation, etc. As it is the only component of the emulator that actually physically realises operations, it is the only component specific to an image encoding scheme. Our emulator is therefore easily extensible. The range of infinite-resolution images can be increased by simply extending the grammar (to context-free, for example) and appropriately enhancing the rules and algorithms in the cancellation calculus.

4 Message passing subsystem

We describe the evolution, in terms of a sequence of refinements, of the `MessageCenter` from a Java `Observer-Observable` implementation to a generic content routing architecture with

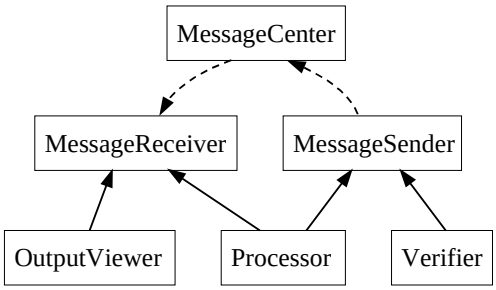

Fig. 4: A diagram of our generic message passing architecture.

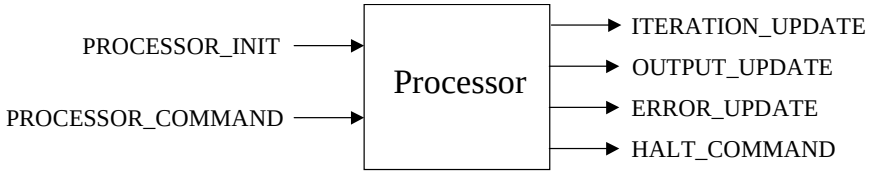

Fig. 5: A diagram of content-types passed into and out of the `Processor`.

a simple design. The observer receives updates with differing content but with the same data-structure. It differentiates updates based on content type rather than data type. We implement an Adapter [6, p. 139] in the observable to wrap the update. A content-type (set by the observable) is provided as extra information by the adapter. Therefore, the observable can send all content packages to the observer, which accepts (or ignores) a packet based on its content type.

As we still use the Java `Observer-Observable` implementation, observable references will have been hard coded into observers. In the next refinement, we replace hard coded references to make the system more modular. We use an implementation of the Observable [6, p. 293] pattern. The Mediator [6, p. 273] component is implemented as a well-known Singleton [6, p. 127]. This mediator forwards update requests from observables to observers. A reference to the mediator can be dynamically obtained by any component, as it is a well-known singleton.

A further refinement allows us to reduce the amount of code in content receivers. All observers receive all updates and have a long sequence of `if()..else()` statements to filter out content types. Observers should elect to receive content they are interested in. The mediator accepts content sender registration requests from an observable and content receiver registration requests from observers. The mediator routes content from an observable to any observer interested in the content.

We also wanted to open our architecture for use as a generic message passing system. The architecture had to remain useful as an observable-observer update brokering mechanism. We created a `MessageSender` abstract class and `MessageReciever` interface. Any classes wishing to send a `Message` must extend `MessageSender` and register the content with the mediator. Any class wishing to receive a `Message` must implement `MessageReciever` and register to receive specific content. The mediator routes `Message`s from a `MessageSender` to possibly several `MessageReciever`s interested in the particular content. The final system is illustrated in Fig. 4.

5 Debugger

Our emulator has a processing core (`Processor`) and a user interface (`OutputViewer`) that communicate via our generic content routing system. The user is given the option of setting a breakpoint in the executing instructions and is informed when a breakpoint has been reached. In this context, our `Processor` and `OutputViewer` are the first two components of a Model-

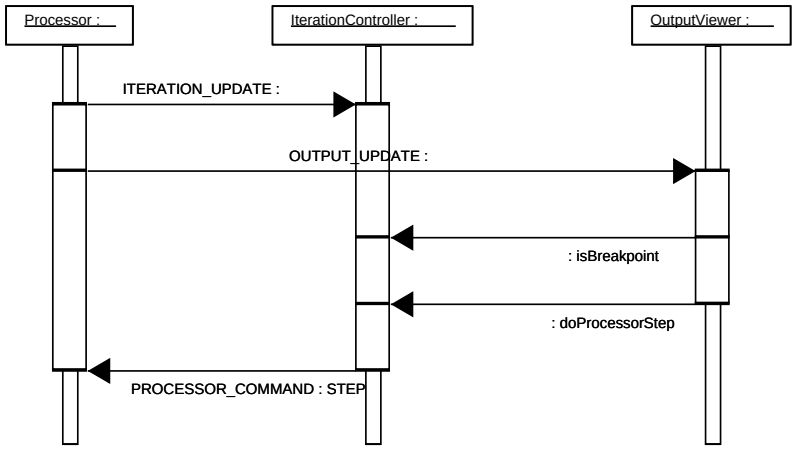

Fig. 6: A UML sequence diagram of our debugger. The role of the `MessageCenter` is hidden for clarity.

View-Control (MVC) pattern. We add a third component `IterationController` to act as a control. The `IterationController` registers to receive updates of the position of `Processor`'s instruction pointer. It also must send commands to the `Processor` instructing it to step and halt. `Processor`'s inputs and outputs are shown in Fig. 5. The timing chart for the MVC communication is shown in Fig. 6, and is explained next.

The view (`OutputViewer`) will have allowed the user to insert a breakpoint (stored in the control). The control (`IterationController`) passes a Command [6, p. 233] to the model (`Processor`) instructing it to execute. After execution of each instruction the model updates the control with the current position of the instruction pointer. If the instruction pointer is at a breakpoint, the control provides this information to the view, otherwise the model is instructed to continue executing. If the view reads breakpoint information from the control, it displays a helpful message to the user, and waits for their response. On receipt of a response, the view instructs the control to command the model to continue execution.

At each breakpoint, the user has the option of stepping through the execution (repeating the sequence of operations in Fig. 6 for each instruction), or of continuous execution. If processing is paused and the view determines that it was not due to a breakpoint (or stepping) the content containing the current state of the image grid is regarded as the output of the computation.

6 Junit

We use junit to automatically run unit tests. Junit is a framework allowing programmers to implement the XP test-first practice with ease. Using junit normally involves subclassing `junit.framework.TestCase`. Numerous unit tests for a specific class may be written in this subclass (referred to as a test fixture).

For each of our classes we created a fixture. Our fixtures contain several tests, which are standard Java methods. The actual test result is compared with the expected result using the `assertTrue` and `assertEquals` methods provided by junit. A simple test on a class involves instantiating it and testing the validity of members available via public `get()` and `set()` methods. We use the reflection "hack" provided by junit to allow easy extensibility of the test repository. The master test class contains only the name of the class containing test fixtures. Thus no changes are required to other test classes if a new test is added to a fixture.

6.1 Integration testing with junit

We identified a method for utilising junit for automating integration testing. The method was first applied to the message passing subsystem of `MessageSender`, `MessageCenter`, and

`MessageReciever` (see Fig. 4), and secondly to `Processor`. These two subsystems were targeted as integration testing subjects because they (i) are two subsystems that are composed of several other classes, (ii) have a high dependency on classes within their subsystem (high cohesion), and (iii) have a low dependency on classes outside their subsystem (low coupling). By implementing internal classes in a junit fixture, we could subclass `MessageSender` and `MessageReciever`. The `MessageSender` could then pose as a `Verifier`, `Processor`, or `IterationController`. The `MessageReciever` could pose as a `Processor`, `OutputViewer`, or `IterationController`. In this fashion, message passing between modules could be tested. By implementing integration testing within the unit test framework, we could automate the whole verification testing process.

7 Conclusion

We have successfully emulated the partial functionality of a super-Turing unconventional model of computation. This required defining an encoding scheme for a subset of its infinite-resolution images. Java aided our agile development as it allowed us to group many classes with different functions together using an interface. This is seen in classes implementing the `MessageReciever` interface. It would not have made sense to use `MessageReciever` as an abstract class as receivers receive differing content types and must process these types differently. Design patterns, in addition to giving us trusted methods of solving common problems, allowed us to provide clear and concise explanations (for example, for the evolution of our message passing architecture).

Building an automated unit test repository was allowed us to take full advantage of the XP test-first practice. Our integration testing is effected using the same technique and so both unit and integration tests could become part of any future regression test suite. Future work on this project may include enhancing the image encoding scheme. We have shown that only the cancellation calculus must be updated to increase the range of acceptable infinite-resolution images as our emulator is a general emulation framework. The software end-product is currently being used to design algorithms for implementation on Fourier optical computers.

The authors gratefully acknowledge the assistance of Damien Woods, and the Theoretical Aspects of Software Systems (TASS) research group, NUI Maynooth. Thanks are also extended to the reviewers of this paper for their constructive comments.

References

[1] I. Antoniou, C. S. Calude, and M. J. Dinneen, eds., *2nd International Conference on Unconventional Models of Computation, Brussels, Belgium, Dec. 2000*, Springer, London, 2001.

[2] T. J. Naughton and D. Woods, "On the computational power of a continuous-space optical model of computation," *3rd International Conference on Machines, Computations, and Universality, Chişinău, Moldova, May 2001*, LNCS **2055**, 288–299, Springer, London, 2001.

[3] A. VanderLugt, "Signal detection by complex spatial filtering," *IEEE Transactions on Information Theory* **IT-10**, 139–145, 1964.

[4] C. S. Weaver and J. W. Goodman, "A technique for optically convolving two functions," *Applied Optics* **5**(7), 1248–1249, 1966.

[5] T. Naughton, Z. Javadpour, J. Keating, M. Klíma, and J. Rott, "General-purpose acousto-optic connectionist processor," *Optical Engineering* **38**(7), 1170–1177, 1999.

[6] E. Gamma, R. Helm, R. Johnson, and J. Vissides, *Design Patterns, Elements of Reusable Object-Oriented Software,* Addison-Wesley, NJ, 1995.