6-2000

Aspects of Information Flow

Andrew P. Black
Portland State University, black@cs.pdx.edu

Jonathan Walpole
Portland State University

Citation Details

"Aspects of Information Flow", Andrew Black and Jonathan Walpole, In Proceedings ECOOP Workshop on Aspect-Oriented Programming, June 2000.

Aspects of Information Flow

Andrew P. Black
black@cse.ogi.edu

Jonathan Walpole
walpole@cse.ogi.edu

Department of Computer Science & Engineering
Oregon Graduate Institute of Science & Technology
Beaverton, Oregon, USA

A Position Paper submitted to the ECOOP'2000 Workshop on Aspect-Oriented Programming

Introduction

Along with our colleagues at the Oregon Graduate Institute and Georgia Institute of Technology, we have recently been experimenting with real-rate systems, that is, systems that are required to move data from one place to another at defined rates, such as 30 items per second. Audio conferencing or streaming video systems are typical: they are required to deliver video or audio frames from a source (a server or file system) in one place to a sink (a display or a sound generator) in another; the frames must arrive periodically, with constrained latency and jitter.

We have successfully built such systems (for example, see reference [Walpole 1997]), but they are not simple to design or construct. Our current research seeks to capture our knowledge of this domain into an information flow framework, called *InfoPipes*. The goal of Infopipes is to make the task of building a system that moves data from one part of the Internet to another as simple as connecting pre-defined components such as buffers, pipes, filters and meters. The latency, jitter and data-rate properties of the resulting pipeline should follow by calculation from the properties of the components.

As we try to understand how Infopipes should be constructed, what interfaces they should offer, and what properties they should have, some decisions seem to be easy to make. The components can be modelled as objects: we have objects representing bounded and unbounded buffers, straight pipes, sources, and sinks. Every component has zero or more inputs and zero or more outputs; straight pipes have one input and one output, sinks are defined as having at least one input but no outputs, and conversely sources have no inputs but at least one output. Various kinds of Y connectors have different numbers of inputs and outputs. In a properly constructed pipeline, every input must be connected to the output of some other component, and vice versa.

Other decisions seem harder, because several alternatives appear equally compelling. One such decision is the activity model: should data be pulled through a pipeline, or pushed? Another is the blocking behaviour of a component: what happens if I try to

pull an item from an empty pipe? Does my attempt block until an item is available, return with a “bubble of air”, or raise some sort of exception?

We feel that it is desirable for such properties to be captured as aspects. One reason for this is that we will probably want to make multiple variants available: pipes that push and pipes that pull, pipes that block and pipes that return air. This is because we are using InfoPipes to model existing components in real operating systems and networks, and these components actually exhibit all of these characteristics. Thus, we expect to need variability in the activity model and blocking behaviour; it seems better to obtain this by combining a base component with an appropriate selection of aspects, than to have to build many versions of the components from scratch.

So, let us look a little more closely at activity and blocking behaviour, to see if it becomes clear how to represent them as aspects.

Activity in InfoPipes

THE PULL MODEL

Imagine an InfoPipe component *P* delivering data to another component *Q*. There are two ways in which they can communicate. In the “Pull model”, illustrated in Figure 1, *Q* sends a pull message to *P*, which (if *Q* is fortunate) replies with an item.

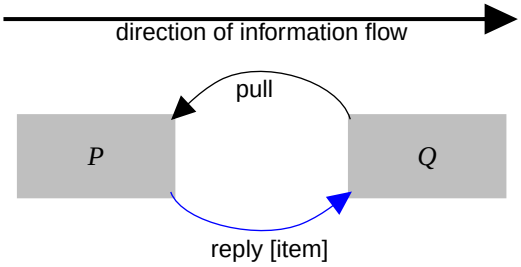

Figure 1. The Pull model of data flow.

THE PUSH MODEL

In contrast, Figure 2 shows the “Push” model of data flow. Here, *P* sends a push message to *Q*, accompanied by the data item. If all is well, *Q* accepts the item and replies with a positive acknowledgement. In both cases, the data flow is the same, but the components have different activity. In the terminology of reference [Black 1983],

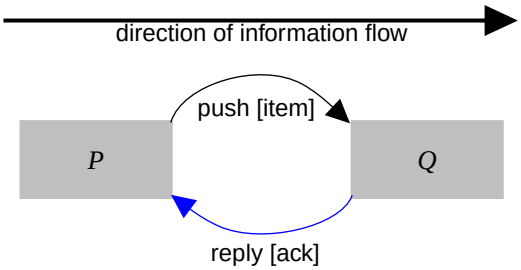

Figure 2. The Push model of data flow.

in Figure 1 *Q* is doing active input, while *P* is doing passive output, while in Figure 2, *Q* is doing passive input, while *P* is doing active output.

Clearly, data flow can be achieved either by making all of the Infopipe components understand pull, or by making them all understand push. But we will maximize the opportunities for constructing pipelines with interesting activity models if every component (other than sources and sinks) understands both pull and push.

PASSIVE COMPONENTS

In addition, both of the figures imply that either *Q* (in Figure 1) or *P* (in Figure 2) contains a source of "motive power", or at least some code that when "ignited" is capable of sending the pull or the push message to the other component. An alternative configuration is possible in which this is not so; see Figure 3. Here, all of the

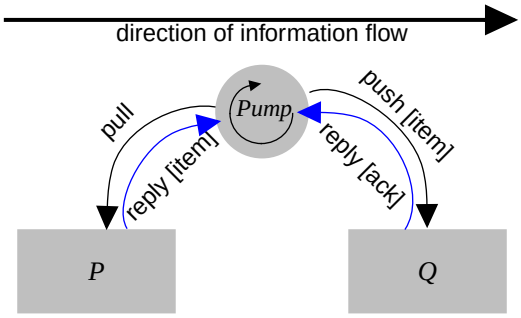

Figure 3. An external *Pump* extracts data from *P* and pushes it into *Q*

activity is contained in a third component, called *Pump*. Both *P* and *Q* wait passively for *Pump* to send them messages; at this point (but not before) they respond.

ACTIVE COMPONENTS

By symmetry, one might expect a fourth configuration; this is shown in Figure 4. Here, both *P* and *Q* are active. *P* pushes items into *Buffer*, and *Q* pulls items out of *Buffer*. If an active component wishes to pass data to another active component, it is essential that there be a buffer (with a capacity of at least one item) between them. In Unix, the *Buffer* of Figure 4 is called a Pipe; when *P* and *Q* are processes, this is the only activity model that is supported in Unix.

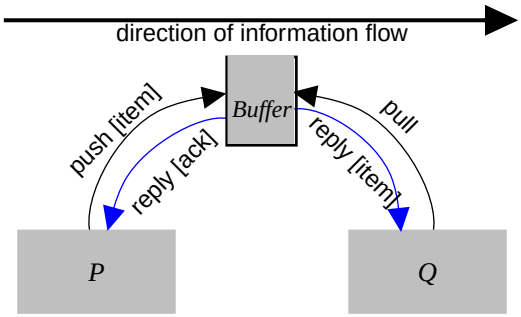

Figure 4. An external *Buffer* receives data from *P* and supplies it to *Q*

ACTIVITY AS AN ASPECT

In an InfoPipe system, *P* and *Q* might be various kinds of devices, or network or operating system components. We no not want to have to provide multiple versions of each, depending on the desired activity model. Instead, we would like activity to be an aspect, which could be "added" to any object where it makes sense. It seems reasonable to define one set of InfoPipe components that are always passive, and then selectively add activity to them.

The alternative would be to make all components active, but to selectively "turn off" activity when it is not wanted. This is probably easier to program, but it has the major disadvantage that the complexity of a active process is already present in every component, whether or not it is required. Building active components from the composition of activity and components seems to be pedagogically much more desirable.

But *how* passive should passive components really be? Clearly, a passive component *P* understand pull and push[item]. But what should *P* do when an item is pushed into it by an InfoPipe *Q*? If *P* has no storage, there are three options.

1. *P* could invoke push[item] on the InfoPipe *O* connected its output. This is probably the simplest option. But from the point of view of *O*, when *P* does this it is behaving like an *active* Infopipe.
2. *P* could block *Q*, (more precisely, block the thread that is invoking push), until such time as someone invokes pull on *P*. This is certainly passive. But it presupposes that *P* is allowed to exhibit blocking behaviour. Whether this should be permitted is the subject of the next section, but we would in fact like to make an InfoPipe's blocking behaviour a separately selectable aspect.
3. The final option is as passive as we can imagine: *P* does *nothing at all!* In other words, it ignores the item that was pushed into it, effectively dropping it "onto the floor". A possible variation is that if a pull invocation arrives from *O* simultaneously with the push invocation from *Q*, then the item is passed to *O*, otherwise it is dropped.

Note that the choices that we make here affect two other aspects: reliability and blocking behaviour. We might have hoped that these aspects would be orthogonal to the activity model, but that does not seem to be the case,

Our first attempt to selectively add activity to an InfoPipe component was by introducing two new methods, startPumping[speed] and stopPumping, into the component. But the implementation of these methods depends on the nature of the component. For example, pumping a straight pipe pulls from its input pipe and pushes onto its output pipe, whereas pumping a source simply pushes items onto its output. To provide a uniform "socket" to which a pump could be attached, we added the method stroke to every component: it captured the behaviour of the component corresponding to a single "stroke" of the pump. startPumping[speed] was then implemented by attaching to the component a process that stroked at the appropriate speed.

Subsequently, we realized that the same behaviour could be achieved by object composition, without the use of aspects at all. Look again at Figure 3 on page 3; if we focus on the relationship between *P* and *Pump*, we see that their *composition* is a compound InfoPipe that has exactly the properties of an active version of *P*; this is shown by the grey box in Figure 5. So, ordinary object composition through the push and pull interface seems to be adequate to express what we though was an aspect! This is a little disconcerting.

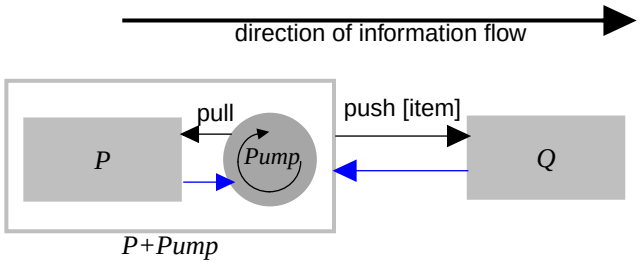

Figure 5. *P* and *Pump* can be composed to create an active version of *P*

The one drawback with the compositional approach is that in most OO languages, the composition of two objects is not itself an object: “P+Pump” has no object identity and cannot be sent messages.

The open questions here are:

1. Can activity be represented as an Aspect?
2. What are the advantages and disadvantages of doing so, compared to representing activity as a component that can be connected to other components (as in Figure 5) to make a composite component?

Blocking Behaviour

As with activity, we can imagine a family of infopipe components that offer the same basic behaviour but which differ only in their blocking characteristics.

Suppose that *P* is an Infopipe whose storage is full, or which has no storage. What should *P* do when an item is pushed into it? Once again, there seem to be several options:

1. Silently drop the item “on the floor”. This is the simplest behaviour, and is the behaviour actually exhibited by real-world networking components such as routers.
2. Apologetically drop the item, that is, return a status code or exception to the caller indicating that the proffered item has, regretfully, been refused.
3. If *P* has storage, for example, *P* is a buffer, then a wider range of dropping behaviours are possible. *P* might drop the oldest or least urgent item already in its buffer storage, in order to make room for the newly arrived item.
4. Block the thread that invoked push, until such time as another thread makes enough space in *P* to accommodate the new item.

A similar range of options are available if an Infopipe attempts to pull an item from an empty Infopipe, *i.e.*, one that has no storage or whose storage is empty.

1. The simplest action is to introduce a “bubble of air” into the pipeline, and to pass it to the invoker. (In our prototype implementation, we use nil for this purpose.)
2. A more polite (but sometimes less convenient) alternative is to return a status code or exception to the invoker indicating that, regretfully, no item is available.

3. It is sometimes appropriate to fabricate a new item and return it. For example, in an audio pipeline, it might be appropriate to fabricate a packet of white noise; in a video pipeline, it is common to re-transmit the previous frame.
4. The final option is to block the thread that invoked pull, until such time as another thread makes an item available.

The problems that must be solved in order to add blocking as an aspect are similar to those involved with synchronization [Matsuoka 1993]. First, we must choose some base behaviour for InfoPipes in the absence of blocking. Pragmatically, this would seem to be one of the first two options described above. It it easy to add an aspect that intercepts the bubble of air or the exception and blocks the calling thread. It is less clear how to ensure that this thread is unblocked *whenever* a new item becomes available. For example, as new methods are added to subclasses of the basic component, the number of ways in which a new item could arrive in the buffer might increase. How can we guaranteed to unblock the waiting thread in all of the right situations? One possibility is to check a predicate (callerBlocked **and** buffer nonempty) after the execution of *every* method on the Infopipe.

Time as an Aspect

In the introduction we mentioned that Infopipes are intended as an abstraction for real-rate systems, which guarantee to move data from one place to another with certain timing properties. An intriguing question is whether time itself can be treated as an aspect. That is, can we write “timeless” Infopipe components, and then add various sorts of timing properties as aspects?

Our initial opinion is that this is not in fact possible, at least not so long as we are thinking in terms of an aspect weaver working with a conventional base programming language. The reason is simple: all the conventional programming languages developed in the second half of the 20th Century are timeless. Like all high-level abstractions, programming languages choose to hide some details of the machines on which they are implemented, and emphasize others. Without exception, they all choose to hide the notion of time!

Timelessness is an empowering choice: without abstracting away from time it would, for example, be impossible to prove that two program fragments are equivalent. This is because, even if the fragments can be proved to effect the same transformation on the state, they are very unlikely to have the same execution time. It would also be impossible to implement the same language on more than one hardware base. But timelessness is also a limiting choice: if a language is essentially timeless, how can it be used to build a system in which timeliness is a critical property? The answer has been to depend on a Real-Time Operating System to provide timing-related operations.

Of course, there are some “unconventional” programming languages, notably Esterel [Berry 1998] and Lustre [Caspi 1987], which are specifically intended for programming real-time systems. However, even these languages do not include a conventional notion of time. Instead, they depend on the so-called “Synchrony Hypothesis”, which states that computing activity takes *no time at all*, that is, computation is instantaneous. Neither do synchronous programming languages like Esterel have a notion of time built into them. Instead, they rely on a sequence of

periodic events from an external sensor. From these the program must synthesize some kind of clock.

The Synchrony Hypothesis allows real-time programmers to build systems that perform adequately, because the time taken to perform simple sensor computations, although not actually zero, is much smaller than the time between the external events to which the system must react. However, this way of dealing with time, or, rather, of *avoiding* dealing with time, is clearly not compositional. Even if the computation associated with one reaction is negligible, can we say the same of 10, 100 or 1000 similar computations? Eventually, the time that these computations take must become similar to the reaction times that the system is required to exhibit.

For these reasons, another branch of our research is directed towards designing a new programming language in which time is a central property. We came to this position reluctantly; if a way of adding a time aspect to a conventional language can be found, we would eagerly adopt it.

TIMING PROPERTIES FOR INFOPIPES

What are the timing properties that we need to express for real-rate Infopipes? Here are some examples that we have considered.

1. Interfacing to real-rate devices, such as audio or video capture devices, that push items into a pipeline at a determined rate, or audio or video output devices, that need to be supplied with items at a determined rate. Processing, such as compression, must be integrated into the pipeline.
2. Add to the above the complexity of multiple active components. So long as only one of the active components is real-rate (*e.g.*, has its own real-time clock), it is a relatively simple matter to "phase lock" all of the components, active and passive, to that real-rate. The active components may be thought of as "virtual-rate"; their time base can be stretched or compressed to correspond to the needs of the real-rate component. Passive components are executed by the active ones in lock-step.
3. However, if the pipeline contains multiple real-rate active components, *e.g.*, a real-rate source and a real-rate sink, something has to give! The rates will *not* be exactly matched. Even two crystal clocks with the same nominal rate will be found to diverge over time. Some data dropping or synthesis will be necessary.
4. Add between the source and sink above an active network, whose latency and throughput vary over a wide range.
5. Now add some data processing steps, such as MPEG encoding, decoding or transcoding, that most definitely take more than negligible time.

Orthogonality of Aspects

Naively, we had hoped that activity, blocking behaviour and time, when treated as aspects, would be orthogonal. This would guarantee that they could be added in many combinations. However, it seems that this is not the case. We have already seen that it is hard to talk about the semantics of activity without also taking about when that activity is blocked. Similarly, a component that must satisfy timing constraints needs to understand the blocking characteristics of all of the methods that it invokes. If it does not, there is a risk that a thread will be blocked and will be unable to carry out a time-critical task. And the implementation of activity (by a process scheduler) is responsible for the ability of a component to offer timing guarantees.

Indeed, we might argue that aspect-oriented Programming becomes an interesting and powerful tool exactly when the aspects are *not* orthogonal, but interfere constructively. How can we promote such constructive interference, and avoid destructive interference?

Summary

We have briefly described an interesting application domain, Infopipes, in which aspects would seem to provide an appropriate way of limiting code complexity, reducing duplication, and improving maintainability. But it is not clear to us, yet, how to apply them. It is also unclear if timing properties can be "added" to timeless objects as aspects. The idea is attractive, but the problems seem daunting.

We believe that these problems are worthy of discussion at the ECOOP'2000 Workshop on Aspect-Oriented Programming.

Acknowledgements

We would like to acknowledge Calton Pu's rôle as the co-developer of the Infopipe concept, and as a valued continuing collaborator.

References

[Berry 1998] Berry, G. The Foundations of Esterel. In *Proof, Language and Interaction: Essays in Honour of Robin Milner.* G. Plotkin, C. Stirling and M. Tofte. MIT Press, 1998.

[Black 1983] Black, A. P. An Asymmetric Stream Communication System. Proceedings 9th SOSP. ACM, 1983

[Caspi 1987] Caspi, P., D. Pilaud, et al. (1987). LUSTRE: a declarative language for programming synchronous systems. Conference Record of the Fourteenth ACM Symposium on Principles of Programming Languages, Munich, West Germany, ACM Press, 1987.

[Matsuoka 1993] Satoshi Matsuoka and Akinori Yonezawa. Analysis of inheritance anomaly in object-oriented concurrent programming languages. In Gul Agha, Peter Wegner, and Akinori Yonezawa, editors, Research Directions in Concurrent Object-Oriented Programming, pages 107–150. MIT Press, 1993.

[Walpole 1997]Jonathan Walpole, Rainer Koster, Shanwei Cen, Crispin Cowan, David Maier, Dylan McNamee, Calton Pu, David Steere and Liujin Yu. A Player for Adaptive MPEG Video Streaming Over The Internet. Proceedings 26th Applied Imagery Pattern Recognition Workshop AIPR-97, SPIE, Washington DC, October 15-17, 1997. http://www.cse.ogi.edu/DISC/projects/quasar/papers/aipr.ps.