Technical Note: SciDAC-SPA-TN-2003-01

On Providing Declarative Design and Programming Constructs for Scientific Workflows based on Process Networks

Bertram Ludäscher[*] Ilkay Altintas[†]

San Diego Supercomputer Center, UC San Diego

August 2003

Abstract

In this technical note we first describe in some detail the structure of the Promoter-Identification-Workflow (PIW) demonstrated at SSDBM [ABB+03]. We then point out some serious shortcomings of the current prototype. At the root of these problems lies a lack of capabilities for programming with collections (such as lists). We then propose a simple solution to this problem, based on a functional programming approach. The use of, *e.g.*, Haskell as an underlying formal model has a number of advantages including clear semantics, compatibility with process networks, and powerful support for typing of workflows. We now have a host of workflow design and programming constructs we would like to add to Ptolemy-II to make it a true SPA tool.

Note that this is early work in progress (despite the typesetting in LaTeX ;-)

Contents

[*]Author's email: `ludaesch@sdsc.edu`
[†]Contributing author

1 Introduction

Consider the scientific workflow in Figure 1. It depicts the implementation in Ptolemy-II [PTO03] of the *promoter identification workflow* (PIW), demonstrated at SSDBM'03 [ABB$^+$03] for the SPA team. Roughly speaking, PIW aims at constructing models of transcription factor binding sites to identify co-regulated genes, starting from some microarray data. Using cluster analysis (*e.g.*, Clusfavor), a set of gene-ids is produced which are fed, one at a time, into the PIW in Figure 1.

Let us try to understand this figure: The uppermost box shows the top-level workflow. The small green box labeled "PN Director" indicates that we use Ptolemy's ***P**rocess **N**etwork Domain* to execute this workflow. The PN director is a software component that orchestrates the workflow execution according to the PN domain [DHK$^+$01]. In a process network [KM77] actors (depicted as boxes) represent processes that communicate through unidirectional channels having FIFO queues as buffers. An actor trying to read from an empty queue will get blocked until sufficient input is available, while writing to a queue always succeeds, *i.e.*, the queues are (in theory) infinite.[1] PN is a natural model for describing systems where (potentially infinite) streams of data samples, called *tokens*, are incrementally transformed by a number of processes executing in parallel [DHK$^+$01].

In the somewhat simplified PIW in Figure 1 we assume that a single gene-id is given as input (see the box labeled StringConst). The actor with the label Gene Sequence Processing is depicted using an icon which indicates that this is a *composite actor* which has a *sub-workflow* nested inside of it.[2]

Looking at the second box from the top, we see that its input is Access Number, *i.e.*, a gene-id, which is routed to a GenBank actor. The latter is implemented as a web service which looks up the corresponding Gene Sequence at NCBI's GenBank [NCB02]. The gene sequence is then fed to the BLAST actor whose output is a list of promoter-ids,[3] which are then fed to a subsequent composite actor labeled Inner Loop. The fact that BLAST produces as ***list*** of promoter-ids from a single gene-id, creates some (avoidable as we shall see) trouble for us. The actors encountered before can be considered as consuming one input data token (*e.g.*, a gene-id) and producing one output data token (*e.g.*, its gene sequence).[4] The BLAST actor, on the other hand, does not have this nice property and "shifts gears" by injecting a sequence of output data tokens (the list of promoter-id) for a single individual input data token (the gene sequence).

A lot of the complexity of the Inner Loop composite actor, whose expansion is shown in the bottom left box in Figure 1, stems from managing the control flow to "handle the situation": The EnumHomolog actor on the left takes as input the BLAST result as a single data token containing a list of promoter-ids. It then breaks up this token and outputs one promoter-id token at a time to the subsequent steps. For example, the GISequencePromoterRegion actor calls an NCBI web service to produce a single *promoter sequence* from the input promoter-id, then extracts from the sequence some *promoter region* which is then fed to the Transfac actor for analysis of transcription factor binding sites. For each data token (a promoter region), Transfac outputs a list of *transcription factor binding sites* (TFBS). It also **signals back** to EnumHomolog when it is done with processing its input promoter region.

Why do we need this backward control arc?

[1]Ptolemy's SDF domain corresponds to a special variant of process networks called ***S**ynchronous **D**ata**F**low* networks. When an actor *fires* in an SDF network, it consumes a fixed number of tokens and produces a fixed number of tokens. *Deadlock*, *boundedness* (of queue sizes), and *execution order* of actors can be statically analyzed [LP95]. While these properties make SDF a very desirable domain, for PIW we had to rely on the PN domain, since some actors produce a varying (and unbounded) number of tokens.

[2]When people, including ourselves, speak of (scientific) *workflows*, they often refer to *dataflow* oriented process networks. The differences will be discussed in a separate technical note. For now we go with the flow and somewhat sloppily stick to the term "workflow".

[3]a *promoter* is a subsequence of a chromosome that sits close to a gene and regulates its activity

[4]This makes them perfect candidates for the SDF domain.

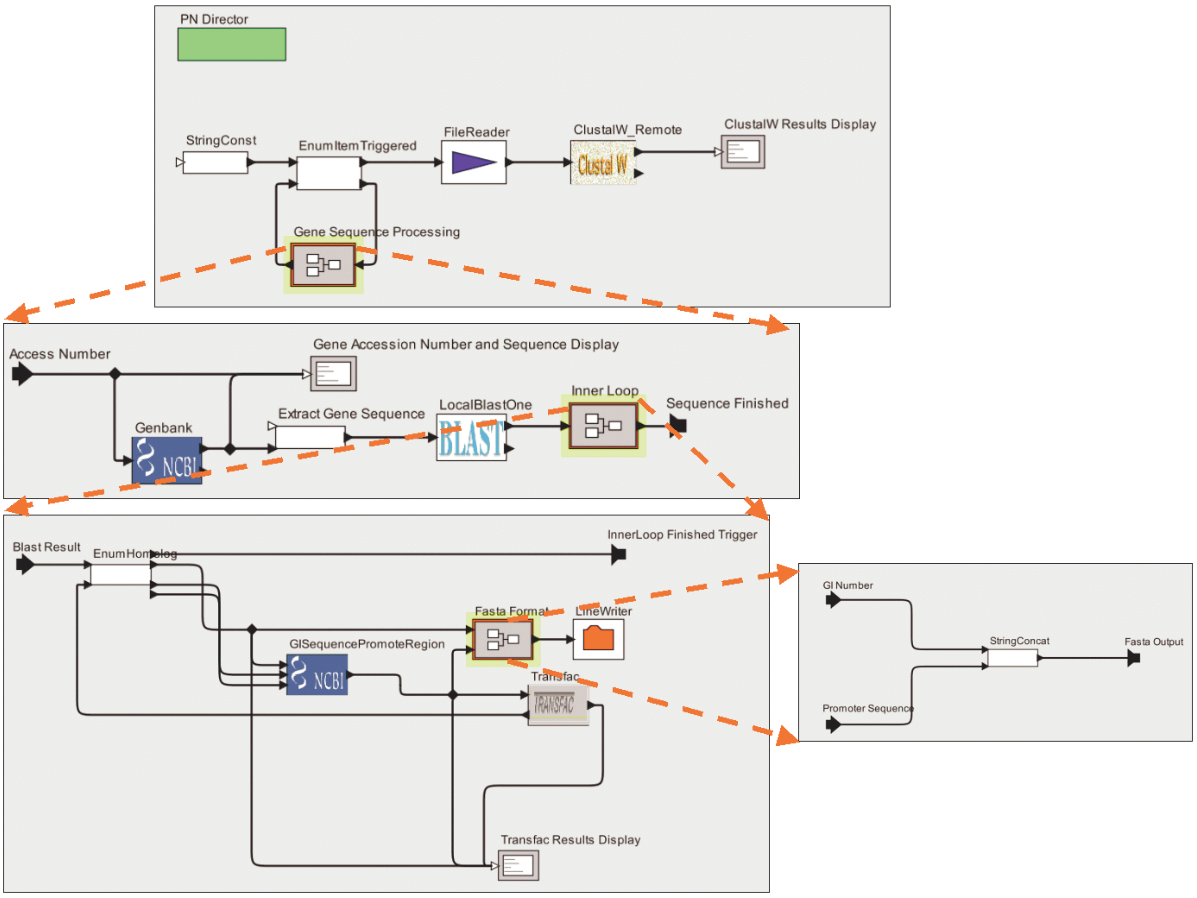

Figure 1. Promoter Identification Workflow (PIW) in Ptolemy-II (SSDBM'03 version [ABB+03])

Recall that EnumHomolog enumerates a list of promoter-ids $[p_1, \ldots, p_\ell]$, coming from a single gene sequence g. Therefore, if we consider a sequence $g_1, g_2, \ldots$ of such gene sequences, then to each g_i corresponds a list of promoter-ids $[p^i_1, \ldots, p^i_{\ell(i)}]$. The backward control arc is used to maintain the "packaging" of these promoter-id lists: Whenever Transfac is done with some promoter-region r^i_j (corresponding to some promoter-id p^i_j) it signals this fact back to EnumHomolog. Since EnumHomolog was emitting the p^i_j's corresponding to g_i in the first place, it can keep track of when a gene-id's processing tasks are completed, *e.g.*, by keeping a counter which is initialized to the number of promoter-ids and decremented on each "call home" received from Transfac. When the counter reaches zero, the EnumHomolog actor can signal this to its parent workflow (via the InnerLoop Finished Trigger), which signals it further up to the EnumItem Triggered actor in the top-level workflow. The latter can then notify the subsequent FileReader actor to pick up the complete results assembled for the gene-id and pass them on to the ClustalW actor. The EnumItem Triggered actor itself is now ready to receive a new gene-id as input and pass it on to the Gene Sequence Processing composite actor and the same procedure is started all over.

Problems. The described implementation of PIW was a useful exercise and proof of concept, showing that we can fairly easily extend Ptolemy-II in various ways, in particular to run scientific workflows over a number of web services. It also made clear that this "ad-hoc plumbing" of custom actors leads to a number of problems that we need to address:

1. First and foremost, the described way of iterating over data collections[5] renders the control flow ***overly and unnecessarily complex***. For example, the backward control flow Transfac⇝EnumHomolog⇝EnumItemTriggered breaks the simple and intuitive forward dataflow of things, and makes it very hard to comprehend the PIW execution.

2. Another unpleasant consequence is that we have essentially ***sequentialized execution*** to one gene-id at a time where we could have benefitted from the parallel execution capabilities of process networks. For example, we should be able to pipeline new items into the Gene Sequence Processing sub-workflow at any time and should not have to wait for Transfac's "synchronization signal".

3. Another serious shortcoming is that our actors are ***designed to fit***. For example, the Transfac actor is designed to accept input in exactly the form emitted by the preceding GISequencePromoterRegion actor, and it sends back control information meant specifically for EnumHomolog.

4. There are a number of other issues that will need our attention, for example:

    - How can we support the design (and static analysis!?) of workflows even if not all necessary actors are implemented yet?[6]
    - How can we support highly user-interactive workflows?
    - How can we deploy workflows?
    - How can we leverage existing ontologies for scientific workflows?
    - How can we best support scientific data formats such as netCDF?
    - How can we interoperate with other SciDAC and IDMAF relevant tools (ASPECT, LIMS, etc.) and/or Grid tools (*e.g.*, OGSA(-DAI) based)?

Clearly, there are quite a few issues we have to overcome before domain scientists can design workflows such as PIW themselves. In the remainder of this technical note we will focus on a solution for (1). As it turns out, this will also take care of (2). We are also working on (3) by developing a ***generic data transformation actor*** which will be able to map outputs of one step to inputs of the subsequent steps.[7]

2 Control-Flow vs. Dataflow

Ptolemy-II's PN (process network) domain and its subdomain SDF (synchronous dataflow) provide an intuitive framework with a precise execution semantics and "free parallelization" for data-intensive processing pipelines. At the core of PN is the idea of unidirectional buffered communication with non-blocking writes and blocking reads (*i.e.*, when no input tokens are available). This is akin to how Unix pipelines work. For example,

```
cat foo | pr1 | pr2 | sort
```

sends a file `foo` to some process `pr1` which sends the transformed output `pr1(foo)` to the process `pr2`, which after some further processing sends its output `pr2(pr1(foo))` to `sort`. The processes in the pipeline execute independently and in parallel, synchronized only in way similar to PN: a process is blocked until it has sufficient input. In Unix the "token unit" is a line of characters (terminated by an end-of-line character).

In dataflow languages such as Unix pipelines and PN, there are some means of exerting *flow control*. For example, Ptolemy-II has a number of flow control actors [DHK+01, Sec. 3.3.4] such as

$$\texttt{switch} : \mathit{input}, \mathit{control} \rightarrow \mathit{output}_1, \ldots, \mathit{output}_k$$

[5]Here "collection" is used in the programming language sense (*i.e.*, programming with lists, bags, and sets), not in the digital library sense for example.

[6]The SEEK project is working on a "design extension" for this purpose [SEE03].

[7]This is the topic of a separate technical note. The generic transformation actor can be instantiated to execute a given XQuery, XSLT script, or Perl script. This could be also the place were some (semi-)automatic wrapping technology such as XWrap could be useful to apply.

which routes its *input* to the port *output*$_i$ selected by the *control* input. By having the value of the control input computed as a function of the input data, one obtains the equivalent of a switch or case statement know from imperative programming languages. Such branching often happens as a result of arithmetic comparisons ($<, =, \ldots$), and Ptolemy-II has indeed actors for these comparisons and other logic operations (`and`, `or`, `not`, ...) [DHK+01, Sec. 3.3.5].

Despite these possibilities, "divorcing" the control-flow from the natural dataflow afforded by process networks requires some effort, and the results are not always satisfying, as explained above for the PIW in Figure 1. Even if we could avoid some of the specialized trigger constructs in PIW and use Ptolemy's flow control actors, the result would most likely still be unnecessarily complex.

At the heart of the specific PIW problem lies a *granularity mismatch* between some actors/web services which produce lists of data elements of a certain type, while other actors/web services can only operate on those one at a time. Fortunately, there is a simple and elegant solution based on standard declarative functional programming constructs.

3 Declarative Collection Handling by Functional Programming

Consider an actor f that can accept an input data token x of type α and produces a corresponding output data token y of type β as a function of x (*i.e.*, $y = f(x)$). We can view f as a function

$$\begin{aligned} f : \alpha &\rightarrow \beta \\ x &\mapsto f(x) \end{aligned}$$

If we apply (or "*fire*") f repeatedly on a sequence $x_1, x_2, \ldots$ of inputs, we obtain a sequence of outputs $f(x_1), f(x_2), \ldots$ In Ptolemy-II terminology [LP95], we say that we have obtained the *functional process* $F : [\alpha] \rightarrow [\beta]$ (mapping sequences of tokens of type α to sequences of tokens of type β) from the actor function $f : \alpha \rightarrow \beta$. Formally we have that

$$F(\mathbf{x}) := \mathsf{map}(f)(\mathbf{x})$$

where $\mathbf{x} = [x_1, x_2, \ldots]$ is a sequence (list) of inputs, and where map is the well-known higher-order function that takes as input a function $f : \alpha \rightarrow \beta$ and a list of type $[\alpha]$, and successively applies f to each list element, resulting in a list of type $[\beta]$:

$$\begin{aligned} \mathsf{map} : \ (\alpha \rightarrow \beta) \rightarrow [\alpha] &\rightarrow [\beta] \\ f \ [x_1, \ldots, x_n] &\mapsto [f(x_1), \ldots, f(x_n)] \end{aligned}$$

Here map is used to explain how to obtain a process over sequences from a function over tokens. As it turns out, map can also be applied in a similar but different way to solve our "granularity mismatch problem" from Section 1.

More generally, the functional programming model, in particular with *lazy semantics*[8] affords a number of modeling constructs that can be nicely blended with process networks. Specifically, collection programming, *e.g.*, iterating over lists can be achieved in a way that is compatible with process networks and which avoids the problematic explicit use of loops.

3.1 PIW in Haskell

We will now reformulate the Promoter Identification Workflow in a functional style using Haskell syntax [Has]. Note that we use Haskell here as a *specification* mechanism not as a means to actually execute workflows. The latter is achieved by mapping the specification to a Ptolemy-II implementation; we will come back to that in the next section. For now, let us preview what benefits we will get by employing a functional specification:

1. The most important advantage is that we get a much simpler and intuitive overall model of PIW.

2. In particular, the "spaghetti control structure" is avoided and we have a simple forward dataflow (much like Unix pipelines) without backward directed control channels.

3. The resulting model affords "free parallelization" instead of sequential execution.

[8]Under *lazy semantics*, a function defers evaluation of its arguments until their values are needed. Haskell is a prominent example [Has].

4. The declarative nature of the model allows optimization techniques based on function rewriting.

5. The underlying powerful type system based on parametric polymorphism can catch "pipeline design errors" early on.

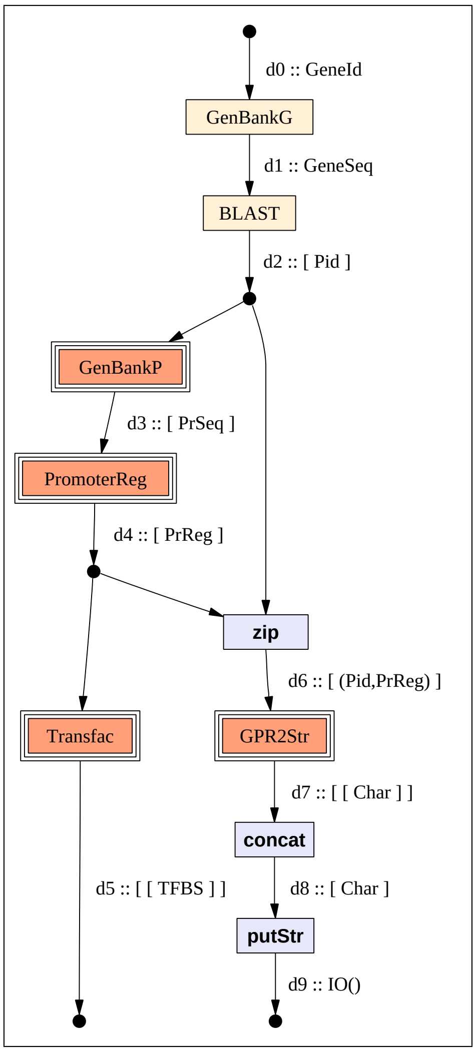

Figure 2. Process network for PIW; boxes with double outline are wrapped by the map construct; built-in functions are **boldfaced**

Consider the PIW process network in Figure 2 (for layout reasons, drawn top-down of left-to-right): The GenbankG actor maps an incoming data token d0 of type GeneId to an output token d1 of type GeneSeq (gene sequence). Using Haskell syntax we can declare the type of this actor as

$$\mathsf{GenBankG} :: \mathsf{GeneId} \rightarrow \mathsf{GeneSeq}$$

Similarly, the signature of the BLAST actor is

$$\mathsf{BLAST} :: \mathsf{GeneSeq} \rightarrow [\,\mathsf{Pid}\,]$$

i.e., the incoming gene sequence d1 is mapped to a *list* of promoter-ids d2.

Next we want to employ a GenBankP actor which can determine a promoter sequence from a promoter-id:

$$\mathsf{GenBankP} :: \mathsf{Pid} \rightarrow \mathsf{PrSeq}$$

Since the output d2 of BLAST was a list of such promoter-ids, we cannot use GenBankP directly, but have to use the process map(GenBankP) on d2 to produce a list of promoter sequences d3. In Figure 2, a box f with doubled lines denotes a process map(f). In particular, we have

$$\mathsf{d3} = \mathsf{map}(\mathsf{GenBankP})(\mathsf{d2})$$

One branch of the PIW now applies the process map(Transfac) to the list of promoter regions d4 to obtain a list of lists d5 of transcription factor binding sites (TFBS).

In another branch, we want to create a file listing the pairs of promoter-ids with their corresponding promoter regions. This is achieved by zipping the lists d2 and d4 to obtain a list d6 of the type [(Pids, PrReg)]. The subsequent steps, deal with producing the result file d9: The pretty-printing actor $\mathsf{GPR2Str} :: (\mathsf{Pid}, \mathsf{PrReg}) \rightarrow [\mathsf{Char}]$ takes a promoter-id and a promoter region and creates a string in the correct file format for the subsequent Fasta analysis (not shown). The GPR2Str actor is also wrapped into an iterative process using map. Finally concat appends a list of strings into a single string (the Fasta formatted file) and putStr writes it.

3.2 Comparison

Clearly, the functional model of the PIW depicted in Figure 2 is much more intuitive than the ad-hoc network shown in Figure 1. Moreover it provides a *complete specification* of the involved function signatures and can be automatically type-checked (for an example, see the appendix) and executed concurrently. Actors can also be reused fairly easily and their composability is governed, in principle, solely by their type signatures (of course the designer still has to ensure that the compositions make sense semantically, but the strong underlying type system at least filters out a large number of mismatches during design time).

Last not least, let us consider the PIW subworkflow in Figure 2 with input d0 and outputs d5 and d6.[9] It corresponds to a composite actor function f_{PIW} with the signature:

$$f_{\mathsf{PIW}}\colon \mathsf{GeneId} \to [\,[\,\mathsf{TFBS}\,]\,],\,[\,(\mathsf{Pid},\,\mathsf{PrRegion})\,]$$

From the actor function f_{PIW} we can create a pipelined process F_{PIW}, accepting a sequence of gene-ids, simply by defining

$$F_{\mathsf{PIW}}(\mathbf{g}) := \mathsf{map}(f_{\mathsf{PIW}})(\mathbf{g})$$

where $\mathbf{g} = [g_1, g_2, \ldots]$ is a sequence of gene-ids. Thus, we have a simple and systematic way to model and implement processes such as F_{PIW}.

In contrast, the PIW in Figure 1 is not only unnecessarily complex and enforces sequential execution, it also makes many implicit assumptions when chaining together actors, resulting in special "designed to fit" actors which cannot be reused. In particular, it does not yield a systematic way to "pack" and "unpack" sequences of tokens and instead uses some complicated ad-hoc iteration mechanisms.

3.3 Functional Rewriting of PIW

Since the functional specification shown in Figure 2 is "declarative", in particular, all involved actors are stateless and referentially transparent, we can apply well-known rewriting techniques from functional programming to simply or optimize PIW. Figure 3 shows an equivalent simplified version of PIW. For example, we have made use of the fact that

$$\mathsf{map}(f) \circ \mathsf{map}(g) = \mathsf{map}(f \circ g)$$

to replace the two mapped actors GenBankP followed by PromoterReg, by a single mapped composite actor containing their composition PromoterReg $\circ$ GenBankP. Similarly, we can replace the zip followed by map(GPR2Str) in Figure 2 by a single zipWith(GPR2Str), where the higher-order function zipWith has the signature

$$\mathsf{zipWith} :: (\alpha \to \beta \to \gamma) \to [\alpha] \to [\beta] \to [\gamma]$$

and takes a function $f : \alpha \to \beta \to \gamma$,[10] and applies it iteratively on pairs of the type (α, β) coming from the input lists $[\alpha]$ and $[\beta]$, yielding a result list of type $[\gamma]$. The implementation in Haskell is:

$$\begin{array}{ll} \mathsf{zipWith}\ f\ (a : as)\ (b : bs) & = f\ a\ b : \mathsf{zipWith}\ f\ as\ bs \\ \mathsf{zipWith}\ \_\ \ \_\ \ \ \_ & = [\,] \end{array}$$

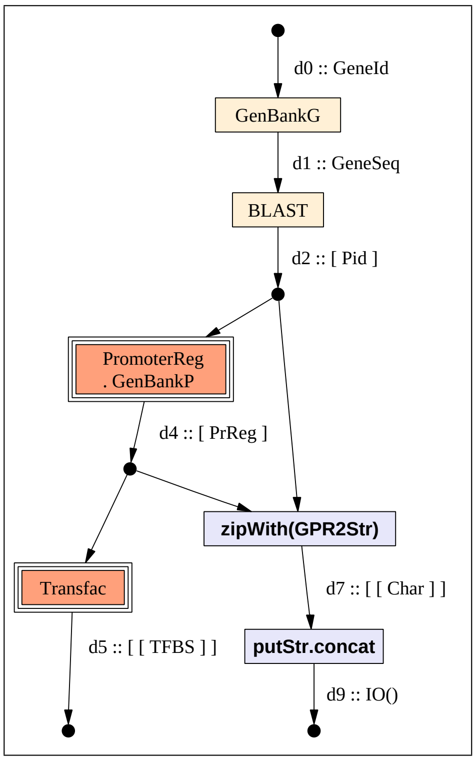

Figure 3. Simplified process network for PIW

[9] We chose d6 over d9, since the former contains the same information as the latter, however in "parsed" form (as a list of pairs (*promoter-id*, *promoter region*)), whereas d9 has glued everthing together into a single string or file.

[10] in our case GPR2Str :: Pid → PrReg → [Char]

4 Implementation in Ptolemy-II

How can we implement the PIW process networks in Figures 2 and 3 in Ptolemy-II? In fact, there is not very much to do: We simply chain together the actors in the way described, making sure their input and output ports have been modeled according to the function signatures given. In particular, atomic actors such as GenBankG and BLAST do not have to be implemented individually once our generic WSDL actor is in place. Instead they will become different instantiations of the WSDL actor.

However, there is one issue requiring some thought, *i.e.*, the composite map actors such as map(GenBankP) and F_{PIW} (Section 3.1). It seems that we have to extend Ptolemy-II in order to support those. In the following we sketch two possible implementations.

4.1 The Array Approach

Ptolemy-II provides the following two array actors which may come handy [DHK+01, Sec. 3.3.7]:

$$\begin{aligned} \mathsf{ArrayToSequence} : \quad & \mathsf{array}(\alpha) & \rightarrow \quad & [\alpha] \\ \mathsf{SequenceToArray} : \quad & [\alpha] & \rightarrow \quad & \mathsf{array}(\alpha) \end{aligned}$$

The first actor takes an input a single token of type $\mathsf{array}(\alpha)$ and produces a sequence (here denoted as a list) of type $[\alpha]$. The second actor provides the reverse functionality.[11] Now the basic idea of wrapping an actor

$$f : \alpha \rightarrow \beta$$

into a process $F = \mathsf{map}(f)$ with the signature

$$F : [\alpha] \rightarrow [\beta]$$

is to simply introduce a pair of the above array actors f_{as} (ArrayToSequence) and f_{sa} (SequenceToArray) "around" f. With this we get

$$F(\mathbf{x}) = \mathsf{map}(f)(\mathbf{x}) = f_{sa} \circ f \circ f_{as}(\mathbf{x})$$

for the list (or "array" in Ptolemy-II terminology) $\mathbf{x} = [x_1, \ldots, x_\ell]$. The first actor f_{as} takes a single data token $\mathbf{x}$ and breaks it into n tokens which are sent, one at a time, to f. The subsequent actor f_{sa} puts the individual elements back together into a single list token $\mathbf{y} = [f(x_1), \ldots, f(x_\ell)]$:

$$\mathbf{x} \rightarrow \boxed{f_{as}} \xrightarrow{x_1, x_2, \ldots} \boxed{f} \xrightarrow{f(x_1), f(x_2), \ldots} \boxed{f_{sa}} \rightarrow \mathbf{y} \qquad (1)$$

One problem remains: While f_{as} "knows" the length ℓ of $\mathbf{x}$, the subsequent inverse actor f_{sa} has no direct knowledge of ℓ thus cannot determine when to "wrap up" the received $f(x_i)$ into a single $\mathbf{y}$. We can solve this problem by modifying the pair of actors such that the f_{as} actor ***forwards*** ℓ to the f_{s2a} actor, *i.e.*, we create an additional edge $f_{as} \xrightarrow{\ell} f_{sa}$ along which the length ℓ is sent. Note that this is a general, generic, and much cleaner solution then the ad-hoc solution in Figure 1. Also it does not require any changes to f.

4.2 The Control-Token Approach

Alternatively, instead of forwarding the length ℓ on a new channel between f_{as} and f_{sa}, we can insert an end-of-array control token $\langle\mathsf{eoa}\rangle$ directly into the data stream. In this case f would simply have to forward (but otherwise ignore) the $\langle\mathsf{eoa}\rangle$ token, which indicates to f_{sa} when it can package the $f(x_i)$ into $\mathbf{y}$.

The control-token approach can also be generalized to support "*pipelined arrays*": Assume that the above array $\mathbf{x} = [x_1, \ldots, x_\ell]$ is not produced as a *single* array token, but instead by a process which produces the array elements *one at a time.* How is this different from the sequence $x_1, \ldots, x_\ell$? The difference is that a pipelined array actor would insert into the data stream the control tokens $\langle\mathsf{boa}\rangle$ (beginning of array) and $\langle\mathsf{eoa}\rangle$ as delimiters between successive arrays.

For example, consider that a conventional array actor f_c successively sends n arrays $\mathbf{x}_1, \ldots, \mathbf{x}_n$ into the network (1) above. This means that f_c fires n times, once for each array. A pipelined array actor f_p would instead fire $\ell(i)$ times for each individual array $\mathbf{x}_i$, and insert control tokens into the data

[11] In Ptolemy-II both actors have an additional parameter ℓ, which can be used to guarantee a fixed length of the produced/consumed sequence of elements. Such a guarantee is necessary for the statically scheduled SDF domain which requires a fixed number of firings at compile-time.

stream to mark the array bounderies:

$$\begin{gathered}\langle\mathsf{boa}\rangle, x_1^1, \ldots, x_{\ell(1)}^1, \langle\mathsf{eoa}\rangle,\\ \vdots\\ \langle\mathsf{boa}\rangle, x_1^n, \ldots, x_{\ell(n)}^n, \langle\mathsf{eoa}\rangle\end{gathered}$$

In this case, the above construction (1) is simplified as follows: The f_{as} actor can be eliminated. The f actor needs to forward but otherwise ignore the control tokens. If the actor to the right of (1) expects a conventional array, then the f_{sa} actor uses the control tokens in the data stream to assemble the **y** arrays. If however, the subsequent array actor is pipelined as well, then f_{sa} can be eliminated, too.

5 Conclusions

The current (SSDBM'03) version of the promoter identification workflow (let's call it PIW-1) can be seen as a "custom implementation" of PIW in Ptolemy-II, similar to how the initial prototype PIW-0 was a custom implementation in Perl (albeit in a more appealing graphical, and *apparently* reusable way). However, practically all of the main actors, both the "*bio web service*" ones (BLAST, GenBank, ...) as well as the *control flow actors* (EnumHomolog, EnumItem Triggered, ...) are *not* readily reusable in contexts other than the specific ones for which they were designed. In other words, they were ***designed to fit***, which is not a scalable solution!

Next Steps. The good news is that we now have a better understanding of the workflow programming constructs needed to create the PIW based on *fully generic actors* (let's call that version PIW-G for now):

- First and foremost, we need *declarative iteration constructs* such as a map actor implementing the corresponding higher-order function. In fact, a whole actor *library* of declarative functional programming constructs (zip, zipWith, concat, if-then-else...) will be useful to empower the workflow designer with convenient, reusable actors. Ptolemy-II has some partial support for these kinds of actors, but extensions by the SPA team will be needed.

- We also need some *generic data transformation actors* that can take XSLT, XQuery, Perl, Python, etc. transformations scripts. An initial version of this actor is in the making. However, we may need to revise and improve its design to more tightly integrate with Ptolemy-II. For example, can we take advantage of its built-in type checking and conversion features for our structure transformations by adding XML Schema types to actor ports? Also we should think how we could automatically generate some of those transformations, *e.g.*, via XWrap Composer.

- The PIW specification in Haskell also shows the power of having a type system (using parametric polymorphism). We should think about how some *semantic typing issues* can be realized in Ptolemy-II or by our extensions (*e.g.*, we may know that $\{\mathsf{CDNA\_seq}, \mathsf{Protein\_seq}\} \prec \mathsf{Sequence}$ and exploit that in automatic type conversions).

- There is a large list of other features and extensions we may want to consider in the future, *e.g.*, the *compilation approach* (AWF + system info → EWF, with different EWF target languages and issues such as space preallocation (CondorG), distributed execution etc.), the *design-mode approach* (allowing the user to design and statically analyse a workflow where actors do not have to exist yet; SEEK is working on this), extended *interactive monitoring, check-pointing, and resume* functionality. Moreover, we need to think about a good integration strategy with ID-MAF, *e.g.*, interoperability with ASPECT.

Acknowledgements. SDM/SPA is a team effort depending on active contributions from many. Special thanks to Zhengang Cheng (NCSU) for tightly working with us on the Ptolemy-II extensions, to Chad Berkley (UCSB) for Java Ant builds to merge the SEEK and SPA codes, and to

Steve Neuendorffer (UCB) for moral support and educating us about the wonders of Ptolemy-II ...

References

[ABB+03] I. Altintas, S. Bhagwanani, D. Buttler, S. Chandra, Z. Cheng, M. Coleman, T. Critchlow, A. Gupta, W. Han, L. Liu, B. Ludäscher, C. Pu, R. Moore, A. Shoshani, and M. Vouk. A Modeling and Execution Environment for Distributed Scientific Workflows. In *15th SSDBM*, 2003. `http://kbis.sdsc.edu/SciDAC-SDM/p1-ssdbm-demo.pdf`.

[DHK+01] J. Davis, II, C. Hylands, B. Kienhuis, E. A. Lee, J. Liu, X. Liu, L. Muliadi, S. Neuendorffer, J. Tsay, B. Vogel, and Y. Xiong. Ptolemy-II: Heterogeneous Concurrent Modeling and Design in Java. Technical Report UCB/ERL M01/12, Dept. of EECS, UC Berkeley, 2001. `http://ptolemy.eecs.berkeley.edu/ptolemyII/designdoc.htm`.

[Has] Haskell Functional Programming Language. `http://haskell.org`.

[KM77] G. Kahn and D. B. MacQueen. Coroutines and Networks of Parallel Processes. *Proc. of the IFIP Congress 77*, 1977.

[LP95] E. A. Lee and T. Parks. Dataflow Process Networks. *Proceedings of the IEEE*, 83(5):773–799, May 1995. `http://citeseer.nj.nec.com/455847.html`.

[NCB02] National Center for Biotechnology Information (NCBI). `http://www.ncbi.nlm.nih.gov/`, 2002.

[PTO03] Ptolemy-II project and system. Department of EECS, UC Berkeley, 2003. `http://ptolemy.eecs.berkeley.edu/ptolemyII/`.

[SEE03] Science Environment for Ecological Knowledge (SEEK) project. `http://seek.ecoinformatics.org/`, 2003.

A Complete Process Network Specification of PIW in Haskell

```
{-----------------------------------------------------------------------------
Functional Specification of the PIW Workflow
File:                   piw.hs
Author:                 Bertram Ludaescher (ludaesch@sdsc.edu)
Contributing author:    Ilkay Altintas (altintas@sdsc.edu)
Last changed:           Wed 08/20/2003
-----------------------------------------------------------------------------}

{-----------------------------------------------------------------------------
The following data constructors are used to "semantically tag" data in a
way very similar to XML tags. ('deriving Show' is for printing those types)
-----------------------------------------------------------------------------}

data GeneId  = Gid String  deriving Show
data GeneSeq = Gseq String deriving Show
data PromoterId = Pid String deriving Show
data PromoterSeq = Pseq String deriving Show
data PromoterRegion = PR String deriving Show
data TFBS = Tfbs String deriving Show

{-----------------------------------------------------------------------------
Next we have the following FUNCTION DECLARATIONS, representing the
actors needed in the PIW. Note that we are mostly interested in the
function signatures. The actual function definition are "dummy
definitions" which produce some symbolic representation of what an
actual example would produce.
-----------------------------------------------------------------------------}

genBankG :: GeneId -> GeneSeq
genBankG (Gid x) = Gseq ( "GG(" ++ x ++ ")" )

genBankP :: PromoterId -> PromoterSeq
genBankP (Pid x) = Pseq ( "GP(" ++ x ++ ")" )

blast :: GeneSeq -> [PromoterId]
blast (Gseq x)  = [ Pid ( "B(" ++ x ++ ")[" ++ show i ++ "]" ) | i <- [1..2] ]

promoterRegion :: PromoterSeq -> PromoterRegion
promoterRegion (Pseq x) = PR ("PR(" ++ x ++ ")")

transfac :: PromoterRegion -> [TFBS]
transfac (PR x)  =  [Tfbs ( "TF(" ++ x ++ ")[" ++ show i ++"]") | i <- ['a'..'b'] ]

gpr2str :: (PromoterId, PromoterRegion) -> String
gpr2str (Pid x, PR y) = "> " ++ show x ++ "\n" ++ show y ++ "\n"
```

```
{----------------------------------------------------------------------------
*** THIS BEAUTIFUL SET OF EQUATIONS DEFINES THE PIW AS A PROCESS NETWORK! ***
----------------------------------------------------------------------------}

d0 = Gid "7"               -- start with some gene-id
d1 = genBankG d0           -- get its gene sequence from GenBank
d2 = blast d1              -- BLAST it to get a list of potential promoters
d3 = map genBankP d2       -- get the corresponding list of promoter sequences
d4 = map promoterRegion d3 -- compute the list of promoter regions and ...
d5 = map transfac d4       -- ... get the list of transcription factor binding sites
d6 = zip d2 d4             -- create the list of pairs of promoter ids and their regions
d7 = map gpr2str d6        -- pretty print into a list of strings
d8 = concat d7             -- concat into a single "file"
d9 = putStr d8             -- output that file

{----------------------------------------------------------------------------
Based on some well-known FUNCTIONAL REWRITINGS, we can SIMPLIFY PIW in
various ways. Here is one such simplified equivalent version of PIW
----------------------------------------------------------------------------}

-- Apart from the use of 'currying' this is identical to gpr2str
gpr2str' :: PromoterId -> PromoterRegion -> String
gpr2str' (Pid x) (PR y) = "> " ++ show x ++ "\n" ++ show y ++ "\n"

-- Explicit definition  of a composite actor
-- (here: GenBankP followed by PromoterRegion)
genBankP_PromoterRegion = promoterRegion . genBankP

-- The definitions of d0, d1, d2 are identical to the ones above
-- d3, d6, and d8 are eliminated
-- d4', d5', d7' and d9' are defined as follows (their values are
-- identical to  the unprimed versions above)

d4' = map genBankP_PromoterRegion d2 -- use of the composite actor
d5' = map transfac d4'               -- as above
d7' = zipWith gpr2str' d2 d4'        -- combines zip and map(gpr2str)
d9' = putStr (concat d7')            -- another (implicit) composite actor

{---------------------------------------------------------------------------
UNCOMMENT the following lines, one at a time, to see the Hugs system report
two simple typing errors.
---------------------------------------------------------------------------}

-- type_error1  = genBankP (blast d1)
-- type_error2  = zipWith gpr2str d2 d4
```

B Sample Trace (Types of Data Expressions)

Note how the various data expressions $\mathtt{d}_i$ are automatically typed by the system. Their symbolic "dummy" values are also instructive since they encode the processing history of data items.

```
__   __ __  __  ____   ___      _________________________________________
||   || ||  || ||  || ||__      Hugs 98: Based on the Haskell 98 standard
||___|| ||__|| ||__||  __||     Copyright (c) 1994-2002
||---||         ___||           World Wide Web: http://haskell.org/hugs
||   ||                         Report bugs to: hugs-bugs@haskell.org
||   || Version: Nov 2002       _________________________________________

Haskell 98 mode: Restart with command line option -98 to enable extensions
...
Reading file "c:\my\Haskell\piw.hs":
Parsing.................................................................
Dependency analysis.....................................................
Type checking...........................................................
Compiling...............................................................
...
Main> d0
Gid "7" :: GeneId
Main> d1
Gseq "GG(7)" :: GeneSeq
Main> d2
[Pid "B(GG(7))[1]",Pid "B(GG(7))[2]"] :: [PromoterId]
Main> d3
[Pseq "GP(B(GG(7))[1])",Pseq "GP(B(GG(7))[2])"] :: [PromoterSeq]
Main> d4
[PR "PR(GP(B(GG(7))[1]))",PR "PR(GP(B(GG(7))[2]))"] :: [PromoterRegion]
Main> d4'
[PR "PR(GP(B(GG(7))[1]))",PR "PR(GP(B(GG(7))[2]))"] :: [PromoterRegion]
Main> d5
[[Tfbs "TF(PR(GP(B(GG(7))[1])))['a']",Tfbs "TF(PR(GP(B(GG(7))[1])))['b']"],
 [Tfbs "TF(PR(GP(B(GG(7))[2])))['a']",Tfbs "TF(PR(GP(B(GG(7))[2])))['b']"]] :: [[TFBS]]
Main> d5'
[[Tfbs "TF(PR(GP(B(GG(7))[1])))['a']",Tfbs "TF(PR(GP(B(GG(7))[1])))['b']"],
 [Tfbs "TF(PR(GP(B(GG(7))[2])))['a']",Tfbs "TF(PR(GP(B(GG(7))[2])))['b']"]] :: [[TFBS]]
Main> d6
[(Pid "B(GG(7))[1]",PR "PR(GP(B(GG(7))[1]))"),(Pid "B(GG(7))[2]",PR "PR(GP(B(GG(7))[2]))")]
  :: [(PromoterId,PromoterRegion)]
Main> d7
["> \"B(GG(7))[1]\"\n\"PR(GP(B(GG(7))[1]))\"\n","> \"B(GG(7))[2]\"\n\"PR(GP(B(GG(7))[2]))\"\n"] :: [[Char]]
Main> d7'
["> \"B(GG(7))[1]\"\n\"PR(GP(B(GG(7))[1]))\"\n","> \"B(GG(7))[2]\"\n\"PR(GP(B(GG(7))[2]))\"\n"] :: [[Char]]
Main> d8
"> \"B(GG(7))[1]\"\n\"PR(GP(B(GG(7))[1]))\"\n> \"B(GG(7))[2]\"\n\"PR(GP(B(GG(7))[2]))\"\n" :: [Char]
Main> d9
> "B(GG(7))[1]"
"PR(GP(B(GG(7))[1]))"
> "B(GG(7))[2]"
"PR(GP(B(GG(7))[2]))"  :: IO ()
Main> d9'
> "B(GG(7))[1]"
"PR(GP(B(GG(7))[1]))"
> "B(GG(7))[2]"
"PR(GP(B(GG(7))[2]))"  :: IO ()
```

B.1 Examples of Caught Typing Errors

We show two simple example of how the powerful typing system can help catch errors at compile-time.

B.1.1 "Granularity Mismatch"

The statement that produces the error is

```
        type_error1  = genBankP (blast d1)
```

The problem is that `blast d1` produces a list of promoter-ids while `genBankP` can handle only one promoter-id at at time:

```
Hugs session for:
c:\Program Files\Hugs98\lib\Prelude.hs
c:\my\Haskell\piw.hs
Main> :load c:\my\Haskell\piw.hs
Reading file "c:\my\Haskell\piw.hs":
Parsing...............................................................
Dependency analysis...................................................
Type checking.........................................
ERROR "c:\my\Haskell\piw.hs":104 - Type error in application
*** Expression     : genBankP (blast d1)
*** Term           : blast d1
*** Type           : [PromoterId]
*** Does not match : PromoterId
```

B.1.2 Currying Needed

The statement that produces the error is

```
        type_error2  = zipWith gpr2str d2 d4
```

The problem is that `zipWith` expects a "curried" function of type $f : a \to b \to c$, while `gpr2str` has the form $f : a \times b \to c$ (hence our definition of `gpr2str'` above):

```
Hugs session for:
c:\Program Files\Hugs98\lib\Prelude.hs
c:\my\Haskell\piw.hs
Main> :load c:\my\Haskell\piw.hs
Reading file "c:\my\Haskell\piw.hs":
Parsing...............................................................
Dependency analysis...................................................
Type checking.........................................
ERROR "c:\my\Haskell\piw.hs":105 - Type error in application
*** Expression     : zipWith gpr2str d2 d4
*** Term           : gpr2str
*** Type           : (PromoterId,PromoterRegion) -> String
*** Does not match : PromoterId -> PromoterRegion -> a
```