Technical Report TR03-017

Department of Computer Science
Univ. of North Carolina at Chapel Hill

DeCo: A Declarative Coordination Framework for Scientific Model Federations

Dean Herington and David Stotts

Department of Computer Science
University of North Carolina
Chapel Hill, NC 27599-3175

stotts@cs.unc.edu

May 21, 2003

DeCo: A Declarative Coordination Framework for Scientific Model Federations

Dean Herington and David Stotts
Department of Computer Science
University of North Carolina at Chapel Hill
Chapel Hill, NC 27599-3175 USA
E-mail: {heringto,stotts}@cs.unc.edu

Abstract

Program federation is assembling a software system from cooperating but independent application programs. We present DeCo, a declarative approach to creating and coordinating federations, and discuss its application in the domain of scientific computing. Manual programming is the current norm for creating model federations; in one example we studied, 80,000 lines of Unix scripts were manually generated to federate five weather simulation models into a system that can produce a 12-hour forecast in under 12 hours. DeCo is meant to automate several aspects of this currently manual process.

Participating programs and data files are described formally. The expression of a compound execution, a federation, is given in the functional language Haskell, extended with operations for large-grain program description and coordination. The declarative expression of a federation in terms of data flow among the component programs captures synchronization requirements implicitly and exploits the inherent concurrency automatically. Haskell compilation, notably its rigorous type checking, ensures the consistency of the federation. Operation of the coordination framework is demonstrated on a federation of Fortran programs that simulate environmental processes in the Neuse River estuary of North Carolina.

1. Introduction

We use the term *program federation* to refer to assembling a software system from cooperating but independent application programs. Combining existing large-scale components has several well-known benefits: reduced cost of construction, better modularity, greater concurrency, and increased potential for reuse.

Manual programming is typically used to combine and coordinate the components of a program federation. While such a manual process can achieve some of the benefits of program federation, it fails to realize these benefits fully. Moreover, a manual process is unnecessarily tedious and error-prone.

We have built an experimental system called *DeCo* (for Declarative Coordination) to automate aspects of the construction of program federations. A program federation is expressed in the functional programming language Haskell [11, 7], extended with operations for large-grain program description and coordination. The DeCo system itself consists of approximately 2150 lines of Haskell. It is built using the Glasgow Haskell Compiler [12] and runs on the Linux operating system.

There are two key elements to DeCo's approach to program federation.

- Federation components, as well as the federation itself, are described in Haskell, facilitating manipulation by the framework. Besides enhancing comprehension and documentation of the federation, this degree of formality allows DeCo to check the consistency of the federation and enables several kinds of automation. Aspects of program federation that are automated include: reformatting data; converting data streams among files, channels, and program values; synchronizing the execution of component application programs; and managing directories and files.

- The federation is specified declaratively, to the greatest extent possible. Declarative specification makes both DeCo's formal automation and a federation author's informal automation simpler and more effective. The declarative approach leads to explicit, concise, and separate specification of program federation issues, much as for component adaptation issues as described in [2].

DeCo is targeted at scientific model federations suited to large-grain coordination, that is, where the unit of execution tends to be an entire execution (possibly repeated many times) of a program component. The constituent models tend to be existing application programs, usually large and written in imperative programming languages such as Fortran. They communicate, as whole programs, through files and operating system channels, rather than as subroutines via shared variables or message passing. The models deal with many, often large files containing data in a variety of formats.

The remainder of this paper is structured as follows. Section 2 gives an overview of the architecture of the framework provided by DeCo. Section 3 describes a case study in which we used DeCo to federate two existing scientific models. Section 4 reviews the ways DeCo helps to automate the task of program federation. Section 5 discusses our experience to date with DeCo. Section 6 mentions related work. Section 7 concludes.

2. Framework architecture

This section outlines the architecture DeCo presents for creating program federations. We first introduce the major concepts by describing the main components of a federation. Subsequently, we treat DeCo's typing and execution models. Then we describe some of the operations used to construct a federation, to prepare for later examples. Finally, we briefly discuss some pragmatic issues concerning files and directories.

2.1. Main components

The preexisting components from which a federation is constructed are application programs and data files. DeCo models these entities with the more abstract notions of *executor* and *stream* to present a simple yet rich semantic model for coordination. These abstract notions are realized as concrete Haskell entities in the Haskell code that constitutes the federation.

A stream is represented by the abstract datatype

```
newtype Data t
```

An entity of type `(Data` t`)` represents an aggregation of data of type t. The type of a stream is arbitrary, and thus so also is its size.

Note that a stream is not necessarily a sequence of elements. `(Data Char)` denotes a stream consisting of a single character, while `(Data [Char])` denotes one consisting of a sequence of characters. The use of the term "stream" is intended to indicate that the stream's contents are read and written in order from the beginning toward the end, not that the stream implicitly contains a sequence of elements. At the same time, the name `Data` is chosen for the abstract type to emphasize that a stream's contents can be treated as a whole, as will be seen.

A stream's contents can exist in one of three forms: as a Haskell *value*, as the contents of a *file*, or as the contents of a *channel* (the data read from or written to an operating system file descriptor). By making `Data` an abstract type, the framework allows the three forms of stream to be used interchangeably while accommodating different sorts of data connection between processes in an efficient manner.

An executor is represented by a Haskell function with abstract result type

```
newtype EX t
```

Having a result type of `(EX` t`)` allows a function to manipulate streams.

The most important kind of executor is one that serves as a proxy in the federation program for an existing external application program. Such an executor invokes a DeCo-supplied utility to start a subordinate process, having set up input and output connections appropriately. On the other hand, an executor may also be implemented purely in Haskell, creating no subordinate process. Such an executor might be used to transform stream contents from one set of types to another, for example. However an executor is implemented, it is used in the same way. Moreover, since an executor is simply a Haskell function, it is first-class: it may be higher-order, it may be partially applied, *etc*.

A stream *connection* represents a unidirectional data flow between executors. The executor providing the stream (and hence defining the contents of the stream) is termed its *producer*. The executor using the stream (and hence relying on the contents of the stream) is termed its *consumer*.

The fact that federation *metadata*—descriptions of the federation's components—are expressed in Haskell is crucial to DeCo's capabilities. First, it means that Haskell type checking ensures the consistency of a federation. Second, it allows DeCo to mediate stream connections between executors where there exist discrepancies.

The expression of the federation itself is also given in Haskell. A federation is cast simply as an executor that interconnects preexisting external components, using streams, other executors, and metadata.

The abstract datatype `(EX` t`)` is an extension of Haskell's built-in type `(IO` t`)`, which is used for input/output actions. Hence, an executor, whose type is `(EX` t`)`, may perform such side effects as part of its execution.

```
io :: IO a -> EX a
```

This ability is crucial for executors that serve as proxies for application programs, as they need to create processes, files, and directories.

A federation is executed with

```
runEX :: [String] -> EX a -> IO a
```

The function `runEX` is passed a list of option strings and an executor. The executor is executed and its result returned.

The design of the executor abstraction—in particular, that an executor can serve as a proxy for an application program—has another advantage. A federation expressed as a Haskell program can be used as an application program in a yet larger federation. Thus, federations are appropriately compositional.

2.2. Typing model

To allow for widely varying data storage formats while providing maximum flexibility for data streams, DeCo defines a two-level typing scheme.

- The (abstract or high-level) type of a stream, expressed as a Haskell type, captures the high-level semantics of the stream data.
- The (concrete or low-level) representation of a stream, encoded separately, specifies the storage format of stream data in a particular context.

This two-level scheme separates the essential type of data from its packaging as a stream. The separation is akin to the difference between abstract and concrete syntax in a programming language.

The two-level approach to stream typing provides several key benefits.

- DeCo's notion of stream compatibility is simple and clear, being defined in terms of Haskell types.
- Stream compatibility is very general, as it is defined to ignore matters of representation.
- DeCo can automatically mediate between a stream producer and its consumer when the produced and consumed representations differ.

Recall that a stream has the Haskell abstract type `(Data` t`)`, for some t. The type parameter t exactly encodes the high-level type of the stream. As a result, Haskell's type checking ensures that streams are used in a type-safe manner. Moreover, Haskell's type inferencing relieves the federation programmer in most cases from the need to declare stream types explicitly.

At the level of data storage, the contents of a stream are deemed to consist of a sequence of bytes. The low-level representation of a stream is cast as a translation between a stream of bytes and a Haskell value of the appropriate type. A decoding function translates a stream of bytes to a Haskell value. An encoding function translates a Haskell value to a stream of bytes.

An important property of (low-level) representations is that there may be more than one of them for a given (high-level) type. The alternative representations for a type are expressed as distinct Haskell types. A representation type r is associated with a Haskell type t with an instance declaration for `ReprType` r t. In this way, both new representations and new associations between representations and types can easily be defined.

Data representations are declarative and, in particular, compositional. For example, the representation

```
TailSeq (ILenSeq 3 (UInt BE 16))
```

specifies a *tail* sequence (a sequence delimited by end of stream) whose elements are implicit-length sequences of length 3 consisting of elements that are unsigned, big-endian integers of length 16 bits. Such a representation is suitable for the two types `[[Integer]]` (lists of lists of unbounded integers) and `[[Int]]` (lists of lists of bounded integers).

2.3. Execution model

A federation is essentially expressed as a directed data-flow graph, where the nodes are executor invocations (*executions*) and the edges are stream connections among them. Federation control flow—that is, the temporal sequence of executions—is derived automatically from the data flow. The characteristics of the stream connections among executions imply the appropriate synchronization among those executions and allow DeCo to realize the inherent concurrency of the federation automatically. The complexities of this data-flow machinery are hidden from the federation programmer by the abstract type `(EX` t`)`.

Complexities of a related sort are hidden by the abstract type `(Data` t`)`. In certain situations, it is essential that a stream be accessed incrementally. That is, it must not be necessary for later portions of a stream to be generated by its producer before earlier portions of that stream may be consumed by another concurrent execution. For example, if a stream connecting two executors could be of unbounded size, it may not be acceptable for the consumer to wait for the entire stream to be produced before starting to consume it. Similarly, if the stream connecting two executors is not of unbounded size but rather subject to unbounded delay during its production, it may not be acceptable for the consumer to wait for the entire stream.

At the same time, it is useful to treat the contents of a stream in its entirety, as a single Haskell datum. Stream processing—especially data transformation—is greatly simplified if the entire contents of a stream can be directly pattern matched, passed among functions, mapped over, *etc*. Eliminating the need for explicit, incremental manipulation of stream contents allows for a much more declarative treatment of stream data.

Fortunately, these two seemingly contradictory views of a stream can be reconciled by exploiting Haskell's ability to read lazily. A Haskell input/output operation that reads and returns the remaining contents of a file or channel (such as `readFile`) returns an unevaluated string whose successive characters are subsequently read from the file or channel when their values are later demanded by the program. (This feature relies crucially on Haskell's nonstrict evaluation semantics.) Using this feature, DeCo is able to provide a fully declarative treatment of streams.

2.4. Federation construction

Although a stream usually can and should be manipulated without regard to its form (as value, file, or channel), this is clearly not always possible. A federation must commit to the form of a stream where it originates and where it terminates, including at the interface to an external program. For this purpose DeCo provides types with which to describe the forms of a stream.

- A (`File` $path$ $repr$ `::` `File` t) represents a file with pathname $path$ containing data of type t in representation $repr$.
- A (`Channel` fdx $repr$ `::` `Channel` t) represents a channel with extended file descriptor fdx containing data of type t in representation $repr$.
- A (`Value` val fin `::` `Value` t) represents a value val with *finish* value fin. (Although the detailed mechanism is beyond the scope of this paper, the finish value enables detection of "run-on" streams in the context of lazy reading of stream contents.)

These types are used with the following conversion functions.

```
fromFile    :: File t -> EX (Data t)
toFile      :: File t -> Data t -> EX ()
fromChannel :: Channel t -> EX (Data t)
toChannel   :: Channel t -> Data t -> EX ()
fromValue   :: Value t -> EX (Data t)
toValue     :: Data t -> EX (Value t)
```

When immediate conversion to or from a value is desired, the following functions are useful.

```
fromVal     :: t -> EX (Data t)
toVal       :: Data t -> EX t
```

In addition, the function `from` is overloaded and can substitute for `fromFile`, `fromChannel`, and `fromValue`, and the function `to` is overloaded and can substitute for `toFile` and `toChannel`.

2.5. Files and Directories

During the execution of a federation, three directories are maintained by DeCo.

- The *top directory* is the directory that was current when the federation began execution. The top directory remains fixed for the duration of federation execution. It serves as a reference point within the invoker's directory environment.
- The *run directory* is a new directory created when the federation begins execution. It is the root of a directory subtree that serves as a repository for new files created by the federation. The run directory name and location are under control of the invoker of the federation. The run directory remains fixed for the duration of federation execution.
- The *current directory* is a directory within the subtree rooted at the run directory that associates a portion of that subtree with the currently executing federation code. The current directory starts out equal to the run directory but may vary under control of federation code. It defines a "directory scope" during execution; in particular, it provides the default initial directory for external program invocations.

The operating system maintains a "current working directory" for a process that may vary during process execution. Varying the current working directory during federation execution, however, would lead to unpredictable results, because executors are implemented as concurrent Haskell threads. Instead, DeCo leaves the actual current working directory unchanged and provides a virtual one (the current directory introduced above) that works properly in the presence of multiple threads.

Federation code manages the current directory with `inDir`.

```
inDir :: FilePath -> EX a -> EX a
```

(`inDir` dir act) creates a directory named dir in the current directory, then performs act with dir as the current directory. In other words, `inDir` opens a new, temporary, directory scope for the execution of a subordinate action. Note that, although the current directory reverts after the subordinate action completes, the file subtree rooted at the newly created directory persists.

DeCo's interpretation of pathnames is extended to provide access to the top, run, and current directories, as shown in the following table. The remainder of a pathname with the indicated initial character is interpreted relative to the corresponding directory.

@	top
#	run
$	current

In addition, a pathname beginning with a `/` character is interpreted as usual, whereas a pathname beginning with a character other than these four (`@ # $ /`) is interpreted relative to the top directory (as if it were preceded by `@`).

Having described the design of DeCo, we now proceed to apply it to a realistic case study.

3. Case study

DeCo was applied to a realistically complex case study in order to assess its effectiveness. The case study involved the federation of two existing environmental models for aspects of the Neuse River estuary in eastern North Carolina. The first models estuary water quality through time, given initial concentrations, inflow rates, and outflow rates of water constituents, plus meteorological data for the modeled time period. The second models chemical processes in the sediment underlying the river, computing fluxes of constituents between water and sediment.

The water model is a single program of approximately 9300 lines. The sediment model consists of two programs, whose total size is approximately 4700 lines. Both models are written in Fortran 77. (Note that DeCo makes no restriction with respect to the source language of a component program. External programs to be federated may be written in any language.) Together the two models read and write dozens of files during their execution.

The goal of combining these two models was to obtain a more precise simulation of the physical, chemical, and biological processes occurring in the Neuse River estuary. When run separately, each model makes simple assumptions about the other's medium: the water model about the sediment, and the sediment model about the water. In the federation, each model provides a more sophisticated simulation of its medium for the other model.

The water model operates in two spatial dimensions. It models depth by dividing the river into half-meter-thick layers. It models length along the course of the river by dividing the river into 59 segments, with an average length of just over one kilometer. Width of the river is not modeled.

Besides the two spatial dimensions, the water model models time. The desired simulation period is divided into many time intervals or steps. During each time step (from earlier to later times), the various physical parameters are computed for each layer of each segment, moving along the river in the downstream direction. The size of a time step is varied heuristically to save computation time: Longer time steps are used when physical processes operate more slowly.

The sediment model, on the other hand, is spatially zero-dimensional; its calculations apply to a single point where water and sediment meet. Given concentrations of relevant chemicals in the bottom water, it computes fluxes for these chemicals between sediment and water.

In federating the water and sediment models, the desired effect is that the two models act as coroutines. That is, as the water model steps through time for a particular segment of the river, an instance of the sediment model for that segment takes the same steps through time. Note that the difference in dimensionality between the two models is accommodated by replicating the sediment model along the length of the river. Each model evolves its own medium, taking initial conditions from and delivering final conditions to the other model for each time step.

While conceptually sound, the approach just described suffers several practical problems.

- Representing the two models as actual coroutines of each other would require major rework of the existing programs, involving either merger of the two programs—thereby losing modularity—or significant restructuring of both to allow them to interoperate more intimately.
- The much greater execution cost for the sediment model compared to the water model (approximately five times) makes it prohibitively expensive to run the sediment model with the same frequency as the water model.
- Moreover, and fortunately so, sediment changes much more slowly than water, so it is fruitless to run the sediment model at the same frequency as the water model.
- Because the calculations of the sediment model depend on environmental parameters measured at only four sampling stations along the river, the sediment model can only sensibly be replicated four times to cover the length of the river.

The solution to the problems described above is to leave the models as separate programs and to arrange for their executions to alternate. The sediment model is replicated four times, once for each *region* of the river. These four instances of the sediment model are run at a lower frequency than the water model. Data passed from the water model to the sediment model are averaged within each region and over groups of water model time steps that correspond to a single sediment model time step. Conversely, data from the four instances of the sediment model are replicated for the segments corresponding to a region and then joined along the river's length before being passed to the water model.

Figure 1 illustrates how the two models divide the study area of Neuse River estuary differently and how this spatial mismatch is resolved. The narrow bands that cross the river's width represent the water model's segments, numbered 2 through 60 above the river. These segments are grouped into four regions, numbered 1 through 4 below the river. Each region contains one sediment sampling station, shown in the figure as a thick dot.

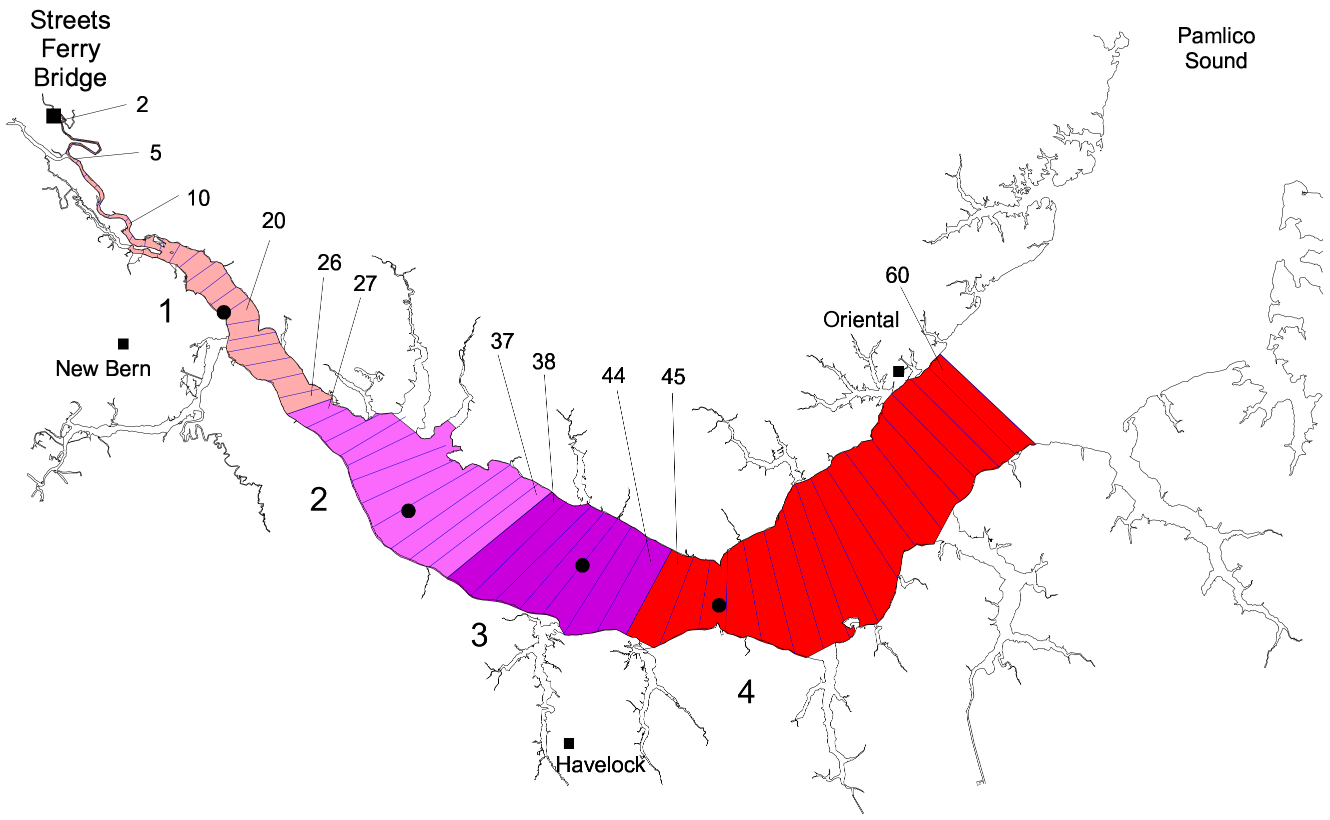

Figure 1. Neuse River study area

In addition to the differences in dimensionality and granularity described above, there are other, smaller differences to be mediated between the two models.

- The units in which data are expressed in the two models differ.
- The forms and formats in which the data are stored differ between models.

The program federation for the Neuse River study is too large to show here in its entirety. Instead we give several fragments as examples of federation code. We do not attempt to explain every detail of these examples; their purpose is rather to give a sense of the level of expression of federation code.

We start by considering a single step of the main iteration, during which the water model is executed once and the sediment model four times, as described earlier. Figure 2 shows the flow diagram for a single step of the main iteration. The step begins—and ends—at the lower left, with flow proceeding clockwise. Compare this diagram with the corresponding portion of federation code, shown below. Function `step` first converts the pathname from the previous step (argument `prev`) and the step information (argument `iter`) to simple Haskell values. Then it creates a fresh subdirectory named by the step number (`n`). In this subdirectory, the current fluxes are derived from the previous step's results, the water model (represented by function `wat`) is invoked in subdirectory `wat`, the information needed by the sediment model is extracted from the water model's main output, the input to the sediment model is prepared, the sediment model (represented by function `sed`, which invokes four copies of the actual sediment model program) is invoked in subdirectory `sed`, the newly computed fluxes are stored, and the current step's subdirectory pathname is returned for use by the next step.

```
step prev iter = do
  prevElts   <- toVal prev
  (n, times) <- toVal iter
  inDir (show n) $ do
    watIn    <- watFluxes prevElts
    watOut   <- inDir "wat" (wat times watIn)
    watToSed <- toVal watOut >>= extractParams
    sedIn    <- sedInput n times watToSed
    sedOut   <- inDir "sed" (sed prevElts sedIn)
    fromVal sedOut >>= to fluxesFile
    curDirElts >>= fromVal
```

The value `fluxesFile` defines a file stream with type `SedAllOut` and representation `sedAllOut` (not shown here). (In Haskell, the names for types begin with uppercase letters and those for values begin with lowercase letters.)

```
fluxesFile :: File SedAllOut
fluxesFile = File "$fluxes" sedAllOut
```

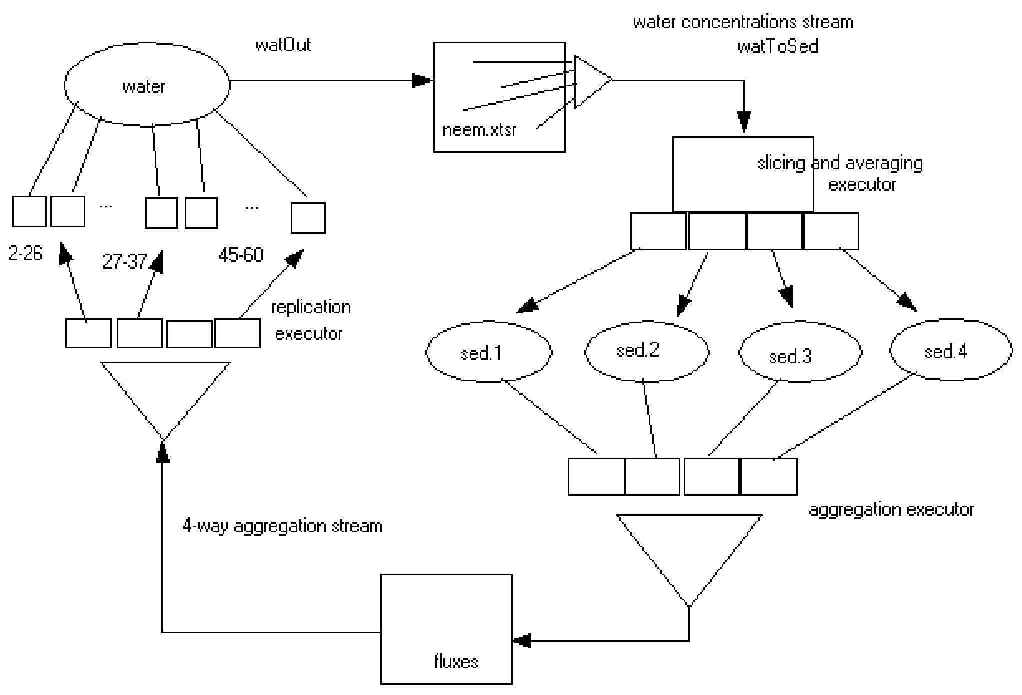

Figure 2. Neuse River study flow diagram

The function `odePOC` defines the proxy executor for external program `odePOC`, one of the two programs comprising the sediment model. The single argument `7` to the program selects batch-like behavior; the program was designed for interactive use as well.

```
odePOC pocIn = executor $ do
  pocIn′ <- wantChannel (Just sedRegIn) pocIn
  (chk, chk′) <- makeFile (Just "Parameter.chk")
                          (Just text)
  body $ \ bh -> do
    chan <- pocIn′ Nothing
    awaitTrigger bh
    process <- startProgram "odePOC" False ["7"]
                    Nothing Nothing [(0, chan)]
    awaitTermination process >>= checkStatus
    chk′ Nothing Nothing
  return (chk :: Data String)
```

4. Automating model federation

In this section we review the ways in which DeCo helps to automate the tasks of program federation. The first three subsections note specific features of the framework, whereas the last describes general capabilities resulting from the fact that DeCo is based on and built in Haskell.

4.1. Stream mediation

Automatic stream mediation is one of DeCo's major benefits. Provided the producer and consumer of a stream agree on the stream's type, mismatches in both data representation and form are automatically resolved.

Recall that the representation for a stream is specified separately from its type. The type specifies the conceptual format for the stream contents, while the representation extends that specification to the contents' actual format in terms of a byte string. A representation mismatch occurs when an actual and a desired stream have the same type but different representations. In this situation, DeCo inserts adaptation code automatically to resolve the representational discrepancy. The adaptation code decodes the byte stream of the actual stream according to the actual representation, then encodes it according to the desired representation. The addition of this extra step is transparent to the federation programmer.

The contents of a stream can exist as a file, an operating system channel, or a Haskell value. With DeCo a stream's form is independent of both its type and its representation, so that streams of all three forms can be used interchangeably. DeCo mediates mismatches in actual and desired stream form by converting the actual form to the desired one. As for stream representation, insertion of the adaptation code is transparent to the federation programmer.

4.2. Program synchronization

Because DeCo uses a data-flow execution model, the federation programmer is relieved of the responsibility for synchronizing the execution of the cooperating application programs. Programs are run when their input data streams are available. For example, if a program takes a file as input, its execution will not be started until the file has been completely written. On the other hand, if the program reads a stream via its standard input, the program can begin execution once its corresponding channel has been opened, even though not all of the stream has yet been written.

The fact that DeCo manages program synchronization reduces the context dependence of a component program. As a result, a component program can more easily be used in different federations, or in different contexts in the same federation. As a program federation evolves, changes to the forms of inputs to component programs are automatically reflected in the synchronization of these programs.

A further benefit of the data-flow approach is that the potential concurrency among component program executions is evident. DeCo naturally exploits this concurrency. Indeed, in cases where two or more component programs communicate in pipeline fashion, it is often important that these programs execute concurrently.

4.3. File and directory management

The management of files and directories, though conceptually trivial, can be a practical headache for a sizable program federation. DeCo eases this burden with two simple, declarative features.

- A portion of federation execution can be performed in a fresh subdirectory of the current directory using `(inDir` *sub act*`)`, which performs action *act* in the newly created subdirectory *sub*. The effectiveness of this simple construct relies on the ease with which an arbitrary portion of federation execution can be expressed as a single expression in Haskell.

- File pathname syntax and semantics are extended by treating several additional initial characters specially. File pathnames beginning with the characters `@ # $` refer to files relative to the top, run, and current directories, respectively. By reducing the dependence of a filename on its context, this feature facilitates changes to the overall directory structure of a federation.

4.4. Substrate support

This subsection mentions two ways that DeCo helps to automate program federation not through specific features but rather due to the use of Haskell as a substrate.

1. The use of Haskell as the base for the program federation language means that Haskell compilation performs consistency checking for a program federation. As the interfaces for program components are expressed in the federation language, federation compilation ensures that the constituent components are combined correctly. In particular, Haskell's strong type system guarantees that the types of components' input and output streams match appropriately. As a further benefit, Haskell's type inference relieves the federation programmer of specifying type information explicitly in most cases.

2. Program federations can be complicated, either because they comprise many and varied components, or because considerable code may be required to adapt some components to others, or both. A common example of the latter situation is the need to change not only the representation but also the type of a stream output by one component and input to another. The complications of a particular program federation are opportunities for *ad hoc* automation. The abstraction facilities of a high-level language such as Haskell provide a powerful set of tools for realizing such automation.

5. Discussion

In this section we give brief assessments of our experience using and implementing DeCo.

5.1. Case study experience

The use of DeCo in the Neuse River case study has been both pleasant and effective. The separate specification of stream type and stream representation copes with varying data formats while allowing streams to be treated abstractly. The abstraction of the form of a stream's content (as file, channel, or value) successfully supports the data-flow approach while allowing (in cooperation with lazy stream reading) the contents of a stream to be treated as a whole. Synchronization of component programs according to their mutual data flows makes for a highly declarative expression of control flow. Simple but declarative DeCo features reduce the burden of file and directory management to a minimum. Overall, these features maximize interoperability and reusability for federation components and make DeCo effective at automating program federation.

Based on the Neuse River case study alone, a conclusion cannot yet be drawn concerning conciseness and efficiency of program federation using DeCo. The Neuse River program federation consists of approximately 850 lines of code. Although this may seem like a large amount of code for coordinating the multiple executions of three Fortran programs, it should be remembered that these programs are executed repeatedly in alternation, that their data formats are rather complex, and that considerable data manipulation is required to combine the programs. The very high code density of the program federation suggests that its size is not excessive for the problem.

A current shortcoming in the Glasgow Haskell Compiler's runtime system has prevented us from precisely comparing the time spent executing the federation itself to the time spent in the federation's subordinate program executions. However, coarse wall-clock timing during execution of the case study shows that the executions of the subordinate Fortran programs heavily dominate. Although more precise measurement is desirable, DeCo performance is clearly not a limiting factor in the Neuse River case study.

5.2. Use of Haskell

The use of Haskell has been extremely positive, both as a base for the program federation language and as a base for the implementation of DeCo.

As the base for the federation language, Haskell offers a highly declarative foundation that greatly eases construction of a declarative domain-specific language. Haskell's abstraction capabilities—especially monads, higher-order functions, polymorphism, and type classes—allow the federation language to be both simple and powerful. Haskell's strong typing provides static consistency checking for federation programs, while its capable type inferencing greatly reduces the number of type annotations needed. Haskell's power and expressiveness make it easy for a federation programmer to provide the "connective tissue" that is inevitably required in federating existing components.

As the base for implementation of DeCo, Haskell has been very successful. First, all of the advantages cited above concerning its use as a base for the federation language pertain to its use as a base for framework implementation. More specifically, however, the power, simplicity, and efficiency of Haskell's concurrency (threads) support made design of DeCo not only feasible but also elegant and relatively easy. The availability of an interface to Posix capabilities made programming the external interactions of DeCo quite comfortable. Finally, the expert and willing assistance of the volunteers who build, maintain, and use the Glasgow Haskell Compiler system was invaluable.

6. Related work

Three strains of prior research are most relevant to our work on DeCo: coordination languages, domain-specific embedded languages, and functional shells and scripting languages.

Coordination languages (also called *configuration languages* or *module interconnection languages*) aim to provide a framework in which to express an application as an aggregation of components. At this high level, DeCo's goal is the same. However, there are significant differences in emphasis. Coordination languages tend to focus on issues of distribution and finer-grained parallelism, whereas DeCo focuses on expressing components abstractly to facilitate adaptation, composability, and reuse. Also, coordination languages are usually deliberately distinct from the computation languages in which the federated components are written, whereas DeCo exploits the fact that Haskell can be used not only for coordination but also for as much computation as is useful for a given federation (for data adaptation, for example). Examples of coordination languages include Polylith [13], Strand and PCL [5], and Linda [3].

A *domain-specific language* (*DSL*) is a language tailored to a particular application domain. A *domain-specific embedded language* (*DSEL*) is a DSL built as an extension to an existing *base* language. DeCo is constructed as a DSEL based on Haskell whose domain is program federation. As such it is part of a recent trend toward basing DSELs on Haskell. (A good explanation of Haskell's popularity in this role is given by Hudak [8].) Examples of other application domains (and representative DSELs) for which Haskell-based DSELs have been built include web programming (HaXml [17] and WASH/CGI [15]), hardware description (Lava [1] and Hawk [9]), animation ([4]), and robotics ([10]).

Shells and scripting languages share with coordination languages the high-level aim of facilitating aggregation of existing computational components, though their style is that of a more traditional programming language. Although one tends to think of a *shell language* as interactive and a *scripting language* as batch-oriented, the two notions are essentially similar, and both can be used in both ways. DeCo can be viewed as a scripting language for federations of applications. DeCo is not intended particularly for interactive use, but it can be used that way, and could easily be extended to be more convenient for such use. Shells and scripting languages are numerous; however, many fewer are especially functional in nature as is DeCo. Examples of functional shells and scripting languages include *Es* [6], scsh [14], and the shell included in Famke [16], a prototype of a strongly typed operating system.

7. Conclusions and future work

We have presented DeCo, a declarative coordination framework for scientific model federations. Specifications in DeCo are concise and convenient. The nature of DeCo's abstractions and the language Haskell on which it is based help to automate the task of program federation.

With DeCo, data are treated abstractly as *streams*, for which the data type, data representation, and form (as value, file, or operating system channel) are specified separately. The type gives the high-level semantics of the stream and is checked by Haskell. The representation gives the low-level encoding of the stream. The set of representations is easily extended to handle formats particular to a federation. The various forms of stream content facilitate reuse of streams in different contexts. DeCo automatically resolves mismatches between the actual and desired representations and/or forms of a stream.

External programs and Haskell functions are treated similarly and abstractly as *executors*. Haskell type checking ensures that executors are combined properly. Control flow is derived from the data flow among executors rather than being specified explicitly. Hence, DeCo automatically synchronizes the execution of external programs and realizes the inherent concurrency of a federation. DeCo also provides constructs to simplify the bookkeeping aspects of program federation, such as file and directory management.

An important characteristic that distinguishes DeCo from other coordination frameworks is that its basic entities—streams and executors—are composable. That is, larger, more complex entities can be built simply and predictably from smaller entities. Along with the abstract nature of streams and executors, this composability makes federation components more flexible, more interoperable, and more reusable.

DeCo was applied to a realistically complex case study, a federation of existing environmental models for the Neuse River of North Carolina. The experience showed that federation specification in DeCo can be comfortable and effective. However, more application experience is needed to substantiate such a conclusion on a usefully wide range of model federations. We plan, therefore, to apply DeCo to a number of federations with varying characteristics.

The highly declarative nature of program federation in DeCo would have been difficult to achieve had DeCo not been based on a language such as Haskell. Unfortunately, Haskell is not well-known in the community where DeCo would be most useful. However, it is plausible that DeCo's Haskell face could be masked by a veneer more comfortable to scientific modelers. We would like to explore the possibility of using a graphical user interface to specify federations, while expressing metadata in XML.

Acknowledgment

This work was supported by the U.S. Environmental Protection Agency, under grant R82-795901-3.

References

[1] P. Bjesse, K. Claessen, M. Sheeran, and S. Singh. Lava: Hardware design in haskell. In *International Conference on Functional Programming*, pages 174–184, 1998.

[2] P. Boinot, R. Marlet, J. Noye, G. Muller, and C. Consel. A declarative approach for designing and developing adaptive components. In *Proceedings of the* 15^{th} *IEEE International Conference on Automated Software Engineering (ASE 2000)*, Sept. 2000.

[3] N. Carriero and D. Gelernter. Linda in context. *Communications of the ACM*, 32(4):444–458, 1989.

[4] C. Elliott and P. Hudak. Functional reactive animation. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP '97)*, volume 32(8), pages 263–273, 1997.

[5] I. Foster. Compositional parallel programming languages. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 18(4):454–476, 1996.

[6] P. Haahr and B. Rakitzis. Es: A shell with higher-order functions. In *USENIX Winter*, pages 51–60, 1993.

[7] Haskell home page. http://www.haskell.org/.

[8] P. Hudak. Modular domain specific languages and tools. In P. Devanbu and J. Poulin, editors, *Proceedings: Fifth International Conference on Software Reuse*, pages 134–142. IEEE Computer Society Press, 1998.

[9] J. Matthews, B. Cook, and J. Launchbury. Microprocessor specification in hawk. In *International Conference on Computer Languages*, pages 90–101, 1998.

[10] J. Peterson, P. Hudak, and C. Elliott. Lambda in motion: Controlling robots with Haskell. *Lecture Notes in Computer Science*, 1551:91–105, 1999.

[11] S. Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, Apr. 2003.

[12] S. Peyton Jones et al. The Glasgow Haskell Compiler home page. http://www.haskell.org/ghc/.

[13] J. M. Purtilo. The POLYLITH software bus. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(1):151–174, 1994.

[14] O. Shivers. A Scheme shell. Technical Report MIT/LCS/TR–635, Massachusetts Institute of Technology, 1994.

[15] P. Thiemann. WASH/CGI: Server-side web scripting with sessions and typed, compositional forms. In *Practical Aspects of Declarative Languages*, pages 192–208, 2002.

[16] A. van Weelden and R. Plasmeijer. Towards a strongly typed functional operating system. In *Selected Papers Proceedings 14th International Workshop on the Implementation of Functional Languages, IFL 2002*, Madrid, Spain, 2002.

[17] M. Wallace and C. Runciman. Haskell and xml: Generic combinators or type-based translation? In P. Lee, editor, *Proc. international conference on functional programming 1999*, pages 148–259, New York, NY, 1999. ACM Press.