A Domain-sepcific Langauge for Infopipes

Rebekah Leslie

Department of Computer Science and Engineering
OGI School of Science & Engineering
Oregon Health & Science University
20000 NW Walker Road
Beaverton, OR 97006-8921 USA

Technical Report Number CSE 04-004

28th July 2004

This material is based upon work supported by the
National Science Foundation under Grant No. CCR-0219686.
Any opinions, findings and conclusions or recomendations expressed in
this material are those of the author and do not necessarily reflect the
views of the National Science Foundation.

A Domain Specific Language For Infopipes

Rebekah Leslie

July 28, 2004

1 Introduction

This paper presents the state of the domain specific language for Infopipes as of June 2004. The paper describes the langauge, preliminary implementation, and remaining development issues.

Section 2 presents the details of the language. This includes a description of the constructs available in the language, the assumptions made about the language capabilities, and the concrete representation. Section 3 presents the implementation of a configuration language for Infopipes. The configuration language is a limited subset of the proposed Infopipes language but the compiler for the configuration language provides an extensible framework for implementing the full language. Section 4 discusses the remaining decisions to be made in the language design and the issues surrounding the implementation of the Infopipes language.

2 Language Description

This section presents a domain specific language for Infopipes. The language is incomplete, but a number of design decisions have been made. Section 2.1 defines the set of features for the DSL. Section 2.2 contains the expectations of the langauge. Finally, section 2.3 illustrates the use of the language through concrete examples.

2.1 Language Constructs

The principle goal in designing the Infopipes DSL is the creation of a language that is exactly powerful enough to describe Infopipes and their operations. The design requires abstractions that fit the Infopipes domain and meet the needs of distributed systems developers. Superfluous operations are undesirable. The language features proposed in this section attempt to address this goal.

2.1.1 Component Definition

Component definition, instantiation, and connection are top level declarations in the Infopipes language. This section describes component definition. Instantiation and connection are left to section 2.1.6.

Component definitions contain methods and control variables. The methods take several forms: constructors, transfer functions, and configuration functions. Every component includes at least one constructor function which initializes instances of the component. Every component also contains a transfer function which defines the computation performed by the component. The implementation coordinates the invocation of these functions. Configuration methods are optional. The application writer invokes these methods at connection time in order to obtain information about the state of the component. Control variables maintain the state of the component. The following sections describe the method types in greater detail.

2.1.2 Constructor Functions

Each component definition contains one or more constructor functions. Constructor functions define the port layout and data flow properties of the component. They are invoked when the component is instantiated.

The application writer defines the port layout of a component using the built in functions *inport* and *outport*. Each port declaration contains the name and type of the port. The type specification has data type and polarity components. Type variables may be used in both cases. The keyword *optional* modifies port declarations. Failure to connect a port declared using *optional* will not cause an error. This capability is designed to be used with control ports.

Constructor functions may take parameters. These parameters provide information used in the initialization process. Generally these parameters are the initial values for control variables. The language allows multiple constructor functions with varying argument lists.

2.1.3 Transfer Function

Transfer functions define the computation performed by a component. Transfer functions have dependency lists that indicate which ports the function interacts with. Multiple transfer functions are allowed so long as their dependency lists are different. Each port must belong to the dependency list of exactly one transfer function.

The body of the transfer function contains arithmetic expressions, port interactions, library calls, and control flow operations. Port interactions are represented using a reciprocal arrow syntax: $\leftarrow$ gets data from an in port and stores it in a variable and $\rightarrow$ sends data to an out port. Section 2.1.5 presents the control flow statements in greater detail.

2.1.4 Configuration Methods

Configuration methods provide information about the state of a component at connection time. They are declared using the keyword *config* before the method declaration. Configuration methods contain arithmetic expressions, library calls, and control flow operations. Unlike transfer functions, configuration methods can return a value to the caller but cannot interact with ports.

2.1.5 Control Flow Operations

The Infopipes language provides three kinds of control flow statements. There is a standard if-else construct as well as two looping constructs. The looping constructs iterate over space or time. Both kinds of loops are bounded.

2.1.6 Component Instantiation and Connection

Like component definition, component instantiation and connection are top level statements in the Infopipes language. The Infopipes DSL uses the syntax presented in the Infopipes literature unchanged.

2.2 Assumptions

This section enumerates the current set of assumptions about the capabilities of the Infopipes DSL. These assumptions are a basis for further design. The assumptions may change if further experience with the language dictates that they are insufficient.

Infopipes must include a mechanism for feedback in order to maintain quality of service parameters. The current design assumes that Infopipelines control the feedback between Infopipe components. Without further restriction this assumption leads to feedback structures with an infinite depth. To avoid this obviously undesirable consequence, control ports (ports used for feedback rather than forward data flow) may be optional. With optional control ports, the user creates a feedback structure that is exactly as deep as their application requires.

As discussed in section 2.1.2, each component has a set of in ports and a set of out ports. The polarities of the ports within a set may vary from each other. This assumption complicates the synchronization of data transfer in the implementation but is essential. Initially it seemed reasonable to assume that the ports in a given set must all have the same polarity. It quickly became evident that there are many cases where this restriction is overly limiting. Theoretically, any polarity mismatches resulting from uniform polarities could be solved using an intermediate component, but this complicates the resulting pipeline and introduces unnecessary latency.

Data flow characteristics are the focus of the Infopipes architecture. For this reason, the unifying characteristic of Infopipe "classes" is shape (port layout) rather than computation. Subcomponents may override all methods in the supercomponent as long as they do not alter the shape. The subcomponent can only alter the polarity of ports in such a way that the polarity becomes more specific. For example, an $\alpha \rightarrow \beta$ polarity may become $+ \rightarrow -$ but a $+ \rightarrow +$ polarity may not become $+ \rightarrow -$. The default characteristics of a top level component are a single in port and out port of type a with $\alpha \rightarrow \beta$ polarity.

Transfer functions describe the computation an Infopipe component performs. Often the Infopipes language will be insufficient to define the transfer functions needed in real applications. Rather than give the Infopipes language the power of a language like C, the language includes a mechanism for calling library functions written in other languages. This ability requires the further assumption that library calls never block. These assumptions severely impact on the Infopipes implementation (see section 4.2).

2.3 Concrete Representation

This section illustrates the use of the language constructs introduced in section 2.1. The emphasis of these examples is the use of the language rather than the syntax. The concrete representation is secondary at this stage in the language development, but is useful in understanding the language concepts.

Figure 1 shows the definition of an n-way merge pipe. This component takes an array of numbers and outputs their sum. This example creates a new top

```
NWayMergePipe
    NWayMergePipe() {
        inport in::(Float[2], +)
        outport out::(Float, -)
    }

    NWayMergePipe(numInPorts::Int) {
        inport in::(Float[numInPorts],+)
        outport out::(Float,-)
    }

    transferFunction[in,out](items <- in) {
        sum(items) -> out
    }
```

Figure 1: Concrete Syntax Example - N-Way Merge Pipe

level component called NWayMergePipe. This component has two constructor functions: one that takes no parameters and one that takes an integer parameter. The former function defines an instance with an array of in ports of length two and a single out port. The latter defines an instance with a variable length in port array. The caller of the constructor provides the desired number of in ports. In both cases, the in ports are active while the out port is passive. NWayMergePipe also has a transfer function that describes the computation of the component. The dependency list of the transfer function shows that the transfer function interacts with the in port named *in* and the out port named *out.* A local variable called items retrieves a piece of data from the in port *in.* The transfer function calls the built-in array function *sum* and sends the result to the out port *out.*

Figure 2 contains an example of a multiplication filter. This example illustrates the use of configuration methods and multiple in ports which are not part of an array.

```
MultFilter
    MultFilter() {
        inport in1::(Int,a)
        inport in2::(Int,a)
        inport in3::(Int,a)
        outport out::(Int,b)
    }

    transferFunction[in1,in2,in3,out]
        (item1 <- in1, item2 <- in2, item3 <- in3) {
        (item1 * item2 * item3) -> out
    }

    config inPorts() {
        return ["in1", "in2", "in3"]
    }
```

Figure 2: Concrete Syntax Example - Multiplication Filter

The constructor function defines three in ports and a single outport. The component operates on integers, as indicated by the data type of the ports. The component has $\alpha \to \beta$ polarity. The transfer function's dependency list contains all of the ports defined by the constructor function. The local variables item1, item2, and item3 store integers from the first, second, and third in ports respectively. The transfer function sends the result of multiplying the input data to the out port. In addition to the standard constructor and transfer functions, the MultFilter component defines a configuration method called *inPorts.* This function returns an array containing the name of each of the component's in ports.

Figure 3 contains the code for a Pump component. This example demonstrates the use of control ports for feedback between components.

```
Pump
    rate::Float

    Pump(startRate::Float) {
        inport in::(a,+)
        optional inport c1::(Float,-)
        outport out::(a,+)
        rate = startRate
    }

    transferFunctioni[in,out](item <- in) {
        item -> out
    }

    transferFunction[c1](fillLevel <- c1) {
        if (fillLevel < 0.5) then -- increase rate --
        else if (fillLevel > 0.75) then -- decrease rate --
    }
```

Figure 3: Concrete Syntax Example - Pump

The definition of the Pump component includes the definition of a control variable called *rate*. The constructor function initializes this variable. The rest of the constructor function is quite similar to those seen in the previous examples except for the use of the keyword *optional*. The keyword *optional* indicates that the compiler should not generate an error if the port *c1* is not connected. The first transfer function in Figure 3 defines the standard data operation of Pumps. Pump uses a second transfer function to handle the operation of the optional port. The dependency list distinguishes the two functions.

3 Existing Implementation

The current implementation works with a configuration language for Infopipes. This configuration language contains a subset of the features required by the full-fledged Infopipes DSL. The language has mechanisms for instantiating, connecting, and defining components. These features, particularly component definition, have limited power.

The value of the implementation is the extensible framework it establishes, rather than the configuration language itself. The implementation of the configuration language contains two parts: a monadic embedded compiler written in

Haskell and a graphical simulator written in Smalltalk. The embedded compiler produces intermediate language code which the graphical simulator interprets.

The embedded compiler produces two different intermediate languages. Instantiation and connection compile to the input language of the graphical simulator. Component declaration translates to the Squeak file in format. The file in format contains valid Smalltalk wrapped with extra information that lets Squeak know what to do with the input. Squeak is the target environment so that the configuration language can take advantage of the graphical simulator and existing component class hierarchy. These Smalltalk elements make the DSL usable more immediately than if it were implemented from scratch.

This section details the inner workings of the embedded compiler and how it can be used in the development of a more powerful language for Infopipes.

3.1 Abstract Syntax

The configuration language consists only of the abstract syntax which is defined using Haskell's abstract data types. The abstract syntax provides all of the information needed to define the semantics of the language and is much more flexible than the concrete syntax. This flexibility is especially useful during the early development stages where the language is still evolving.

Figure 4 shows the abstract syntax for the configuration language.

```
data PipeTerm = InstantiateNoPos Var Var
                | InstantiateWithPos Var Var (Int, Int)
                | Connect Var Var Var Var
                | Filter Var [Exp]
                | Component Var Var [Method]
                | Redefine Var String [Exp]
data Var = PortVar String
           | ClassVar String
           | InstanceVar String
data Exp = Const Int
           | Add Exp Exp
           | Sub Exp Exp
           | Times Exp Exp
           | Return Exp
           | Assign String Exp
           | Variable String
type Program = [PipeTerm]
type Method = (String, [Exp])
```

Figure 4: Abstract Syntax For Infopipes Configuration Language

The data type PipeTerm defines the basic operations on Infopipe components: instantiation, connection, definition, and redefinition. The Var data type describes three kinds of variables: class, instance, and port. The compiler performs static checks that require a distinction between these variable types. Exp contains some standard arithmetic expressions as well as the commands for function return and variable assignment. Program and Method are type synonyms which provide more descriptive type signatures in the compiler.

PipeTerm is the most interesting data type in the DSL because it contains the top level Infopipe constructs. The constructor functions InstantiateWithPos and InstantiateNoPos represent instantiation with and without a specified display position for the component. Connect expresses the connection of two Infopipe instances. Filter and Component define new components. The first creates a new filter component and defines the computation of the new filter. Component is more general. It defines any kind of new component and allows for the definition of multiple methods in the new component. Finally, the Redefine constructor overrides the computation of an existing component with a specified behavior.

3.2 Static Checking

A key feature of the compiler is that it performs static checks during the compilation process. The instantiation and connection of components will only work properly if certain properties hold. All of these properties can be verified statically, yet the Smalltalk Infopipes implementation can provide no such checks. The embedded compiler adds these checks by utilizing the environments discussed in the next section (see section 3.3). The Maybe monad and an error string in the state provide feedback to the compiler driver when errors are found.

The first opportunity for static checking presents itself in the instantiation of components. Two common errors could occur in this situation. First, the user could attempt to instantiate a component that is undefined. Second, the user could give the instance a name that is already in scope.

Component connection has even more opportunities for errors to occur. Connection involves two ports, each of which is associated with an instance. The instances referenced could be undefined. The port name used may not be a valid port of the instance. Both of these issues generate an error. In addition, the first port must be an out port and the second port must be an in port. The compiler returns an error if the user supplies an incorrect port type.

Component definition generates errors if a component with the same name already exists or if the specified superclass does not exist. Redefinition causes failure if the component being overridden is undefined.

3.3 Environments

The static checks in the compiler require three environments to maintain variable information. Figure 5 shows the data types used in the implementation of the environments.

```
data PortType = IN
              | OUT
type PortEnv = [(String, (String, PortType))]
type ClassEnv = [String]
type ComponentEnv = [(String, String)]
type Env = (PortEnv, (ClassEnv, ComponentEnv))
```

Figure 5: Environment Data Types

The class environment (ClassEnv) stores the list of defined component classes. The default class environment contains the entire Smalltalk component hierarchy. When new components are defined the class name becomes part of the environment. The instance environment (ComponentEnv) stores the names of instances that are in scope and the associated class. Instantiation commands populate the initially empty instance environment. The port environment (PortEnv) contains the ports associated with each component. The type of the port (input or output) is stored as well. Component definition adds ports to the environment which initially contains all of the port information for the classes in the component hierarchy.

As a convenience, the Env type combines the three environments into one. The access and update functions for class, instance, and port environments are defined in terms of this master environment. During compilation, the State monad stores and maintains the master environment. As shown in the next section, the compiler makes extensive use of the environments for storing and checking information.

3.4 Compiling Programs

Each data type in the abstract syntax (except Var) has an associated compile method. The compile method transforms the abstract syntax into Squeak input and performs the static checking. A type class called Compilable links the compile methods together. The type class makes the langauge easily extensible. If the compiler writer adds a new expression type to the abstract syntax, the expression is automatically linked into the compiler by adding a compile method for the expression type. This section explains the details of the compile method for the top level PipeTerms.

The first kind of pipe term is instantiation. Figure 6 shows the code for compiling instantiation terms.

```
InstantiateNoPos (ClassVar c) (InstanceVar i) ->
    do (s, env, _) <- get
        case (lookupCE env c) of
        False ->
            do () <- set (s, env, (eInst c) ++ (cUDef c))
                return Nothing
        True -> case (lookupCompE env i) of
                    Nothing ->
                    let s' = s ++ "newPart " ++ c ++ " "
                                ++ i ++ " 0@0\n"
                    in do s'' <- set (s', addCompE env i c, "")
                            return (Just s')
                    _ -> do () <- set (s, env, (eInst c) ++ (iDef i))
                            return Nothing
InstantiateWithPos (ClassVar c) (InstanceVar i) (x, y) ->
    do (s, env, _) <- get
        case (lookupCE env c) of
            False ->
            do () <- set (s, env, (eInst c) ++ (cUDef c))
                return Nothing
            True -> case (lookupCompE env i) of
                    Nothing ->
                        let s' = s ++ "newPart " ++ c ++ " " ++ i ++ " "
                                ++ show x ++ "@" ++ show y ++ "\n"
                        in do s'' <- set (s', addCompE env i c, "")
                            return (Just s')
                    _ -> do () <- set (s, env, (eInst c) ++ (iDef i))
                            return Nothing
```

Figure 6: Compiling Instantiation Terms

There are two kinds of instantiation term: InstantiateNoPos and InstantiateWithPos. They operate in exactly the same manner except that the latter takes the (x, y) position on the screen where the instance will be displayed by the graphical simulator. The function obtains the output string and environment from the state. If the component being instantiated does not exist in the class environment, the function returns an error. Next, the compile function checks if the instance name is already in use in the current scope. If the instantiation term passes both of these checks the function updates the state with the Squeak input for instantiation appended to the output string. Instantiation terms also cause updates to the instance environment to reflect the addition of a new instance to the current scope.

Figure 7 shows the compile function for connection terms.

```
Connect (InstanceVar i1) (PortVar p1) (InstanceVar i2) (PortVar
p2) -> do (s, env, _) <- get
    case (lookupCompE env i1) of
        Nothing -> do () <- set (s, env, eConn ++ (iUDef i1))
                   return Nothing
        Just c1 -> case (lookupPE env c1 p1) of
                   Nothing ->
                   do () <- set (s, env, eConn ++ (pUDef c1 i1 p1))
                       return Nothing
                   Just IN ->
                   do () <- set (s, env, eConn ++ (pType1 i1 p1))
                       return Nothing
                   Just OUT ->
                   case (lookupCompE env i2) of
                       Nothing ->
                       do () <- set (s, env, eConn ++ (iUDef i2))
                           return Nothing
                       Just c2 ->
                       case (lookupPE env c2 p2) of
                           Nothing ->
                           do () <- set (s, env, eConn
                                       ++ (pUDef c2 i2 p2))
                               return Nothing
                           Just OUT ->
                           do () <- set (s, env, eConn
                                       ++ (pType2 i2 p2))
                               return Nothing
                           Just IN ->
                           let s' = s ++ "connect " ++ i1
                                       ++ " " ++ p1 ++ " "
                                       ++ i2 ++ " " ++ p2
                                       ++ "\n"
                           in do () <- set (s', env,"")
                                   return (Just s')
```

Figure 7: Compiling Connection Terms

The compiler performs three static checks on each of the two ports in the connection term. It checks if the instance being referenced is in scope, if the port is a valid port for the instance, and if the port is the correct type (input or output). Most of the code for compiling connection terms relates to these static checks.

The compiler stores appropriate error strings in the state when the static checks fail. If the term passes all of the checks the function generates the Squeak input for connection terms and stores it in the state.

The code in figures 8 and 9 describes the compilation function for component definition terms. The terms Filter and Component both describe the definition

```
Filter (ClassVar c) fun -> do (s, env, err) <- get
    let env1 = addCE env c
        env2 = addPE env1 c "input1" IN
        env3 = addPE env2 c "output1" OUT
        s1 = s ++ "\ndefineClass\n" ++ "Filter" ++ " subclass: #"
        ++ c ++ "\n\tinstanceVariableNames: \'\'"
    ++ "\n\tclassVariableNames: \'\'"
    ++ "\n\tpoolDictionaries: \'\'"
    ++ "\n\tcategory: \'RBK\'!\n"
                ++ "\ndefineMethod\n" ++ c
            ++ " methodsFor: \'initialization\'!\n"
            ++ "defaultFunction\n"
        in do () <- set (s1, env3,err)
                s2 <- compile fun
                case s2 of
                Nothing -> return Nothing
                Just s' -> do () <- set (s' ++ " ! !\n\n", env3, err)
                               return (Just (s' ++ " ! !\n\n"))
```

Figure 8: Compiling Filter Definition Terms

of new components. Filter is a common special case of Component where the user wishes to define a new filter component that inherits everything except the computational behavior of the existing component. The only arguments to Filter are the name for the new class and the computation the new component will perform. Component defines a new component that inherits from a specified existing component. A variable number of methods may be added/overriden.

The Component case checks if the specified superclass exists before performing compilation. Both definition methods add the new component and its associated ports to the environment. The compile function adds the file in format representation of a new class to the output string and recursively compiles the functions of the class.

```
Component (ClassVar c) (ClassVar sup) funs -> do (s, env, _) <-
get
    if (lookupCE env s) then
        do () <- set (s, env, eComp ++ (cDef c))
            return Nothing
    else
        let env1 = addCE env c
            env2 = addPE env1 c "input1" IN
            env3 = addPE env2 c "output1" OUT
            s1 = s ++ "\ndefineClass\n" ++ sup ++ " subclass: #" ++ c
                ++ "\n\tinstanceVariableNames: \'\'"
                ++ "\n\tclassVariableNames: \'\'"
                ++ "\n\tpoolDictionaries: \'\'"
                ++ "\n\tcategory: \'RBK\'!\n"
                    in do () <- set (s1, env3,"")
                            s2 <- compile (funs, c)
                            return s2
```

Figure 9: Compiling Component Definition Terms

Figure 10 contains the code for redefining the computational aspect of an existing filter component.

```
Redefine (ClassVar c) varName fun -> do (s, env, _) <- get
    if (lookupCE env s) then
        do s' <- compile fun
            case s' of
            Nothing -> return Nothing
            Just s'' -> let s1 = s ++ "[ : " ++ varName ++ " | "
                                    ++ s'' ++ " ] !"
                        in do () <- set (s1, env, "")
                                return (Just s1)
    else do () <- set (s, env, eComp ++ (cUDef c))
            return Nothing
```

Figure 10: Compiling Redefinition Terms

Redefine specifies a new computation for a filter at the instance level rather than at the class level. The arguments are the class whose computation should be modified, a local variable name, and the contents of the replacement function. If the class being redefined does not exist an error is returned. Otherwise the function recursively compiles the new computation and wraps the result in a Smalltalk block.

There are additional compile functions for the data types Exp, [Exp], (String, Method), and [(String, Method)]. These are called from the top level PipeTerm compile method. Their purpose is to compile the methods used in component definition and redefinition.

The compiler supplies a run function to invoke the compile function on a program. This function extracts the result of compilation from the State monad. If an error occurs during compilation, run prints the error at the command line. A successful compilation creates a file that can be input directly into Squeak.

4 Remaining Issues

Challenges remain in both the development of the Infopipes language and its implementation. This section describes the challenges that have been identified at this point in the design process.

4.1 Design Decisions

Language development is inherently an iterative process. Section 2 establishes the key feature set of the language, though this set is insufficient to describe the full range of systems that the Infopipes architecture could be applied to. This section enumerates potentially useful constructs that the current langauge description does not contain. Decisions regarding these features should be made after there is more experience using the language.

An accompanying library containing the basic data types and operations is essential to the usefulness of the Infopipes language. While the need for a library is evident, the contents of the proposed library are more nebulous. The needs of the system developer are the principle consideration in writing the library. A related issue involves the definition of data types. The system developer will inevitably need data types that the library does not provide. Currently there is no syntax for data type definition.

The Infopipes literature describes two key features that the current version of the language does not address. These features are composite components and the declaration of quality of service and environment requirements. Unlike some of the constructs proposed in this section, there is no question about the need for these features.

The nature of inheritance in the Infopipes language is incomplete. Currently the language assumes a standard object-oriented inheritance mechanism. There is potential for domain specific improvements to this approach. One potential improvement involves overriding methods from a supercomponent. In the current approach the user must completely redefine methods if they wish to add behavior. A pattern emerges where the user wants to extend the behavior of the supercomponent rather than completely redefining it. The langauge should take advantage of this pattern by providing a mechanism for method extension. This mechanism could be similar to the notion of backwards inheritance in Beta. Another pattern that emerges involves port declaration in constructor functions. If a subcomponent adds initialization behavior, it must also redefine its ports. It is rare that these port declarations vary from the supercomponent due to the limitations on overriding the data transmission characteristics of the supercomponent (see section 2.2). For this reason it may be useful to eliminate the port redefinition requirement.

Two issues arise in regards to the interaction of the new language with the existing Smalltalk implementation. The Smalltalk implementation includes a graphical simulator. There is disagreement about whether the graphical components used in the simulator should be wrappers for the underlying component implementation or whether the graphical components are in fact the implementation. A second issue is if it should be possible to declare new top level components. The Smalltalk component hierarchy contains a variety of shapes which may represent the full range of useful components. Only experience using the language will tell if the user needs to define top level components. If the set of top level components is fixed then some complexity in the component definition syntax can be removed.

4.2 Implementation Issues

The conversion of the configuration language into a complete implementation is not entirely straight forward. While a good framework exists, several problems remain unsolved.

There are two issues with the notion of transfer functions as defined in section 2.1.3, particularly in regards to the integration of the new language with the Smalltalk implementation. Only Filter components have transfer functions in the Smalltalk implementation. This contradicts the idea that all components use a transfer function to describe their computation. If the new language is to work with the Smalltalk implementation, the Smalltalk components should be refactored so that their mechanism for performing computation is consistent. The language definition also assumes that transfer functions interact directly with ports. In the Smalltalk implementation, the push and pull methods call the transfer function. Only the push and pull methods have access to ports. A decision must be made between the theory of transfer functions presented in this paper and the one that exists in the current Infopipes implementation.

Data and thread management present additional obstacles to implementing the Infopipes language. Ports require a mechanism for maintaining the order in which they receive data items. Access to control variables must be synchronized because components in different threads of control modify them. A decision needs to be made about who performs garbage collection when functions in other languages are called. Another issue is what thread of control the outside function operates in. Efficiency and the ability to reason about the system are the principal concerns in choosing data and thread management strategies.

5 Conclusions

This paper presents several aspects of the domain specific language for Infopipes. There is a limited implementation and a proposal for the constructs the new language should contain. The paper also outlines the desired capabilities of the DSL. Future directions for expansion of the language have been identified, along with the issues that surround the implementation of these features. Work remains to obtain a full featured implementation, but this paper provides a basis and direction for this future work.