

Serulian: Final Project Description

Project Name: Serulian

Team Members: Chao Cai (ccai@seas.upenn.edu), Joseph Schorr (jschorr@seas.upenn.edu)

Faculty Advisor: E Lewis

1 Abstract

Web application development is rapidly gaining popularity. However, the development process itself can be a slow and painful one. To write a web application, one must create a framework and presentation layer for the client side browser, a logic and computational layer for the server, a database querying layer for storage, and often an authentication and access layer for security. This is not counting all the glue that links each layer to all others. We see all this tedious work as a problem, and would like to propose a solution.

Our project, Serulian, seeks to address exactly the issues mentioned above. The goal of the project is to create an easily adoptable development framework for creating powerful web applications. The core mechanism for development in Serulian is BlueSource, a fully typed, object-oriented language with a syntax very similar to existing languages such as C, C++, C#, Java, and JavaScript. In addition, we have provided an XML markup schema for describing the presentation layout of the program on the client side. Finally, we are also including a core library of functionalities and controls as a basis for building larger programs.

2 General Overview

Over the last few years, we have seen web application development truly take off. There is an ever-growing interest in creating applications that are accessible wherever you are, through a conventional web browser. Unfortunately, the web application development process itself is still long and arduous. First, one must learn and use one or more languages to implement the client side, another set of languages for server interactions, and possibly more for database querying. In addition, one must also deal with the infrastructure that glues the server together with the client. Cross-browser compatibility and security are more issues that must be addressed. This is a lot of work, much of which is especially wasteful because, for example, the same protocols to marshal data between the client and the server are written over and over. Even where libraries are provided, the interface still must be learned and matched, despite how simple and repetitive some of this work is.

All web application development is divided into four key components: client side framework and layout, client side scripting, server side scripting, and database. Serulian is an effort to address the entire development system. Basically, it allows developers to write web applications with nearly the same simplicity they would expect from writing desktop applications. We created a new language with features that are designed specifically for dealing with such web behavior as querying and updating database tables, transporting data between the server and the client, and even structuring client-side presentation.

First, we've created an object-oriented, typed language framework that compiles down into JavaScript. This gives us the JavaScript benefit of being able to run on web browsers without requiring the user to download additional software, which means that the developer can reach more end users. However, it takes away the unwieldiness of developing in JavaScript, the chaos of its loose typing, and the difficulty of finding and fixing errors in an interpreted environment.

Second, we encapsulate and abstract much of the glue between client, server, and database. Using the concept of state, we are able to express work processes such as serialization, transfer, and de-serialization as one abstract notion, and minimize the impedance mismatch problem that arises from client-server-database interactions by unifying data representation. Using server functions, we can extend the server-client unity by providing clear, defined entry points between client and server in the form of function invocation, a concept familiar to most programmers. Using server classes, we wrap database tables and columns in an

object-oriented representation and allow code to deal with table data directly in code, rather than through indirect queries.

Third, we introduce a presentation layer compatible with major browsers, echoing the theme of reaching the largest user base possible. Its schema and style definitions will be reminiscent of such existing technologies as ASP and CSS, which allowing those who are already familiar with those products to easily transition into ours. It will include a wide set of controls out of the box which may easily be tied to written client code, as well as an extensible framework that gives flexibility toward construction of new controls.

3 Related Work

In the area of web application development, there are numerous existing technologies that address parts of what we are attempting to do:

- Client side content layout: HTML, DHTML, CSS
- Client side scripting: JavaScript/JScript/ECMAScript, VBScript
- Server side scripting: Perl, PHP
- Server side database access: Oracle/MySQL/PostgreSQL

Our project seeks to provide a simpler, more unified development process by allowing the user to specify the entire web application through just the use of BlueSource without having to worry about underlying client-server interactions that are standard to many web applications.

There has also been efforts to simplify the web application development process, most notably through the Microsoft's .NET (ASP.NET in particular) and Java's J2EE. Both these platforms attempt to address the primary issue plaguing web application developers, that of a divide between the server and client side architectures. Currently, and without these platforms, any web application in development must have each half coded separately in different languages and frameworks, resulting in a large amount of tedium when trying to get them to work together in harmony. Each platform addresses this issue by reducing the work down to coding for the server side, with client side scripting either being generated automatically (in the basic case of J2EE) or having the language used for both client and server be one and the same (in the case of ASP.NET using JScript or VBScript). Ultimately, however, each of these platforms much work within the limitations imposed by the divide, as they merely give the illusion of bridging it. In addition, and perhaps most important, these platforms, by trying to bridge the divide fall the "one-size-fits-all" dilemma, where they can handle server side and client side well, but neither extraordinarily (in general, they work much better on the server side, rather than the client side as they have a larger selection of languages to choose from.)

Within the academic community there has also been investigations into the architecture of web-based applications², as well as potential productivity gains from integrating several facets of the application environment³. However, most of this work remains either theoretical or "toy"-ish in nature, without a true infrastructure for large application development.

In terms of specific concepts we are introducing, we are able to draw analogies to previous work. For example, the concept of state is much akin to serialization, where every object is transformed into a form for ease of mobility over the network and between logically separate executable entities. We simply took this a step further and introduced the idea of snapshots and entrypoints which, as we will later describe, root state in as a core part of the language.

Server functions on the other hand may remind one of remote procedure calls, which allow for a client to remotely delegate execution to a server, which then returns the requested data when done. We simply once

²Deriving Architectures of Web-Based Applications, by Weiquan Zhao et al.

³A Combined Runtime Environment and Web-Based Development Environment for Web Application Engineering, by Martijn van Berkum et al.

again have built them directly into the language so that with the use of just one keyword, the function may be marked to execute on the server instead of the client.

The querying capabilities of server class have pre-existing roots in The LINQ (Language Integrated Query) Project, which focuses heavily on merging SQL-like syntax with conventional code so that database queries may be more effectively integrated into the language. Here, we adopt a similar idea, mapping classes to tables and class instances to columns, allowing for specification of table columns with special class variables, and placing select-from-where structures directly into code expressions.

We feel that our main point of innovation is in integration of all of these seemingly separate concepts into one overarching development framework. Had we implemented each on its own and branded everything as one product, we would not have achieved much at all. Instead, we are combining them to realize the potentials of a greater system. For example, with the concept of state, we can marshal and unmarshal arguments and return values by having the sender simply encode data into state, and having the receiver decode state back into data. Furthermore, just as the original creators of the first compilers realized that coding in assembly was too tedious and abstract constructs such as loops and conditionals may ease the development process, we are now realizing that some existing technologies like JavaScript are becoming the new "low-level", where abstract concepts such as state and server classes may hide a lot of the tedium so that developers may focus on the next steps in creativity.

4 Language Features

Here, we will assume familiarity with most aspects of object-oriented programming, and only describe in detail those language features that deviate significantly from existing practice.

4.1 The Basics

Our benchmark comparison language will be C#. In C#, each class creates a new derived type. This is no different here, except that we took it a step further and corresponded each primitive type (such as int and string) to a class as well. Each class may have class variables and functions, and in addition may also have more complex constructs such as properties, operator overloads, indexers, and enumerations. Modifiers such as public, private, and protected have the same meaning here as they do in other languages. Class elements may be declared static, although some constructs such as enumerations are inherently static and hence do not need to explicitly be defined as such.

BlueSource adopts single inheritance, but each class may implement multiple interfaces. The type inheritance tree is rooted at BlueCore.Primitives.Object, and reaches every class in the system. Subtype polymorphism is implemented essentially the same way as in C#, where subtypes may be placed where supertypes are expected. Furthermore, types may have generics similar to Java 5 and C# 2, allowing for the creation of generic container classes such as List.

Within functions, each logical unit of execution is a statement. Statements include local variable declarations, expressions, and control flow structures. Within statements, often expressions are expected to deal with basic units of computation. Expressions may refer to variables on various naming scopes, literals, or be a combination of operators and operands to describe complex operations.

With respect to naming scopes, on the highest level BlueSource uses a namespacing system similar to that of C#. Every class must be declared within a namespace, and every class may be globally accessible from anywhere by their fully namespace-qualified name. On the next naming scope level, if class A and class B are declared within the same namespace, then B may refer to A without attaching the namespace prefix. The third level is that of class elements, so any code within any class element may refer to other class elements on the same class by their name. Fourth, within functions, function parameters and local variables declared within current and parent scopes may be referred. Local naming scopes may be created with curly braces, so that arbitrarily deeply nested scopes may be created.

For an example, see Appendix 8.1

4.2 State

Our first major deviation from the conventional modern-day object-oriented language is the concept of state. The fundamental idea here is that every object, in addition to its native data representation, has an associated XML representation that we call its state. By default, there is a representation where all member variables are recursively stated, so that upon conversion back from XML into data, we are guaranteed that all needed information may be restored. However, if a developer knows that only certain information is required to save and restore the object's state, the developer may use a class element named **state** to write a custom specification. The **state** construct is a special property containing two portions: a **snapshot** block that described how to transform data to XML, and a **entrypoint** block for going from XML back into data.

A major source of tedium when coding web applications is the marshaling and unmarshaling of data when data is sent over the network. State allows for a standard serialized format applicable to all objects, so that dealing with such network protocols are no longer a first priority but are rather only an advanced optimization. Even any work left to optimize information storage is simplified, as each class can rely on the fact that all other classes are properly stated to only worry about direct information it is holding, rather

than handling its member variable's state definitions directly.

A second major concern is state preservation of web applications, which is why we originally arrived at the notion of state. Here, we take an analogy to desktop applications where, at the worst, a snapshot of the process's local memory may allow for somewhat inefficient saving and restoration. The default XML state of an application is the "local memory" in the web context, so we can save that onto the server, attributing the state to some user if authenticated, or a cookie reference if unauthenticated. In a desktop saving, most applications have much more efficient saving mechanisms, partially because it is easy to specify which pieces of data to save and restore. The **snapshot** and **entrypoint** blocks provide the same ease, so that in a web context the developer may also simply specify what to preserve.

For an example, see Appendix 8.2

4.3 Server Functions

Server functions allow any function to be marked to execute on the server rather than the client. These are important if there are sensitive operations, for example authentication and database access, that should be handled on the server. Also, if there are frequent database accesses within a code section, then executing that code on the server will reduce the total overhead of transporting data back and forth between the server and the client. Without server functions, a developer would have to learn a server-side language, write the code that will handle the server-side functionalities without many of the features found in BlueSource, and interface with BlueSource's state system on the raw XML level. With server functions, the developer only needs to add the **server** modifier in front of a function, and that code will at runtime execute on the server. The network exchange between client and server will happen upon dispatch and return of the server function, and to the calling function, there will be no apparent difference in calling a client function or a server function. All of the network protocol handling will be done implicitly, so that the developer will not have to worry about any of it.

For an example, see Appendix 8.3

4.4 Server Classes

We next looked into directly interfacing with databases through BlueSource. Here, the two major concerns are creating and updating tables, and querying them. Aiming for an object-oriented data abstraction to tables, we introduce server classes. When a class is marked with **server**, a table for it is created in the server-side database. In addition to normal class variables, fields may be created that act like class variables but directly correspond to columns in the database. Each instance of the class, on the other hand, map to a row in the database. When a new instance of the server class is created, a new row is inserted into the database, and modifying the fields on that instance will result in database updates on the corresponding columns of that row.

For querying database tables, we adopted the LINQ approach of embedding SQL-like syntax directly into the language. As of now, the SQL expression is only valid on the initializer of a local variable declaration. The queries are in the "FROM *TableName* SELECT * WHERE *conditions*" form, evaluating to a List containing objects of the type corresponding to the queried table. We plan to move to far more complex queries in the future, but this groundwork demonstrates the capabilities of BlueSource to directly integrate table data into its flow of execution.

For an example, see Appendix 8.4

4.5 SIML

The Serulian Interface Markup Language (SIML) is an XML format closely integrated with BlueSource to describe the client-side visual layout of the application. Technically, we could have allowed BlueSource to

directly manipulate the browser's Document Object Model tree, and that option is still available since we are wrapping JavaScript libraries to work with BlueSource. However, as one of our primary goals is ease of development, we sought a more clean and concise format that better conveyed visual information.

Just as every object has an associated string representation in Java and C#, in a web programming context it would also be convenient for them to each have an associated layout to be easily embedded directly into the application's visual layer. By default, an object's associated SIML representation is as simple as the object's class name in a string formatted for browser display. However, by creating a template for a class, the developer may define in XML how the class will appear in the browser. For example, if the class is a calculator, then the developer may specify where the display digits (likely a numeric member variable) will be positioned, where each button (likely stored as a List of Button class objects) will appear, and so on. Note that the buttons may then recursively look into their templates to render themselves.

For an example, see Appendix 8.5

5 Technical Approach

We will now discuss how various aspects of our projects were implemented.

5.1 Lexing and Parsing

We used the ANTLR compiler construction framework to create a C# lexer and parser by describing the structure in a grammar file. Our lexer and parser are LL(k), with a lookahead of 4 for the lexer and a lookahead of 3 for the parser. The result of this stage was an AST (named BlueAST) that closely mirrored the original parse tree. Comments and whitespace are ignored at the lexer level, while line numbering and file information is collected at the parser level. In a few locations, due to the complexity of our language, we were forced to employ syntactic predicates, which allow for potentially inefficient backtracking to simulate "infinite lookahead". However, we minimized the usage of such backtracking to reduce the associated performance penalty. Looking forward, we may be able to improve here by converting to LALR parsing in an attempt to remove the necessary backtracking and reduce grammatical ambiguity.

5.2 Scoping

Here we'll discuss how we constructed an intermediate representation suited for the later stages of typechecking and compilation. The term "scoping" is used to represent the building of names for each logical object within the abstract syntax tree. For example, items such as classes, functions, properties, and variables must be scoped, to define how their names are referenced later in code.

5.2.1 Annotated AST Generation (BuilderObjects)

We next traversed the BlueAST structure in C# to create an annotated abstract syntax tree representation dubbed BuilderObjects. To describe various sets of AST node types, we created an interface describing the behavior expected from each set. For example, anything implemented the IFunction interface would have the properties of a function, including a name, a parameter list, and a return type, whereas anything implementing the IExpression interface may be placed wherever an expression is expected.

Ideally, we would prefer multiple inheritance since many of these behaviors are exactly the same for the majority of expressions. However, we were able to simulate multiple inheritance by having tree nodes derive from a base class named BuilderObject, and defining the behavior there. As an extra safety net, for an interface-specific function or property, we checked through reflection that the current object does implement the expected interface before allowing the code body to execute. The C# "is" keyword was very convenient in this respect.

5.2.2 Static Type Information (BlueType)

As we generate the AST, we also create objects that store relevant type information to be used in later stages. The class BlueType encapsulates such information as the name of the type, a reference to its base class, and additional information such as the implemented interfaces, type parent, and any generic type parameters. However, because we attempted to generate as much relevant naming, typing, and scoping information in one pass, we run into situations where referred types may still be in raw parse tree form and hence we currently have no information on such types. Hence, we delay the population of most type information to a single second pass where resolution of the types attached to typed items are done.

However, because types may refer to other types because of parent relationships, generic relationships, and so on, some type objects are in fact not resolved when they are first accessed and are forced to resolve at that time. This opens up the potential problem of cyclic resolutions, where types are trying to resolve based on information from each other. We have worked around this problem in the current implementation by marking types as resolved before such cyclic dependencies may occur, but a major future improvement here will be to break down type resolution into multiple passes, so that only the class and class element level information are fully collected in the first typing pass. Subsequent expressions and statements would then

have full access to this information and will no longer need to defer type resolution as we do right now. This hierarchical approach to type resolution will clear out the cycle problems mentioned above.

5.2.3 Namespacing Information (NamespaceRegister)

When a class is created, it may be declared inside any arbitrary namespace. Hence, we need a global namespacing system such that each namespace level contains classes and sub-namespaces on that level. We use namespace registers for this purpose: the Global Namespace Register (GNR) contains references to top level namespace registers, and each of those contain their own classes and namespaces. Namespaces may be nested arbitrarily deep in this fashion, but no two namespace elements (interfaces, classes, sub-namespaces) may share the same name. For example, if `BlueCore.Primitives.Integer` were the only class in the system, then the GNR would contain one entry for the namespace `BlueCore`, `BlueCore` would contain one entry for the namespace `Primitives`, and `Primitives` would contain one entry for the class `Integer`.

5.3 Static Typechecking

Once the nodes of the abstract syntax tree have been properly scoped and typed, it's time to verify that the code is actually in working order. Type mismatches can frequently occur, especially in transitioning from an untyped language such as JavaScript to a typed language like BlueSource.

Typechecking is performed in a recursive-descent fashion, from the program level down. In general, each `BuilderObject` recursively typechecks its children before typechecking itself. On the class and interface levels, naming and inheritance rules are verified. On the statement level, return type conformance and such rules as no break statements outside loops and switches are applied. The majority of operational type conformance and resolution are determined on the expression level, where type relationships apply almost everywhere. At any time, if a type error occurs, the compiler will halt and report the error with as much relevant information as possible.

5.4 Compilation

Once the code has been determined to be statically type-safe, we can then move into compilation. Once again, the strategy is recursive descent. In the current implementation, we simply pass around a `CodeBuilder` object which accumulates the string to be outputted to the resulting output file. Many of the simpler constructs are directly translated into their JavaScript equivalent, while more complex elements require extra code to be inserted. For expressions whose type must be verified at runtime such as cast expressions, a runtime type system is maintained.

For each class, a prototype of the class with all its functions, variables, and other class elements are created. JavaScript allows for functions to be declared and stored into variables, so that we may represent all aspects of a class with a single object and refer to that object when we need to access that class's members. Each class is saved into its own source file, loaded only when necessary. Furthermore, a scope file (extension `.bsco`) is also created, containing only class level and class element level data in the form of XML. Think of the scope file as the BlueSource equivalent of an automatically generated header: this is how BlueSource development can be componentized. The scopes act as the interface, hiding the underlying implementation details.

Another major improvement may be applied to this aspect of the compiler. While code accumulation is fine for simpler languages, for a language of our complexity it is becoming growingly cumbersome. We are considering, as a next step, designing a linear intermediate language representation that translates down to only simple JavaScript constructs of loops, conditionals, variable declarations, and expressions. BlueSource will first map to this internal representation, which then turns into JavaScript, our "assembly".

5.5 Just-In-Time Dynamic Loading

Code retrieval takes up a lot of the load time of a web application. To relieve this burden, we implemented Just-In-Time Dynamic Loading (JITDL) available for all classes. Essentially, when the application first starts up, only those classes marked with a Preloaded attribute are directly fetched. The rest of the classes are not immediately loaded, but rather a class shell is created in its place. When the class is first accessed, a round-trip to the server replaces the class shell with the actual class definition. This reduces initial load time by a potentially enormous amount, since otherwise classes may refer to other classes and create a chain of load dependencies.

5.6 State

Despite the novelty of having a language-level correspondence between data and XML, state was not difficult to implement. Once snapshots and entrypoints are parsed and compiled into JavaScript, they themselves are as XML generating and parsing code blocks, acting as the conversion function between XML and whatever BlueSource object they are attached to.

To take the XML-data interchangeability idea even further, we allow for XML to be declared in code, which are recognized as literals of type XmlNode. We also introduce a "create" function that takes in a type on its generic parameter and a block of XML as its argument, and attempt to apply the entrypoint of the specified type to the XML to instantiate a new data object. This is simply an application of state, and other than syntax parsing, was mostly achieved with just BlueSource code library additions.

We attempted to preserve the notion of reference equality with state. A unique hash is associated with each object in the runtime system, which is stored onto the state of an object. There is a global reference table mapping hash IDs to existing copies of state, so that no state XML is generated twice. When restoring XML back to data, this is also helpful in identifying where two references should point to the same object, so that reference equality comparisons and dependencies are not distorted because of stating and unstating.

5.7 Server Functions

State in fact takes care of a very important aspect of server function implementation: marshalling and unmarshalling of data on the two sides of the network. When a server function is invoked, the arguments are stated and sent to the server, the server unstates the arguments, executes the function, and sends the return value back to the client after stating it. The client then unstates the return value and continues on executing as if the server call was a local call.

Originally, we had envision server functions to be a predefined set of common library calls written in Perl that BlueSource code may invoke natively, similar to raising system calls. However, we wanted to give users the flexibility to specify what they want executed on the server, as well as the simplicity of applying the BlueSource they already know to the server. Hence, we employed a technology named SpiderMonkey that allowed Perl to execute JavaScript, so that nearly all code written may be executed on either the client or the server. Hence, if a server function needs to make calls to other functions in the program, there is a server copy ready to execute.

The only exceptions are client-specific functions such as presentation manipulation (JavaScript alerts are a good example) that must be done on the client, and these may be marked with the "client" keyword. If a server function, in its course of execution, calls a client function, then an exception will be thrown. In the future, we hope to statically build the call graph and check for execution dependencies, complaining before the compilation stage if there is a server to client dependency anywhere in the system.

5.8 Server Classes

Our next step is interfacing with the database. As a reminder, we are binding classes to tables, fields to columns, and instances to rows. In addition to the columns determined by class fields, each row also has an

implicit reference ID so that they may be uniquely identified and retrieved quickly.

Table creation and updating are done through wrapper calls in Perl and SQL to the database. Tables are created when the application is first deployed, since all relevant tables are statically determined. Row creation occurs on the instantiation of a new object, and involve simply a call to a predefined server function that takes care of SQL-side transactions. Updating are handled in a similiar fashion. Querying involves translating the BlueSource querying expression syntax to the actual SQL syntax, and we have created a translator to MySQL for now. In the future, we plan to make this effectively database-platform-independent by providing translators to major database querying languages.

Handling query results is a far more interesting problem. Technically, a very inefficiency but correct approach would be to simply retrieve the entire returned rowset and use it on the server or state it for the client. However, we thought of scenarios of hundreds of thousands of rows, and wanted Serulian to be able to scale well even against such massive databases. Hence, when queries are invoked, we only return the first x rows, betting on the common case that the results will be accessed in order. When a row outside the selected range is requested, we preserve the last set of rows selected, and go to the server to fetch a range of x rows that includes the requested row, exploiting the spatial locality condition that nearby rows may be requested next. In a way, this is very similar to hardware caching, and we can derive some interesting techniques for which rows to choose. As of now, x is a fixed constant defaulted at a reasonable number like 20, but the developer may override that constant to allow for smaller or larger blocks of rows to be fetched at once. In the future, we are considering application of machine learning techniques to adjust parameters such as x to optimal values based on runtime experience.

In this first iteration of Serulian, because of time constraints, we sacrificed efficiency for correctness in choosing to update synchronously. While synchronous updating may sometimes result in long waits for round-trip results, we are guaranteed data integrity within the database. Currently, to alleviate this problem, we introduced update blocks whereby a developer may defer all database updates until the end of the block. Looking forward, a safe approach for statement-by-statement updates would be to timestamp all rows by the time-of-last-update. If a request to modify a row has been issued by the running program, then we will first fetch its corresponding row in the server. If the time-of-last-update is later than the time of fetch of that row by the running program, then a concurrency exception would be thrown. The default handling of this exception, at the top level, would be to alert the user that the last write could not be done because the underlying data had changed, and request that the user reevaluate the data given the new value in the database.

5.9 SIML

At its very core, the implementation of SIML is simply manipulation of the browser's Document Object Model. Each object is associated with a template, and at runtime, the top presentation level object (usually the Application class) is inserted into the DOM and begins rendering its own template and calling its children's render functions. However, SIML is more than just manipulation of standard HTML elements, for that would eliminate the possibility of colorful and stylistic web applications that technologies like Flash offer. With the introduction of Scalable Vector Graphics (SVG), drawing and rendering shapes in the browser is becoming easier and easier. Taking a forward-looking approach, we adopted SVG to render native Serulian graphical objects such as rectangles and circles, which then build up to more complex layouts. This allows us to scale our graphics to fit regardless of the browser window size, a huge benefit over rasterized images.

While SVG works on the latest Firefox and Safari browsers, a major downside is that it is not currently supported on Internet Explorer without a plug-in. However, Microsoft (as usual) already has their own version of vector graphics built into Internet Explorer named Vector Markup Language (VML). Hence, we take a dual compilation strategy: both SVG and VML rendering code is generated from the SIML files. When the client browser requests the layout of the page, the proper code is returned depending on the requesting browser type.

However, to maintain our focus on application reach and hence cross-browser compatibility, we are considering the possibility of rendering graphics on the server and sending them to the client as images in the future. This will be an all-else-fails plan that will be applied less and less until the old browsers are all but retired.

5.10 Other Features

Here we will discuss implementation of features that may not necessarily be crucial to the core functioning of Serulian, but are nevertheless extremely convenient and helpful once implemented.

5.10.1 Generics

Type parameterization is in fact one of the most complex features implemented in BlueSource. It allows for simplification of code in a lot of instances, the most canonical being generic collections such as Lists. To implement generics in static typechecking, rather than cloning the class like many other compilers, we decided to directly pass the generic information along in the type and scope objects that are associated with relevant BuilderObject nodes. This is far less memory-intensive, but require much more intricate information-passing. Especially tricky are expression level function invocations and member accessing where generic instances on variable declarations must be preserved, and inheritance chains with generic parents where the generics of each class must be satisfied with the generic instances of their direct children. The manipulation of these hashtables were originally scattered throughout the BuilderObject tree, but a recent refactoring has allowed us to encapsulate them into the ScopeInstance object, which takes care of typing information and member resolution.

5.10.2 Iterators

Taking a page from C#'s implicit iterators, we created a similar **iterator** construct as a class element that is, while not essential to Serulian, nevertheless a very convenient feature to have.

Iterators within object oriented languages are generally one of the most used, yet least understood and therefore, least implemented, features. In languages such as Java and version 1.0 of C#, to properly implement an iterator, one would have to create a new class for each existing class, derive it from a predefined interface, and have it return the items in the proper order using normal coding rules. Although this appears at first to be relatively easy to do, in practice, for the more complex data structures, this can be quite tedious. Thus, we felt that having the ability to implement iterators "natively" would be of great benefit. The usage of iterators is as follows: Within any given iterator construct, the contents of the primary block are the same as any function with a few important differences. First, the keyword **yield return**, is used in place of **return**. This statement tells the iterator to return the specified expression when the iterator reaches that statement. In addition, a **yield in** statement allows the developer to return not a single expression, but in fact the contents of another iterator, given by the expression to the statement. Finally, the **yield break** statement indicates that there are no more items remaining to iterate and that the calling **foreach** or **yield in** should terminate. Multiple **yield returns** may be used to return items, and they are returned in the order in which the **yield return** statements are encountered. This continues until the end of the code block is reached, or a **yield break** is encountered.

For an example, see Appendix 8.6

The implementation of iterators in Serulian is a bit more complex than it is in C#. Normally, implicit iterators are implemented by simply generating the code into a state machine of sorts, using gotos to continue where the iteration code last left off. Unfortunately, the goto statement is not supported in JavaScript, and as such, we had to improvise a way around this fact. This simulation is done as follows: A state machine is constructed as in C#, however, each statement that would be compiled into its native form is instead simulated by a set of additional states. A prime example is that of a while loop. In this implementation, a while loop is transformed in a state which checks the condition, a state that executes if that condition is true and a state that jumps after the loop if the condition is false. By having the state machine jump to the

"check" state at the conclusion of each while loop iteration, we can simulate a while loop without actually creating one. This method is used for all other statements, save **try-catch**, which is not allowed to contain a **yield** statement, and thus, can be compiled normally.

6 Conclusion

In conclusion, we have developed a framework for web application development that aims to be both easy to adopt and powerful to use. We have completed the originally planned goals for the semester and added on many unanticipated language features along the way, such as function level generics, iterators, and even the entire concept of server functions. Looking back, given the time constraints we were under and the other academic responsibilities we had, we are content with our progress to date. Looking back, we should have planned ahead even more for compiler extensibility than we had, to avoid costly refactoring later on. However, we planned enough and introduced enough modularity such that even when the inevitable refactoring occurred on the scoping stage, few things from other stages really needed modification. Other than that, we just wish we had even more time to work on other features, such as more extensive server class queries.

Ultimately, what we made is a fully functional compiler that turned an object-oriented language into a full web application, spanning the entire client-server-database infrastructure. We will be expanding this language and its corresponding compiler beyond the scope of what was done this year, all while using it to code our own web applications from now on.

7 References

- Martijn van Berkum, Sjaak Brinkkemper, and Arthur Meyer, "A Combined Runtime Environment and Web-Based Development Environment for Web Application Engineering", Lecture Notes in Computer Science, Volume 3084, Jan 2004, Pages 307 - 321
- ECMA, "ECMAScript Language Specification", 1999,
<http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf>
- Rajendra Kumar Komandur, "C# Grammar File", 2003,
<http://wwwantlr.org/grammar/1067486869200/csharp.g>
- Terence Parr, "ANTLR Reference Manual", 2005,
<http://wwwantlr.org/doc/index.html>
- W3C, "Scalable Vector Graphics 1.1 Specification", 2003,
<http://www.w3.org/TR/SVG11/>
- W3C, "Vector Markup Language", 1998,
<http://www.w3.org/TR/NOTE-VML>
- Weiquan Zhao, David Kearney, "Deriving Architectures of Web-Based Applications", Lecture Notes in Computer Science, Volume 2642, Jan 2003, Pages 301 - 312

8 Appendix: Code Samples

8.1 A Basic Serulian Application

This example defines a Square class, declares a new square, and displays its computed area with a message box.

```
using BlueCore;
using BlueCore.Primitives;
using BlueObjects;
using BlueObjects.JavaScript;
using BlueObjects.Collections;
using BlueObjects.Exceptions;
using BlueObjects.Xml;

namespace ShapeTester {
    class ShapeTester : IApplication {
        public function<void> Main() {
            var<Square> sq=new Square(10,10,20);

            Alert(sq.Area().toString());
        }
    }

    class Square {

        public var<int> m_Length=0;
        public var<int> m_X=0;
        public var<int> m_Y=0;

        public Square(int x, int y, int length) {
            m_Length=length;
            m_X=x;
            m_Y=y;
        }

        public function<string> toString() {
            return('Square: ' + this.Area().toString());
        }

        public function<int> Area() {
            return(m_Length*m_Length);
        }
    }
}
```

8.2 A Serulian Application, With State Block Added

This example expands on the previous one, with a custom XML state definition included.

```
using BlueCore;
using BlueCore.Primitives;
using BlueObjects;
using BlueObjects.JavaScript;
using BlueObjects.Collections;
using BlueObjects.Exceptions;
using BlueObjects.Xml;

namespace ShapeTester {
    class ShapeTester : IApplication {
        public function<void> Main() {
            var<Square> sq=new Square(10,10,20);

            Alert(sq.Area().toString());
        }
    }

    class Square {

        public var<int> m_Length=0;
        public var<int> m_X=0;
        public var<int> m_Y=0;

        public Square(int x, int y, int length) {
            m_Length=length;
            m_X=x;
            m_Y=y;
        }

        public function<string> toString() {
            return('Square: ' + this.Area().toString());
        }

        public function<int> Area() {
            return(m_Length*m_Length);
        }

        state {
            snapshot {
                <Length>
                {
                    yield m_Length;
                }
                </Length>
                <X>
                {
                    yield m_X;
                }
                </X>
                <Y>
                {
```



```

class Square {

    public var<int> m_Length=0;
    public var<int> m_X=0;
    public var<int> m_Y=0;

    public Square(int x, int y, int length) {
        m_Length=length;
        m_X=x;
        m_Y=y;
    }

    public function<string> toString() {
        return('Square: ' + this.Area().toString());
    }

    public function<int> Area() {
        return(m_Length*m_Length);
    }
}
}

```

8.4 A Serulian Application Utilizing a Server Class and Query

Here, we use a server function GetXs to query from the database a list of x coordinates from all Squares stored in the database.

```

using BlueCore;
using BlueCore.Primitives;
using BlueObjects;
using BlueObjects.JavaScript;
using BlueObjects.Collections;
using BlueObjects.Exceptions;
using BlueObjects.Xml;
using BlueObjects.Server;

namespace ShapeTester {
    server class MyServerClass {
        public field<Square> Square;
        public field<int> Length;

        public MyServerClass(Square sq) {
            Square=sq;
            Length=sq.m_Length;
        }
    }

    class ShapeTester : IApplication {
        public function<void> Main() {
            //Create an instance of the server class (this will add a new row
            //    every time the app is called)
            var<MyServerClass> msc=new MyServerClass(new Square(10,10,50));

```

```

        //Get the x list
        List<int> xList=this.GetXs();

        foreach(int x in xList) {
            Alert(x.toString());
        }
    }

    public server function<List<int>> GetXs() {
        var<QueryResults<Square>> results <- from MyServerClass
            select Square where Length=50;

        var<List<int>> xList=new List<int>();

        foreach(Square s in results) {
            xList.Add(s.m_X);
        }

        return(xList);
    }
}

class Square {
    public var<int> m_Length=0;
    public var<int> m_X=0;
    public var<int> m_Y=0;

    public Square(int x, int y, int length) {
        m_Length=length;
        m_X=x;
        m_Y=y;
    }

    public function<string> toString() {
        return('Square: ' + this.Area().toString());
    }

    public function<int> Area() {
        return(m_Length*m_Length);
    }
}
}

```

8.5 SIML Example

This is a very simple SIML example that creates a click button template to be applied to the class ShapeTester.

```
<?xml version="1.0" encoding="UTF-8"?>
<s:Template AppliedTo="ShapeTester.ShapeTester" xmlns="http://www.w3.org/1999/xhtml"
    xml:lang="en" lang="en" xmlns:s="urn:serulian:BlueObjects.SIML">
<b>This is the HTML and SIML code that will be displayed on screen for the ShapeTester class</b>
<s:Button>
<font color="red">Click Me!</font>
</s:Button>
</s:Template>
```

8.6 A class Implementing an Implicit Iterator

The below example implements an implicit iterator for the points of the square in the Square class. In the main ShapeTester class, we call this iterator in the foreach loop and yield the single point in the square if it has a length of 0, and all four points otherwise.

```
using BlueCore;
using BlueCore.Primitives;
using BlueObjects;
using BlueObjects.JavaScript;
using BlueObjects.Collections;
using BlueObjects.Exceptions;
using BlueObjects.Xml;

namespace ShapeTester {
    class ShapeTester : IApplication {
        public function<void> Main() {
            var<Square> sq=new Square(10,10,20);

            foreach(Point p in sq) {
                Alert(p.m_X.toString() + " " + p.m_Y.toString());
            }
        }
    }

    class Square : IEnumerable<int> {
        private var<int> m_Length=0;
        private var<int> m_X=0;
        private var<int> m_Y=0;

        public Square(int x, int y, int length) {
            m_Length=length;
            m_X=x;
            m_Y=y;
        }

        public iterator<Point> Enumerator {
            yield return new Point(m_X, m_Y);

            if (m_Length > 0) {
                yield return new Point(m_X + m_Length, m_Y);
            }
        }
    }
}
```

```
        yield return new Point(m_X, m_Y + m_Length);
        yield return new Point(m_X + m_Length, m_Y + m_Length);
    }
}

class Point {
    public var<int> m_X=0;
    public var<int> m_Y=0;

    public Point(int x, int y) {
        m_X = x;
        m_Y = y;
    }
}
}
```