OnRamp: Enabling a New Component-Based Development Paradigm

Geoffrey C. Hulette, Matthew J. Sottile
Department of Computer Science
120 Deschutes Hall
University of Oregon
Eugene, OR 97403-1202
{ghulette,matt}@cs.uoregon.edu

Robert Armstrong, Benjamin Allan
High Performance Computing & Networking
Sandia National Laboratories
7011 East Ave
Livermore, CA 94551
{rob,baallan}@sandia.gov

ABSTRACT

Often the adoption of component-based scientific software requires the developer to abandon comfortable practices and embrace an unfamiliar software methodology. OnRamp provides a mechanism for the developer to generate CCA components, through commented markup in their original software, and keep their familiar software development practices. The developer uses these annotations to identify which methods in their code belong to which Port interfaces, and which Ports belong to which CCA components. Taken by itself, the markup is sufficient to create a skeleton of components representing the exported functionality of the original code, but because the entire code is available to OnRamp the implementation can also be generated. The functionality of the original code is wrapped in annotation-specified components, exporting part or all of its original functionality. OnRamp provides a model for component software engineering that allows developers to improve their code with their familiar software and in their established software practices and also interoperate with external developers easily when the need arises.

1. INTRODUCTION

While component-based software engineering (CBSE) is an important tool for managing software complexity, software frameworks themselves introduce a learning curve that is a barrier to adoption. Particularly in scientific simulation, software engineering is almost never a priority and the onus of learning a component environment and bending an established development practice to a new system increases resistance to adoption. OnRamp introduces a paradigm for CBSE that allows scientists to keep their familiar development practices and add an automatically generated wrapper of their code that is component based yet automatically tracks changes in the core software. In this way the familiar code and practices can be maintained and improved and, when interoperability issues arise in a collaborative environment, a means to participate through the use of interoperable components is also available. Presumably, if collaborative or modular codes are attractive to the developer, eventually the old practices will be abandoned in favor of a CBSE style of development. OnRamp allows for a smooth transition without requiring an "all or nothing" adoption of CBSE.

This approach is particularly attractive when several scientific simulation developers are brought together to create a multi-physics or multi-scale code that combines the best of each. In this use case each of the codes have been in development for years, each in their own programming language and each with their own disparate programming and build practices. Further, while the codes may use conceptually similar structures (such as meshes or particles), the representation chosen by each project will likely be different. Assistive tools are very attractive for bridging the gaps between the codes that address issues of language interoperability, type and data mapping, and differing build practices.

Many new code projects have adopted either component-based or object-oriented design methods, but a huge amount of code exists that was designed using monolithic methods. This code is often still maintained and a shift to modern software engineering practices is not practical for many reasons. These range from lack of resources to re-engineer the existing code to a skepticism on the part of the developers that such a shift would be worth the time and effort to both port the code and prove that the re-engineered code does not deviate from the existing, trusted code. The OnRamp project is intended to provide assistive and semi-automated tools to componentize codes that fall into these categories. Automatic tools both reduce the time and effort required to transform existing code, and well designed tools can ensure that human mistakes in the porting process will not occur that introduce errors in the computations. The OnRamp project targets the Common Component Architecture (CCA) tool-chain and design pattern[1] [4].

In this paper, we describe a set of source annotations that can be used to mark up source code in a non-intrusive manner and to express these higher level properties. OnRamp is a tool that extracts these annotations to organize calling interface information provided by commercial compiler front-ends, generating the necessary component Interface

[1]The CCA project is part of the US Department of Energy SciDAC program.

CBHPC '09, Nov. 15-16, Portland, OR, USA

Description Language (IDL) and binding implementations. The CCA project adopts the Scientific Interface Description Language (SIDL) defined by the Babel tool [10, 7]. This tool will be described in the context of the CCA Bocca toolchain [3], and demonstrated for C language code. Finally, we will describe the current and ongoing work to apply these annotations to programs in Fortran and C++.

2. ONRAMP REQUIREMENTS AND WORKFLOW

The use of OnRamp by an end user is relatively light weight and reflects the original OnRamp design goal of minimizing perturbations to the existing development and maintenance practices for code projects. There are two sorts of workflow concerning OnRamp:

1. The workflow required of the user/developer of OnRamp. Simplicity and nonintrusiveness of the user's workflow *is* a requirement in the design of OnRamp, probably the most important requirement.

2. The workflow of OnRamp itself. Because OnRamp is designed to sit at the top of the CCA tool chain, it must use the rest of the CCA tools to accomplish its function.

2.1 Requirements

It is important to recognize that unlike tools that generate wrappers to make codes in one language callable by those written in another, decomposing a program into a set of components requires information not present in the pure syntax of the language. Components represent portions of program code that are related based on semantics at the application level of abstraction. These application level semantics are not explicitly stated in the language syntax, but are easily identified by a programmer familiar with the code. Given this knowledge of the application-level semantic structure of the code, it becomes possible to inform syntax-directed wrapping tools, like OnRamp, on how to best break up the code into components and generate wrapper code to bind the legacy code to components.

To encourage adoption of component design techniques for legacy codes, it is important that a tool minimize perturbation of the existing code. By this we mean that the tool must not require changes to the legacy code itself beyond annotations that will not disturb the code and existing programs and build structure based upon it. As such, we adopt a comment-based annotation scheme inspired by the Doxygen [1] and Javadoc [2] code documentation generator tools. In this paper we focus solely on the code annotation, interface and wrapper generation process, and do not attempt to address orthogonal (but critically important) issues of build and configuration systems. We do not believe that at the current time it is possible to automate the integration of legacy build systems (many of which are hand coded, lacking any standard practices) with the CCA build system. This is a significant obstacle to adoption of component-based systems for legacy codes, but attempting to address it in this paper would distract from the discussion we are presenting. To a large degree it is hoped that OnRamp will obviate this concern. Because it orchestrates the CCA Bocca tool to create not only components but their attendant build system, the user need not be concerned with maintaining a new build system but rather just using it. A brand new build tree is created along with the components every time OnRamp is invoked. It is expected that the user will only change their original code, and the annotations and not the automatically generated components and their build system.

The basic sequence of developer activities using OnRamp to componentize a code is summarized:

1. **Refactor the design of existing code into components**. The physics and functional decomposition of the code is the principle way to view it as components. However using TAU[17] or another source code analyzer/profiler to examine call stack traces during execution, the developer will determine likely methods that belong together in the same port interface, and interfaces that belong to the same component. In general, a component should group methods that call each other frequently and disparate components should call on methods in other components less frequently. The call stack trace will aid the component developer in component/port decomposition of their code.

2. **Annotate the existing code**. The developer inserts the annotations according to the analysis in item 1 and other considerations related to the specific application. In C and C++ projects, this can be either comment annotations of header files (the preferred method) or subroutines in source files. In Fortran projects annotations will necessarily occur in source files.

3. **Iterate 1 and 2 above**. The cost for picking the wrong component/port decomposition is low. Because it only involves a relatively few annotations, new decompositions can be experimented with and tested with little effort.

2.2 OnRamp Workflow

OnRamp itself orchestrates a variety of tools underneath it to accomplish automatic generation of components from annotations. There are three major parts that underlie OnRamp functionality:

1. **Execute static analysis tools**. The static analyzer based on the Program Database Toolkit (PDT) [11] extracts the set of function interfaces and data types used in the code from the sources, and parses the OnRamp annotations to decorate the PDT analysis output with OnRamp meta-data. These data include the mappings from function and (if applicable in the language of the code) object type names to SIDL methods, interfaces, classes, and (where CCA is to be used) ports and components.

2. **Generate component wrappers and attendant build system**. OnRamp uses Bocca to generate empty components (*sans* implementation) in a specified language with its build system. Because these are CCA components, language interoperability is built in due to the use of Babel[10].

3. **Marshal arguments**. OnRamp incorporates its own type-mapping facility that marshals arguments from the developer's code into the Babel/SIDL interfaces required by CCA. Type-map's are contained in a developer-augmentable library that marshal developer types in a particular language to and from corresponding SIDL types.

   The developer invokes the type-map through OnRamp with source code embedded annotations as well. This type-mapping facility is necessarily programming language specific, each type-map will have to be implemented for every language for which it is used.

Annotations indicate the developer's choice of refactoring into components. OnRamp reads these annotations and empty components are generated by a series of commands issued automatically to the Bocca tool [8]. Bocca tracks all of the SIDL symbols in the generated wrapper project, their relationships, and dependencies, and checks consistency SIDL interfaces and dependencies. Bocca then generates a corresponding build directory layout, class and component definitions without user implementation code, and a build system for compiling the implementations (see Figure 1, *Step 1-2*). This includes all language bindings requested by the OnRamp user so that the component, regardless of implementation, can be used by any one of C, C++, Fortran, Python, and Java.

OnRamp uses the annotations and the results of the static source code analyzer to map the implementation from the original code to the empty components generated previously. It then fills in the empty implementations that wrap the original code and then calls the Bocca-generated build system to complete the build of the SIDL wrappings (Figure 1, *Step 3*). Bocca provides tools for splicing OnRamp-generated blocks (the code that must be written manually if OnRamp is not used) into the implementation files. The build system generated is based on autoconf and GNU make for portability. The build is customizable by anyone familiar with autoconf and will function in the absence of Bocca. Importantly, if changes are made to the original annotations or method signatures, OnRamp uses Bocca is used to update the information and maintain consistency in the build system.

The language interoperability of CCA components is obtained by mapping the method signatures in the original source code into SIDL. OnRamp has a default mapping of the these types into the SIDL and if the developer makes no other annotations this is what OnRamp generates. However, it is more effective to provide a generic way of mapping types to and from the SIDL in a form that coheres better with the developers original intent and in a form that is more natural. OnRamp provides a type-mapping library that contains some generally useful type-maps (e.g. arrays, etc.), but more importantly, allows the developers themselves to augment the library with type-maps directly. Type-maps are invoked by annotations as well. OnRamp uses results from the static source code analyzer and the type-map library to transmute the types to and from the SIDL representation necessary for CCA components.

3. BACKGROUND

OnRamp relies on annotations to add the extra semantic information necessary to support automatic componentization. This is not uncommon in other fields of computer science. One example is the semantic web where extra annotations are made that relate to purposes outside of rendering the content in a browser[16]. Even in the HPC arena, OpenMP relies on pragma's to give OpenMP aware compilers directives as to the developer's design for parallelism[6]. While there appears to be no other work beyond OnRamp directly related to componentizing code from source annotations, there are a number of similarities to problems solved in previous work that automate language interoperability and application generation. These correspond to the first two parts of OnRamp functionality mentioned in the Section 2.

3.1 Typemaps and Language Interoperability

Scientific programmers (and those in the general computing community) have employed more than one programming language in a single program for a long time. Over the last decade, this has become especially apparent with the popularity of scripting languages such as Python used to tie together codes written in C, C++, and Fortran with a convenient scripting layer. By far the most well known of the tools supporting the generation of glue code between compiled languages and scripting languages is SWIG (Simple Wrapper Interface Generator) [5]. SWIG includes facilities to parse C or C++ to be wrapped, code generation and type mapping mechanisms to generate wrapper code layers, and customization of type mapping patterns for individual projects.

Although first created for the scientific community, SWIG has limits that have motivated the development of specialized tools for supporting languages and language features common in scientific codes. First, SWIG lacks any support for wrapping codes written in Java, Python, or Fortran, specifically those based on the Fortran 90 and later language standards. Fortran 90 introduced a rich array data type that associates dimension and indexing information (such as strides) with the base memory pointer of the actual array contents. The Chasm project [12] was initiated to deal with cross language and cross compiler interoperability of Fortran 90 arrays. In particular, the Chasm project provides a common interface to compiler-specific array descriptors (also known as "dope vectors") allowing programmers to manipulate F90 arrays without knowing the underlying representation. This is particularly important when crossing language boundaries to languages that do not have rich array data types, such as C. Programmers could use Chasm to access dimension information in addition to the base pointer of the array. Adopting the more limited F77 style of array passing causes this metadata to be lost.

Second, the input format of SWIG forces a separation of wrapping knowledge from the maintained project source code, because SWIG annotations are not compilable C statements or C comments. OnRamp seeks to unify the wrapper generation capabilities of SWIG and related tools with the ability to deal with the primary sources for richly typed languages such as Fortran [14] [13] [9].

3.2 Application Generation

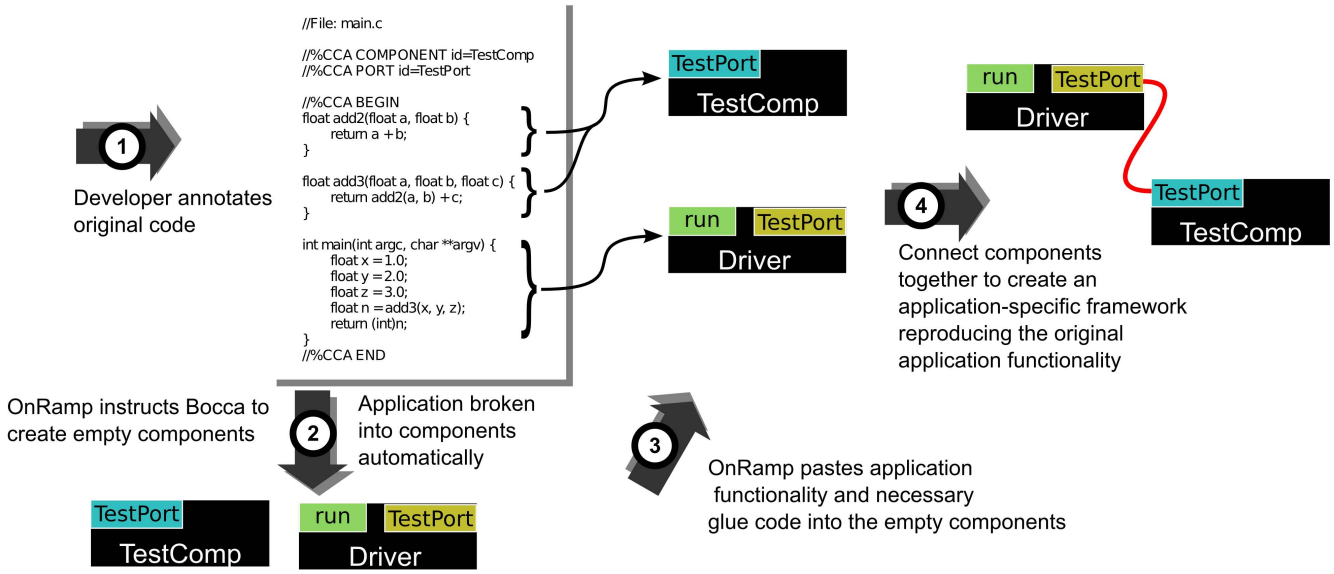

Figure 1: OnRamp workflow diagram. *Step 1*: **The developer indicates the methods, the CCA Port interfaces to which they belong, and the CCA components to which the Ports belong in commented annotations in the developer's original source (here as an example C).** *Step 2:* **OnRamp processes these annotations and the method signatures to instruct CCA's Bocca to create empty CCA components sans implementation.** *Step 3:* **OnRamp uses source code analysis and the context of the original annotations to substitute the original source code as implementations in the erstwhile empty components.** *Step 4:* **If the annotations are sufficient to decompose all of the original application's functionality (optional), then the components can be instantiate and connected to recover the original code's behavior.**

OnRamp relies heavily on Bocca and its ability to generate language interoperable components and their attendant glue code, complete with build system, automatically. The best analogue to Bocca outside of high-performance computing is Ruby on Rails[15]. Rails generates running web applications automatically from only a few specifics. Like Bocca and unlike OnRamp, Rails generates an application that is prototypical of the developer's ultimate requirements, and it is expected that the application will continue to be developed in the Rails-generated code tree. The use model for Bocca also generates components that are likely only "close" to what the developer really wants and further development on the component is expected to occur in the generated source code and build tree.

OnRamp is different from both Bocca and Rails in the sense that the generated source tree is not where future development is expected to occur. The OnRamp generated code is viewed as a derivative of the "main" development tree that is not modified by the developer but rather is a format that makes their code more available for use by others. This places a different burden on OnRamp than on tools like Bocca and Rails. OnRamp provides a "component view" of the original scientific application: every aspect of the representation of the code as components must be orchestrated by OnRamp and represented by annotations in the original code. Ultimately everything having to do with components must be representable as OnRamp annotations in some way. While the current version of OnRamp falls short of this requirement an important part of its capabilities is a "just do this" escape hatch that allows users to override its normal operation and insert code directly in the components. Viewed in this way, OnRamp is less of a tool and more of a new paradigm for producing component-based applications.

4. ANNOTATION SPECIFICATION

The first step in the OnRamp process is to annotate the user's source code. Annotations tell OnRamp how to interpret ambiguous type signatures, express uses/provides relationships between components, and generally constrain or augment the generated code.

OnRamp annotations are written in a simple domain-specific language. Each annotation has a name, following by a list of key-value pairs representing parameters, similar to this:

```
{ANNOTATION_NAME} {ARG}={VALUE} ... {ARG}={VALUE}
```

Annotations are embedded in specially formatted comments within the host source code files, similar to the schemes used by code documentation tools like Doxygen and Javadoc. Embedding annotations within comments reduces disturbance to the existing code, as compilers ignore the contents of comments in most cases. The only notable exception is the use of comments in the Fortran language to simulate pragma facilities present in languages like C and C++, as used by OpenMP [6]. Because OnRamp comment annotations use their own introductory keyword (i.e. `%CCA`) they avoid interference with all of the existing comment-based tools of which we are aware.

A comment is considered to contain one or more annotations if it begins with any amount of whitespace (including newlines), followed by the `%CCA` introductory keyword. Each line thereafter within such a comment (including the line containing the `%CCA` keyword) may contain a single annotation. Single line comments, then, may only contain one annotation. Whitespace characters other than newlines are ignored. In multi-line comments, blank lines between annotations are also accepted and ignored. So, the following forms are all acceptable (using C++ comments):

```
/* %CCA {ANNOTATION}
  {ANNOTATION} */


/*
   %CCA
   {ANNOTATION}

   {ANNOTATION}
*/


/*%CCA {ANNOTATION} */


//%CCA {ANNOTATION}
```

Annotations are applied to routines or data structures (*program elements*, or just *elements*). It is permissible to annotate either the declaration or the definition of a particular element, but not both. So, for example in C++, annotating the same function in both the header file and the implementation file is considered an error.

Annotations obey very simple scoping rules. An annotation is assumed to apply to the element immediately following its containing comment. There are also special `BEGIN` and `END` annotations used to define scoping regions within the source code. Annotations that appear immediately before a `BEGIN` annotation apply to every element in corresponding region. Consider the following example:

```
//%CCA A1
void foo();
//%CCA A2
//%CCA A3
//%CCA BEGIN
void bar();
void row();
//%CCA END
void get();
```

Here, the declaration `foo` is annotated with `A1`. Declarations for `bar` and `row` are each annotated with both `A2` and `A3`. Finally, the declaration for `get` has no applicable annotations.

Nested `BEGIN`/`END` blocks are allowed, and an `END` must be in the same file as its matching `BEGIN`. Annotations may also be placed within a `BEGIN`/`END` block before individual elements, and these effectively override the outer annotations for the elements to which they apply.

4.1 Interface organization

Some OnRamp annotations are used to define the decomposition of an existing monolithic code structure into a set of components and ports. The `PORT` annotation applies to a routine, and indicates that it belongs to a named CCA port, identified by the `id` parameter to the annotation. Similarly, the `COMPONENT` annotation indicates that a routine should be included in a named CCA component, also identified by `id`.

```
//%CCA COMPONENT id=myComponent
//%CCA PORT id=myPort
//%CCA BEGIN

void foo();
void bar(int x, int y);
void row();

//%CCA END
```

In this example, all three routines within the block would be placed within a port named `myPort` provided by a component named `myComponent`. If we want to instead place `bar` in a different port, say `myPort2`, we place an annotation immediately before the routine. This will override the port specified in the surrounding `BEGIN`/`END` block.

```
//%CCA COMPONENT id=myComponent
//%CCA PORT id=myPort
//%CCA BEGIN

void foo();

//%CCA PORT id=myPort2
void bar(int x, int y);
void row();

//%CCA END
```

Here, note that the `PORT id=myPort2` annotation *only* applies to `bar`, not to `row`.

4.2 Type disambiguation

Wrapper generation tools must be able to map types from the host language (e.g. C) to the target language (e.g. SIDL). Sometimes, the mapping is fairly obvious, as in the case of integers or strings. The problem becomes more complex when dealing with *ambiguous types* – those that do not have a clear one-to-one correspondences across the language boundaries (see Figure 2). We use annotations to disambiguate the mapping. In the case where the user chooses not to annotate the ambiguous type, OnRamp will default to using a conservative mapping.

4.2.1 Array types

In the context of scientific programs, the most common sources of type ambiguity are arrays. Consider a function that takes a two-dimensional integer array as an input and returns the sum of the values contained within the array. In Fortran 90 and later (hereafter referred to as Fortran), one can declare this function interface as:

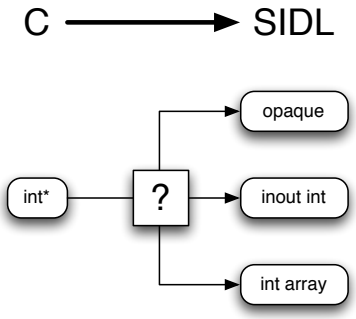

Figure 2: The ambiguous types problem – all of the shown type mappings are valid, but which one to choose?

```
FUNCTION ADDER(X)
  INTEGER, DIMENSION(:,:), INTENT(IN) :: X
  INTEGER :: ADDER
END FUNCTION
```

No additional arguments are necessary due to the presence of intrinsic functions defined as part of Fortran which operate on array descriptors that accompany the contents of arrays carrying information about the shape and dimensional extents. On the other hand, no such descriptor exists in C, so a reasonable equivalent signature would be:

```
int adder(int *X, int nrows, int ncols);
```

Even the use of multidimensional array syntax in C (e.g., `X[][]`) requires the explicit passing of dimension size information, as the argument `X` carries with it no descriptor information to allow code to dynamically discover the shape of `X`. Even worse, there is nothing about the first argument that indicates that the pointer is to be interpreted as an array in the first place.

OnRamp takes a conservative approach, and by default maps all pointers to a catch-all *opaque* type in SIDL. While always correct, this mapping is generally inelegant, since a `opaque` element is devoid of type information. An array annotation informs OnRamp that the pointer is in fact to be interpreted as an array and provides guidance for OnRamp to acquire necessary dimensional information.

To properly wrap this interface within a component interface and provide a SIDL description based on SIDL array types, one must be able to determine that the three arguments to the C function are related and together represent a single array.

We can use the `ARRAY2D` annotation to tell OnRamp that a pointer-typed function argument represents a two-dimensional array, with the number of rows and columns provided as function arguments, like so:

```
/* %CCA ARRAY2D elements=X rows=nrows cols=ncols */
int adder(int *X, int nrows, int ncols);
```

Similar annotations exist for 1D and 3D arrays, with plans to add annotations for higher dimensional arrays in the future. We also support SIDL rarrays [7] through the `RAWARRAY` annotation.

To complicate matters, there is no well defined pattern that C programmers must follow when passing arrays and their corresponding dimensional arguments. It is entirely possible to encounter other signatures, such as:

```
int adder(int *X, int nrows, int ncols, int unrelated);
int adder(int *X, int unrelated, int nrows, int ncols);
int adder(int nrows, int ncols, int *X);
```

Often a single style is adopted within a single software project, but styles may vary between projects. To minimize perturbation of legacy codes, we must *not* force any changes to code to match a predefined style, but instead provide mechanisms to adapt to each project individually. Within an annotation, argument parameters may be specified with either a symbolic name (e.g. `nrows`), or the positional offset of the argument within its list. This gives the user some flexibility in creating a few annotations that, combined with `BEGIN/END` blocks, can apply to many similar functions within a project.

4.2.2 Structured types

The other category of types that require attention for disambiguation are structured types, such as C `structs`, C++ classes, and Fortran user-defined types.

In previous versions of SIDL/Babel that do not support structures, the only resort available for mapping such a structure was to use the `opaque` type. With the recent addition of structures to SIDL, OnRamp is now able to map most structured types onto SIDL equivalents. This mapping is accomplished by a recursive descent into the structure, applying type mappings to the contents.

Structs can be mapped either as formal arguments or as declared data types. The first case is not really different from the general case of regular function arguments. You can pick out a particular structure field to map by name, e.g. `myStruct.x`. The complete structure will be mapped recursively, with type maps applied to the sub-structures or primitives picked out by the annotations.

```
struct MyStruct {
  int x;
  int *y;
  double i;
};

//%CCA ARRAY elements=myStruct.y length=myStruct.x
int usesMyStruct(struct MyStruct myStruct);
```

Alternatively, the structure can be mapped at the point of declaration. In this case, the mapping will be applied wherever the struct is encountered in the code, and need not be explicitly specified for each function. Note that here, the field names need not be qualified:

```
//%CCA ARRAY elements=y length=x
struct MyStruct {
  int x;
  int *y;
  double i;
};
```

OnRamp also provides special support for the `complex` type provided within SIDL. It is not uncommon to encounter codes written in C or C++ that contain a structure representing complex numbers, such as:

```
struct complex {
  float re, im;
};
```

The entire structure can be mapped to a SIDL complex type using the `COMPLEX` annotation. Like regular structs, this annotation may be applied at the point of declaration to make the mapping universal, or on a per-use basis in function arguments.

4.2.3 Miscellaneous annotations

There are a few other annotations that are used to alter function type signatures. The `IGNORE` annotation will cause an argument that appears in the host language function signature to be elided from the generated SIDL function signature. This can be useful if, for example, a length parameter is merged into an array type with the `ARRAY` annotation, such that it no longer needs to be explicitly passed into the function:

```
//%CCA ARRAY elements=X length=len
//%CCA IGNORE arg=len
double sum(double *X, int len);
```

This will result in a SIDL signature:

```
double sum(in Array<double> X);
```

While not strictly necessary, judicious use of `IGNORE` can improve the readability of the generated SIDL interface.

The `RETURNVIAPARAM` annotation is applied to functions specified in the host language to have a return value. It alters the generated SIDL signature such that the return type is void, and the the value is passed out of the function with an additional `OUT` argument. The name of the new argument, as well as its relative position in the argument list, are specified with annotation parameters. For example, the following use of this `RETURNVIAPARAM` annotation:

```
//%CCA RETURNVIAPARAM arg=squaredVal pos=last
double squared(double val);
```

would result in the following SIDL signature:

```
void squared(in double val, out double squaredVal);
```

This pattern is sometimes useful if there is a pre-determined desired SIDL signature.

Finally, the `INTENT` annotation is able to alter any type-compatible function argument to have the SIDL `in`, `out`, or `inout` specifier. In general, any pointer type in the host language can be specified as `out` or `inout`. This will cause the referenced type to be mapped, and then passed with the specified direction. For example:

```
//%CCA INTENT arg=val dir=out
void foo(double *val);
```

Without any annotations, `val` would be mapped to `opaque`. With the `INTENT`, it is mapped as:

```
void foo(out double val);
```

For more information on the various supported annotations, see the OnRamp documentation [18].

4.3 Inter-component relationships

Up to this point, we have only considered components from the perspective of a "provides" port. The ultimate goal of OnRamp though is to achieve full application componentization, addressing the issue of ports that are "used" as well as provided. A more complicated code may export several components, providing and using several ports with inter-dependencies that require resolution of these dependencies automatically. The current work with OnRamp, demonstrated in [9], used call-graph analysis to determine where in a program functions that had been absorbed into ports were used.

The approach is straightforward, and in the most likely use cases, full automation is possible. A first pass over the code yields all of the annotation information defining the component decomposition of the routines (as discussed above in Section 4.1). With this list of routines and their locations in the component-based organization of the code, we can then perform a second pass over the code to determine the locations of all invocations of these functions. These locations correspond to call sites where traditional function calls must be replaced with the component-oriented process of acquiring a handle to the port providing the routine from the CCA framework runtime layer, and replacing the original function call with a call to the appropriate function pointer inside the handle.

This approach is seriously limited in cases involving the use of function pointers in the annotated code. In order to replace a function call with a component invocation, we must be able to identify which function is being called. Where function pointers are used, we are not currently able to match call sites to componentized methods. This is a difficult problem in general, and an area of current active research. In particular, we are examining static analysis methods that may enable us to recover the possible functions being called through function pointers in some cases. For the remaining cases where static analysis cannot recover the function call information, we plan to provide annotations to disambiguate the possible meanings of the call site.

5. TXPHYSICS USE EXAMPLE

Here we walk through a sample of a production plasma physics code [20], written in C from Tech-X Corporation. We annotate the code and run OnRamp to convert part of the ion physics package [19] to a CCA port and a component. The code serves as a demonstration of OnRamp's use in realistic scientific simulation development. In Figure 3 we show the annotation of a single routine for charge exchange in a plasma. There are five similarly annotated methods in the full example; for clarity, we show a single representative example function here.

File `./txphysics-2.0.0/txionpack/txionpack.h`:

```
/**
 * This routine does charge exchange
 * It is based on the Schlasuna algorithm
 * and is based on an algorithm
 * by G. Maynard
 */
//%CCA COMPONENT id=txphysics
//%CCA PORT id=ionpack
//%CCA RAWARRAY elements=xsec_array length=array_size
//%CCA RAWARRAY elements=yep_array length=array_size dir=in
//%CCA RAWARRAY elements=yzp_array length=array_size dir=in
void get_chg_xchg_xsec(double xsec_array[],
                       double yep_array[],
                       int yzp_array[],
                       int array_size,
                       int yzc);
```

Figure 3: Annotations (in red) for one of five methods wrapped from the Tech-X txphysics package. The `COMPONENT` directive identifies the component, named "txphysics", in which this method will be implemented. The `PORT` directive identifies the port, named "ionpack", which will have this method as one of its members. The `RAWARRAY` directive directs the type mapping system to convert the array in the method's argument list to a SIDL `rarray` type. This will become the type used in the SIDL representation of this method on the "ionpack" port interface.

Figure 3 shows the annotations added to the original header file in the original code. Once the annotations have been added, OnRamp is invoked on the header file, like so:

```
$ onramp.py -p myionpack \
    ./txphysics-2.0.0/txionpack/txionpack.h
Parsing files and generating database
Running anc
Project: myionpack
Database file: myionpack.db
```

OnRamp analyzes the source code and parses the annotations,

invoking Bocca and the type-map library to create CCA ports and components. The new files are created in the usual Bocca directory structure:

```
$ ls ./myionpack
BOCCA     config.log    configure.ac1  Makefile
buildutils  config.status  depl  make.project   ports
components  configure   external  make.project.in  README
config    configure.ac   install  make.rules.user  utils
```

Bocca generates a set of skeleton files for the new "txphysics" component, which provides the "ionpack" port. OnRamp fills in the component bodies with wrapper code which converts the arguments from SIDL to their original C representation, calls the original function, and then converts the return value back into its SIDL data type (see Figure 4 & Figure 5).

The only remaining step is to tell Bocca where to find the txphysics library for linking, and then run the familiar "configure; make" sequence to build the component.

File `./myionpack/ports/sidl/myionpack.ionpack.sidl`:

```
package myionpack version 0.0 {
    interface ionpack extends gov.cca.Port {

// Four other methods elided
    .
    .
    .
  void get_chg_xchg_xsec (
  inout rarray<double,1> xsec_array(array_size),
  in rarray<double,1> yep_array(array_size),
  in rarray<int,1> yzp_array(array_size),
  in int array_size, in int yzc);
  }
}
```

Figure 4: This is the SIDL interface that OnRamp generates from the annotations and method signature of the code in Figure 3. The interface defines a port that will be provided by the "txphysics" component identified in Figure 3. The component, via this port, is usable by any other CCA component written in any other language.

Using the same `myionpack` Bocca project, we add a `Driver` component that uses the port interface `ionpack` provided by the "txphysics" component we just generated. After another "configure; make", the components are ready to be hooked together and run (see Figure 6).

File `./myionpack/components/myionpack.txphysics/\ myionpack_txphysics_Impl.c`

```
/*
 * Method:  get_chg_xchg_xsec[]
 */

void
impl_myionpack_txphysics_get_chg_xchg_xsec(
  /* in */ myionpack_txphysics self,
  /* inout rarray[array_size] */ double* xsec_array,
  /* in rarray[array_size] */ double* yep_array,
  /* in rarray[array_size] */ int32_t* yzp_array,
  /* in */ int32_t array_size,
  /* in */ int32_t yzc,
  /* out */ sidl_BaseInterface *_ex)
{
  *_ex = 0;
  {
/* DO-NOT-DELETE
 * splicer.begin(myionpack.txphysics.get_chg_xchg_xsec)
 */

 // Argument conversion code generated by OnRamp:
 double *__onramp_arg_0001 = (double *)xsec_array;
 // ... Remaining argument marshalling code elided ...
  get_chg_xchg_xsec(__onramp_arg_0001, __onramp_arg_0002,
                    __onramp_arg_0003, __onramp_arg_0004,
                    __onramp_arg_0005);
/* DO-NOT-DELETE
 * splicer.end(myionpack.txphysics.get_chg_xchg_xsec)
 */
  }
}
```

Figure 5: This is the implementation template created by Bocca at the behest of OnRamp. OnRamp will generate the code necessary to convert the arguments from their SIDL representation to the types needed for the C function call, and then generates the appropriate call (shown in red). The marshaling code is straightforward but verbose, and so mostly is elided here. The wrapper code shown here must be generated in the host language (C in this case, since that is what txphysics uses), but the compiled component will be callable by any other CCA component written in Fortran, C++, C, Python, or Java.

6. CONCLUSIONS AND FUTURE WORK

OnRamp is a new system for automatically componentizing legacy codes. It supports the creation of CCA components from a variety of host languages, and minimizes code perturbation through its use of annotations embedded in code comments.

OnRamp is currently a work in progress. A preview release is available [18]. While OnRamp is in development and should be considered "beta" quality software, it is robust enough to be used on real codes like TxPhysics.

Presently, OnRamp is capable of generating code based on *provides* component relationships. We are working on adding robust support for refactoring code to automatically respect *uses* relationships, including the somewhat complicated cases involving analysis of call sites using function pointers.

We plan to extend OnRamp in the near future to include an improved type-map library structure and language interface, so that users can easily add their own type-mapsand annotations for their own use and for the wider community.

Finally, we plan to continue our ongoing effort to expand OnRamp to support to the complete set of Babel languages, including F90 in the near term.

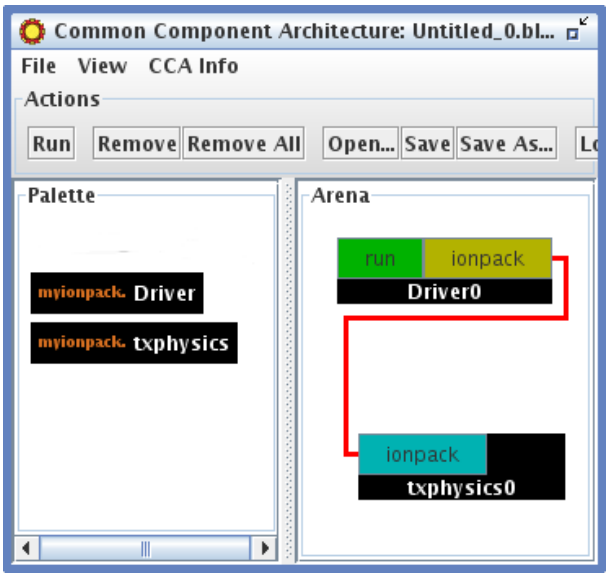

Figure 6: CCA wiring diagram using the OnRamp generated "txphysics" component.

7. REFERENCES

[1] doxygen. `http://www.stack.nl/~dimitri/doxygen/`, August 2009.

[2] javadoc - The Java API Documentation Generator. `http://java.sun.com/j2se/1.5.0/docs/tooldocs/windows/javadoc.html`, August 2009.

[3] Benjamin A. Allan, Boyana Norris, Wael R. Elwasif, and Robert C. Armstrong. Managing Scientific Software Complexity with Bocca and CCA. *Scientific Programming*, 16(4):315–327, August 2008.

[4] Robert C. Armstrong, Gary Kumfert, Lois C. McInnes, Steven Parker, Benjamin A. Allan, Matthew J. Sottile, Thomas Epperly, and Tamara Dahlgren. The CCA component model for high-performance scientific computing. *Concurrency and Computation: Practice and Experience*, 18(2):215–229, 2006.

[5] David M. Beazley. SWIG: An easy to use tool for integrating scripting languages with C and C++. In *Proceedings of the 4th Annual Tcl/Tk Workshop*, 1996.

[6] Rohit Chandra, Ramesh Menon, Leo Dagum, and David Kohr. *Parallel Programming in OpenMP*. Morgan Kaufmann, October 2000.

[7] Tamara Dahlgren, Thomas Epperly, Gary Kumfert, and James Leek. *Babel User's Guide*. CASC, Lawrence Livermore National Laboratory, Livermore, CA, 2004. `http://www.llnl.gov/CASC/components/docs/users_guide.pdf`.

[8] Wael R. Elwasif, Boyana R. Norris, Benjamin A. Allan, and Robert C. Armstrong. Bocca: a development environment for hpc components. In *CompFrame '07: Proceedings of the 2007 symposium on Component and framework technology in high-performance and scientific computing*, pages 21–30, New York, NY, USA, 2007. ACM.

[9] Geoffrey Hulette, Matthew J. Sottile, Benjamin A. Allan, and Robert C. Armstrong. Using CCA and Onramp to Generate an Application-specific Framework from a Monolithic Application. (unpub), poster, SC 2008, `http://eprints.cca-forum.org/156/`, November 2008.

[10] Scott Kohn, Gary Kumfert, Jeff Painter, and Cal Ribbens. Divorcing language dependencies from a scientific software library. In *Proceedings of the 10th* SIAM *Conference on Parallel Processing*, 2001.

[11] Kathleen A. Lindlan, Janice Cuny, Allen D. Malony, Sameer Shende, Bernd Mohr, Reid Rivenburgh, and Craig Rasmussen. A Tool Framework for Static and Dynamic Analysis of Object-Oriented Software with Templates. *SC Conference*, 2000.

[12] Craig E Rasmussen, Matthew Sottile, Sameer Shende, and Allen Malony. Bridging the language gap in scientific computing: the Chasm approach. *Concurrency and Computation: Practice and Experience*, 2005.

[13] John Reppy and Chunyan Song. Application-specific foreign-interface generation. In *Proceedings of the Fifth International Conference on Generative Programming and Component Engineering*, pages 49–58, October 2006.

[14] R.D. Rivenburgh, C.E. Rasmussen, K.A. Lindlan, B. Mohr, and P.H. Beckman. Automatic generation of Perl extensions to C++ and Fortran 90 class libraries. In *OReilly Open Source Software Convention*. OReilly, 2000.

[15] Sam Ruby, Dave Thomas, and David Hansson. *Agile Web Development with Rails Third Edition*. Pragmatic Bookshelf, 2009.

[16] Toby Segaran, Colin Evans, and Jamie Taylor. *Programming the Semantic Web*. OReilly Media, Inc., 2009.

[17] Sameer S. Shende and Allen D. Malony. The Tau Parallel Performance System. *Int. J. High Perform. Comput. Appl.*, 20(2):287–311, 2006.

[18] Matthew Sottile and Goeffrey C. Hulette. OnRamp SciDAC web site. `https://outreach.scidac.gov/scm/?group_id=28`. Follow instructions for anonymous subversion checkout.

[19] P. H. Stoltz, M. A. Furman, J.-L. Vay, A. W. Molvik, and R. H. Cohen. Numerical simulation of the generation of secondary electrons in the high current experiment. *Phys. Rev. ST Accel. Beams*, 6, 2003.

[20] P. H. Stoltz, S. A. Veitzer, R. H. Cohen, A. W. Molvik, and J.-L. Vay. Energy loss, range, and electron yield comparisons of the crange ion-material interaction code. *Nuclear Instruments and Methods in Physics Research Section A*, 544, 2005.