PROJECT TITLE	:	Center for Technology for Advanced Scientific Component Software
DOE AWARD #	:	DE-FC02-08ER25839
INSTITUTIONAL PI	:	Dr. Matthew Sottile
INSTITUTION	:	University of Oregon, Eugene, OR, 97403
PERIOD OF PERFORMANCE	:	June 2008 — June 2010
FINAL REPORT DATE	:	June 30, 2010

1 Summary of work

The UO portion of the larger TASCS project was focused on the usability subproject identified in the original project proposal. The key usability issue that we tacked was that of supporting legacy code developers in migrating to a component-oriented design pattern and development model with minimal manual labor. It was observed during the lifetime of the TASCS (and previous CCA efforts) that more often than not, users would arrive with existing code that was developed previous to their exposure to component design methods. As such, they were faced with the task of both learning the CCA toolchain and at the same time, manually deconstructing and reassembling their existing code to fit the design constraints imposed by components. This was a common complaint (and occasional reason for a user to abandon components altogether), so our task was to remove this manual labor as much as possible to lessen the burden placed on the end-user when adopting components for existing codes.

To accomplish this, we created a source-based static analysis tool that used code annotations to drive code generation and transformation operations. The use of code annotations is due to one of the key technical challenges facing this work — programming languages are limited in the degree to which application-specific semantics can be represented in code. For example, data types are often ambiguous. The C pointer is the most common example cited in practice. Given a pointer to a location in memory, should it be interpreted as a singleton or an array. If it is to be interpreted as an array, how many dimensions does the array have? What are their extents? The annotation language that we designed and implemented addresses this ambiguity issue by allowing users to decorate their code in places where ambiguity exists in order to guide tools to interpret what the programmer really intends.

1.1 Change of scope

When the UO project was initiated after Dr. Sottile departed the Los Alamos National Laboratory, we were tasked with creating the "CCA-lite" implementation of the framework tools. This lite version simplified the build process and overall toolchain by removing the dependency on SIDL and Babel for language interoperability, and removing the dynamic linking support that was observed to be unused in practice by users. The CCA-lite project was based on an alternative framework to the Babel/CCAFFEINE (see [1]) frameworks that implement the full CCA specification with support for dynamic configuration of components and dynamic library-based composition. It was observed during the first phase of the CCA effort prior to TASCS that few users took advantage of these features other than a small number of "power users". For the rest of the users, these features simply got in the way and added complexity to using the toolchain.

For reasons of simplifying the tool set that CCA provided to end users, the TASCS community as a whole decided to retarget the efforts from CCA-lite to instead provide automated tools that

removed much of the complexity of targeting the full-featured part of the CCA toolchain from the user. Instead of providing two frameworks (lite and full), one full framework would exist with a set of supporting tools that gave it the appearance of the original CCA-lite. To an end user, this accomplished the same goal as CCA-lite (reducing the cost of entry to a component based design model), while at the same time reducing the size of the CCA infrastructure. Instead of maintaining two simultaneous lines of framework infrastructure, one development line would exist with assistive tools to aid users in targeting it.

The reasons for this were clear from a software engineering and project maintenance perspective. Instead of two frameworks tracking one standard, requiring constant attention to ensuring that they both accurately tracked the CCA standard, reducing the number of frameworks removed the potential for version skew and deviation from the common goal.

2 Annotation driven analysis and generation

The manner by which these assistive tools functioned was based on analysis of existing codes. Developers who already had codes written in a pre-component design style could use their codes themselves as the starting point for the componentization process. This concept is described in further detail in [2] and [3].

A critical issue that needed to be addressed was the lack of sufficiently rich semantic information present in the source code alone of an existing code with respect to the intended decomposition of the software into a component model. For example, a raw code typically exposes a set of programmatic entities (routines, data structures, namespaces, etc...) that are intended for code organization, but not necessarily functional and semantically meaningful groupings. It is not possible to accurately infer the relationships between data structures and routines to automatically determine which should be grouped to form a higher-level component. To achieve this, annotations are required to allow the programmer to decorate the source code in a syntactically neutral manner to express this information. Another example of this semantic ambiguity was mentioned earlier related to data types. Much of our work on the annotation language was focused on annotated data types and the expression of "type maps" that define how the user intends data types in one language to be mapped onto another. For non-primitive types such as C structs and Fortran user defined types, this is very important.

The final product of our work is embodied in the following results that are available as open source software.

- The OnRamp annotation language
- A PDT-based source analyzer that associates annotations with syntactic elements of the language.
- An OCaml code generation tool
- The "Typo" type mapping engine

2.1 Annotation language

The OnRamp source annotation language is very simple by design. As mentioned above, one key goal was to make annotations "syntax-neutral", which means that annotations do not perturb the underlying language or program in any way. This is a common approach taken in systems based on

annotating an existing language, such as OpenMP. In OnRamp, we chose to define an annotation language that resides in program comments. The form of an annotation is as follows:

```
// TAG NAME KEY=VALUE KEY=VALUE KEY=VALUE ...
```

The `TAG` is used to distinguish an OnRamp comment from any other. This is followed by the `NAME` of the annotation. What follows are a sequence of key/value pairs that are considered to be parameters to the named annotation. For example, if the named annotation is used for namespace management, then the parameters specify which namespace is being dealt with and how. Similarly, the annotation name may correspond to an abstract data type (such as a 2D array), and the arguments define how the actual language artifacts map to the abstract type (base pointer, dimension information, etc...).

2.2 Source analysis

A source analysis tool based on the Program Database Toolkit was created as well. This tool takes source code in C, C++, or Fortran, and produces a database representation that can be queried. Queries may include searching for specific routines, their parameters, or various types that are used. During our work, we created a new interface to the existing PDT project that was based on SQLite. The existing PDT query interface was based on a legacy C++ library that requires any query of interest to be hand coded using C++ iterators. We recognized that the SQL language provides a well defined method for expressing queries that has been standardized in the broader software engineering community. The SQLite library provided a corresponding public domain implementation that is available on all platforms of interest. The use of SQLite reduced the complexity of our code and increased the sophistication of the queries that we could perform for extracting information about the source being analyzed. Furthermore, a wide variety of language bindings exist to the SQLite library allowing us to write tools based on the PDT data without the requirement of a wrapper around the C++ interface.

We contributed the SQLite binding back to the PDT project in hopes that others who could benefit from PDT could have an alternative to the C++ Ductape interface and can use an interface based on an industry standard data query language. Our code is now distributed under the `contrib` portion of the PDT package.

2.3 Code generators and Typo system

The final major component of the project was the code generation and type mapping tool. This is described in detail in [4]. This work focused on the interpretation of both static analysis results of code (such as function interface and type information) and annotations associated with code artifacts. Two steps form the core of this work.

1. Type map application and reduction
2. Code generation

Type map application and reduction is implemented using a term rewriting strategy in which types correspond to term variables in a formula, and rewrite rules exist that replace terms with those that correspond to a type mapping. For example, in the context of the CCA project we are concerned with mapping native language types to and from the SIDL language. An integer in C

may be represented as having the type term CINT, while the equivalent in SIDL may be SIDLINT. A rewrite rule exists that states that when binding C to SIDL, all instances of CINT are replaced with SIDLINT. As terms are rewritten, the marshalling code associated with a rewrite is created to be used at code generation time. In trivial cases like integers and floating point numbers, very little exists in this generated code. The generated code becomes more interesting when considering richer data types like arrays.

A term rewrite rule for an array is responsible for mapping the language representation of the array (such as a typed pointer `(int *)`) onto a SIDL representation (`Array<int>`). During this process, the array contents must be marshalled. In the basic, default case, in which the type mapper only knows the base address of the pointer without details like the dimension or extents of the array, the mapping is to a low level SIDL "raw array".

Annotations come into play because they can be used to map arrays with defined lengths and dimensions into SIDL. Consider the following function signature:

```
void norm(double *x, int l)
```

The intent of the programmer who wrote this is to interpret the array pointed at by `x` as being a 1D array with length `l`. An annotation tells the tool that for this function, `x` represents an array with length `l`. This allows the type mapper to map the array onto a regular SIDL array (instead of a low-level raw array), and associate a length with it. The resulting code that is generated associated with the type mapping and term rewrite rules implements this for both the marshalling and unmarshalling sides of the language binding.

This work was in-progress at the time of completion of this grant. We are continuing this line of work though, as it forms the basis of the Ph.D. work of Geoffrey Hulette. We believe that this approach of formalized type mapping based on term rewriting techniques driven by annotation guided static analysis has a wide reaching applicability in the computing world and plan to actively continue this research.

2.4 Software releases

All of the code for this work has been packaged and deployed for download by users on the SciDAC Outreach web site. The full source repository is available along with archives of the major releases during the lifetime of the project.

`https://outreach.scidac.gov/projects/cca-onramp/`

3 Additional work

In addition to the work on the OnRamp project, we have pursued basic research exploring high level languages for coordinating and composing computations using a dataflow model. This model is currently employed by workflow systems popular in the grid computing community. In the tools used to support component composition in the TASCS project, a simple special purpose language was created that is implemented by the CCAFFEINE framework. During 2009, we conducted research on an Embedded Domain Specific Language (or EDSL) specifically for the purpose of composition of computations using a workflow style. Unlike the grid community where composition

involves independent, potentially distributed components, we considered the additional case where the composition of components occurred within a single program similar to that which the TASCS tools support.

This work was published in [5] and was based on an earlier effort [6]. The focus of [5] was to embed the flow description within the high level functional language Haskell. Typed streams were used to represent the flow of data between components, and instances of an abstract stream typeclass were demonstrated to implement the actual connection between components. This work was exploratory and has not continued beyond the prototyping stage described in the published work, as it was secondary in importance with respect to the usability project deliverables. A demonstration was implemented to accompany the publications that implemented coordination of pure Haskell components, as well as an implementation that supported multi-lingual components via a basic system call interface.

After publication in 2009, the authors (Sottile/Hulette) were invited to present an extended version of the paper at the International Workshop on Peta-Scale Computing Programming Environments, Languages and Tools (WPSE 2010) in Kyoto, Japan. The code that accompanies the paper and presentations is available as open source software at:

`http://github.com/mjsottile/hsworkflow`

4 References

[1] C. Rasmussen, M. Sottile, C. Rickett (2005). "A Gentle Migration Path to Component-Based Programming" Proceedings of the International Conference on Parallel Computational Fluid Dynamics (PCFD) 2005, Washington, D.C.

[2] G. Hulette, M. Sottile, R. Armstrong, and B. Allan (2008). "Using CCA and Onramp to Generate an Application-specific Framework from a Monolithic Application". Supercomputing 2008 Conference, Austin, TX.

[3] G. Hulette, M. J. Sottile, B. Allan, and R. Armstrong (2009). "OnRamp to CCA: Annotation-driven static analysis and code generation.", 2009 Workshop on Component-Based High Performance Computing (CBHPC 2009), held in conjunction with Supercomputing 2009, Portland, OR.

[4] G. Hulette (2010) "Reducing and Removing Extraneous Typemaps." University of Oregon, Department of Computer and Information Sciences Technical Report CIS-TR-2010-03.

[5] M. J. Sottile, G. Hulette, A. D. Malony (2009). "Workflow representation and runtime based on lazy functional streams.", Workshop on Workflows in Support of Large-Scale Science (WORKS), held in conjunction with Supercomputing 2009, Portland, OR.

[6] G. Hulette, M. Sottile, A. Malony (2008). "WOOL: A Workflow Programming Language", Proceedings of eScience 2008.