Generating Embedded Systems Software using a Component Based Development Approach

Dr. Mark Dixon

***Abstract*—This work examines how a predominantly graphical approach to software development, that was designed to be deployment platform agnostic, can be used to target embedded software systems. The general aim of the approach was to provide engineers with a development method that was general enough to be applied across a multitude of problem domains. The development technique employs a component centric approach, in which target platform specifics are hidden from the language design. Deployment specific mapping tools are then used to target each type of system. Embedded software systems however are probably the most demanding type of target system, due to limited resources and lack of software infrastructure support. This paper describes a method of mapping an example component based design to a target embedded system.**

***Index Terms*—software, graphical, component, embedded**

I. INTRODUCTION

Modern software engineering often involves the use of many different development languages, run-time platforms and deployment architectures. The decision of which tools and techniques to be used is often dictated by the nature of the target system. Broadly speaking target systems can be classified as being desktop applications, mobile applications, embedded systems, web applications, and distributed applications including Service Oriented Architectures (SOAs). Many of these systems are multi-tier of course, meaning that many modern solutions are actually a hybrid of the aforementioned categories.

The presence of multiple deployment possibilities has led to fragmented development approaches, not just at a language level but also at a tools level. For example, embedded systems development is vastly different to SOA development in terms of implementation languages, development approaches, tracing, debugging etc. Work was undertaken to provide a single development technique capable of supporting different target platforms. This paper not only discusses the general technique but also extends upon this to examine how an example model can be used to target an embedded system developed in the 'C' programming language [1], which is a very common language used to target embedded systems.

The general development technique, which is currently known as the Razor Development Environment (RDE), consists of a component oriented graphical notation supported by an underlying 3GL type language. The enforced use of a component based structure and strong support for re-use among components allows the majority of a solution to be developed using existing generic components which do not include any target specific information. A small number of platform specific components can then be linked with the main solution to produce a deployable system.

Manuscript received September 10, 2012. Mark Dixon is with the School of Computing and Creative Technologies at Leeds Metropolitan University, England, where he undertakes teaching, research and Ph.D. supervision. email: m.dixon@leedsmet.ac.uk.

DOI: 10.5176/2010-3043_2.3.188

II. THE DEVELOPMENT ENVIRONMENT

A. General Architecture

The RDE provides a selection of tools and techniques that allow the development of software systems using a reusable component based approach. At the core of the environment sits a Document Object Model (DOM) that provides a canonical representation of the application under development. This DOM defines the available components along with how they are configured and connected via their interfaces. It may be populated using a graphical notation, a textual 3GL type language or an XML document. All representations are semantically equivalent; hence DOM contents can be manipulated using any representation, independent of the original input method. Any number of deployment tools can then be developed to produce systems via interrogation of the DOM. A graphical representation of the environment is shown in Fig. 1. This highlights the deployment to embedded system targets.

A developer may work on components with only a limited regard for the architecture on which they are to be deployed, hence most components are generic re-usable objects. Only target specific components need to take into account deployment and domain specifics. Due to loose coupling and strong reusability support, both generic and platform specific components can be easily sewn together to provide a complete solution.

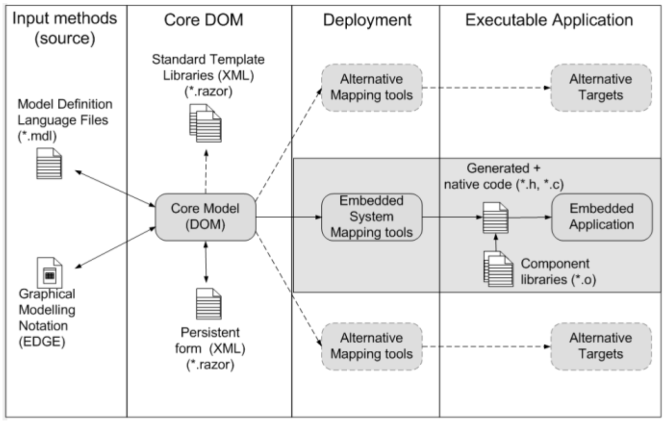

Input methods (source)
Core DOM
Deployment
Executable Application
Standard Template Libraries (XML) (*.razor)
Model Definition Language Files (*.mdl)
Alternative Mapping tools
Alternative Targets
Generated + native code (*.h, *.c)
Core Model (DOM)
Embedded System Mapping tools
Embedded Application
Component libraries (*.o)
Graphical Modelling Notation (EDGE)
Persistent form (XML) (*.razor)
Alternative Mapping tools
Alternative Targets

Fig. 1. The Razor Development Environment (embedded systems)

© 2012 GSTF

The RDE has implicit support for automated testing. The ability to specify compliance tests when defining component interfaces provides support for concepts such as Programming by Contract [2] and Test Driven Development (TDD) [3]. Implicit testing support also allows for better independent development of components, since associating well defined tests with specific interfaces provides a mechanism of ensuring semantic compliance.

B. Design Philosophy

The RDE was designed by taking into account many commonly agreed upon design principles, mainly derived from the Object-Oriented paradigm. Many of these concepts enhance the capability for component re-use, which is fundamental to the RDE approach.

The RDE is extremely *interface centric* in nature. In fact, only interface definitions may represent a type within the system (languages such as Java, C++ and Objective C allow implementation classes to be used as the type of variables and parameters). This design principle is very important for re-use since only interfaces are ever passed as values between component services, also the inheritance model is simplified since implementation inheritance is no longer required. The demotion of the traditional *Class* may seem radical but it allows for much better support of the Open-Closed principle [4] and helps address the well known fragile base class problem [5] since inheritance hierarchies are interface based, rather than implementation based. The Liskov Substitution Principle [6] is also well supported due to the interface centric nature and the ability to ensure semantic compliance.

Systems are defined by identifying component instances and binding them together via their external ports. Each port represents a service, attribute or compound port (i.e. nested ports based on other interfaces). Support for compound ports is a very important abstraction mechanism and allows interfaces which are commonly used together to be wrapped into a single conduit type port.

Each component instance acts as an implementer of one or more interfaces, either directly through terminal ports or by delegation to sub-components. This multi-interface ability supports the Interface Segregation Principle [7] which promotes the idea of providing many fine grain interfaces in order to help reduce dependencies.

The rules that determine the legality of port bindings between components are based on a signature which does not include the name of the port, only the type information. This loosens the coupling somewhat between components, again promoting re-use. Within the RDE lower level components can be connected via ports rather than being embedded within the higher level components. This enhances support for the Dependency Inversion Principle [8] that suggests higher level components should not be dependent on lower level components. The dependency injection pattern [9] is also supported by the binding together of higher and lower level components. The weakened port binding semantics also mean that components can be independently developed, since their visible namespace is only defined within the component itself. Hence, the implementation never needs to be aware of port names that exist outside of the immediate component.

Finally, well known classical design patterns [10] are generally much easier to support in an interface centric component based approach which exhibits low coupling.

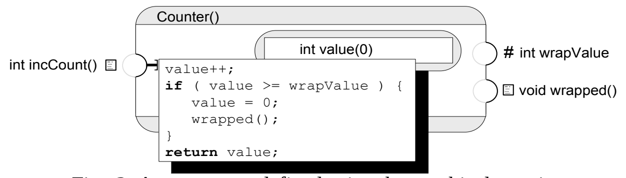
C. The Graphical Notation

One of the primary aims of the RDE was to support a predominantly graphical model based approach to development. Abstracting to a graphical representation not only simplifies development but better supports the ability to hide deployment specifics. It has been suggested that model driven approaches are better placed to deal with the complexities of modern platforms, while also allowing better representation of problem domain concepts [11].

A graphical notation has been defined to allow a declarative style definition of RDE based software systems. This notation, known as (Razor's) EDGE, allows for the construction of an entire system via the use of a single diagram type. Rather than base this on an existing notation, for example by defining a UML profile [12], the decision was made to create the simplest notation possible; while still supporting all required concepts of the underlying DOM.

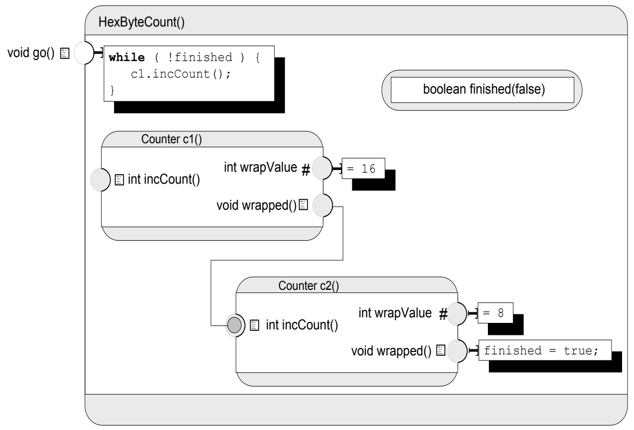
The key elements represented within the notation are interface and component definitions. Both are represented using the same graphical shape, which allows for the use of a single model to define all parts of a system. Both host a number of *provided* and *required* ports, which in essence identifies the direction of dependency when a port is bound. The *nature* of a port, which determines whether it provides a service; stores an attribute; or is compound, is identified using a small icon shown adjacent to the port symbol.

An example of a component is shown in Fig. 2. This defines a simple 'Counter' that counts to a value determined via a *required* attribute port. Once the wrap value is reached the counter resets and a call is made to an outgoing service port. An accompanying example that makes use of the 'Counter' is shown in Fig. 3. This highlights the reuse of other components as parts, and the binding of ports between those parts. Each time the first counter (c1) wraps, it causes the second counter (c2) to be incremented. Once the second counter wraps, it causes the main service to terminate. The first counter wraps on the value of 16, the second counter wraps on the value of 8.



```
value++;
if ( value >= wrapValue ) {
   value = 0;
   wrapped();
}
return value;
```

Fig. 2. A component defined using the graphical notation



```
while ( !finished ) {
   c1.incCount();
}
```

```
= 16
```

```
= 8
```

```
finished = true;
```

Fig. 3. A component with nested 'parts' and port 'bindings'

© 2012 GSTF

Notice that the `wrapped` service port of part 'c1' is bound to the `incCounter` service port of part 'c2'. Although the signatures of these ports vary somewhat, i.e. the `incCounter` returns an integer and the `wrapped` port has a void return type, this binding is permitted and the returned value is simply discarded.

As well as being able to represent the internals of a component using a structural model, components can also be defined as finite state machines. This allows for the definition of control processes which are often used in embedded real-time control systems. Configuration values and configuration types are also provided and used to support customization of components during instantiation. These are analogous to constructor parameters and generic types respectively.

Terminal service implementation code may be defined directly using a simple 'C' like grammar or as *native* code. The built-in language only consists of standard statements and expressions. Support for creating new objects (run-time instantiation) is handled using components, since this is a deployment specific aspect and is not always supported by certain domains such as resource constrained embedded systems.

D. The Model Definition Language

A Model Definition Language (MDL) was developed as an alternative mechanism for populating the RDE DOM. Although regular use of the EDGE graphical notation is the eventual aim, the ability to quickly change the grammar rules within a parser make a 3GL textual language more appealing during the research and development phase.

The MDL provides the same constructs as the graphical notation as keywords within the legal grammar. The language supports definitions of interfaces, components (using the 'implementation' keyword) and binding of ports. The service implementation code uses exactly the same grammar as it does within the EDGE graphical notation; hence MDL is a superset of the imperative code used in the EDGE model. MDL code that is equivalent to the previous example is shown in Fig. 4.

```
implementation HexByteCount {

      provides {
          void go(); // entry point
      }

      parts {
          // Two contained counter part instances.
          Counter c1();
          Counter c2();

          // Boolean type part
          Boolean finished(false);
      }

      bindings {

          // bindings for the out ports of the contained parts
          c1.wrapped -> c2.incCount;
          c1.wrapValue = 16;

          c2.wrapped { finished = true; }
          c2.wrapValue = 8;

          // binding for the provided 'go' port
          go { while ( !finished ) c1.incCount(); }
      }
}
```

Fig. 4. A component defined using MDL based code

A more in-depth discussion of the RDE approach is available [13] and serves as a more verbose source of information for the interested reader.

III. Targeting Embedded Systems

The most obvious way of mapping an RDE defined system to an embedded system is to generate 'C' code from the populated DOM. There are of course alternatives to this including the direct generation of assembly language. However at the moment the generation of 'C' is seen as the best compromise between portability and performance. This does not rule out other techniques however and the fact that the RDE is deployment agnostic means this could be achieved by building alternative mapping solutions. Since 'C' is currently generated, the building of a system includes the use of a traditional tool-chain in order to compile then link the object code.

A. General Considerations

Embedded systems are often constrained in terms of available memory (especially RAM), performance and software infrastructure. i.e. many embedded systems are said to be *bare-metal* in the sense that no operating system is present. These limitations must be taken into account when mapping from an RDE model to implementation code.

Since the RDE is a component based approach in which re-usable elements are sewn together via external ports it is necessary to replicate these relationships in generated systems. When targeting embedded systems it is important that as little configuration information is generated as possible (to reduce overhead) and that this information is stored in ROM whenever possible. There are situations of course when different component instances require their own data, e.g. to store attribute port values. Hence, there is usually a need to divide configuration information between RAM and ROM on the target system.

The code which implements the services must be sharable between all instances once compiled. e.g. if there are ten instances of a particular component the machine code should only exist once, rather than multiple times. Hence the information regarding externally defined ports, which is stored within the generated configuration information, needs to be passed as a parameter to each service. This approach is exactly the same as the passing of the implicit 'this' pointer in languages such as C++ and Java. Also service call overhead should be as minimal as possible, given that the performance level of the target CPU may not be particularly high.

B. Use of Control Blocks

The bulk of the generated code is in the form of *control blocks* which represent the configuration information. In practice these are generated as initialized variable definitions within the 'C' files. It would be also possible to generate the service implementation code directly from a populated DOM, but for now the use of *native* services is assumed. i.e. the service implementation code is provided as 'C' code by the developer. The implementation code of the services interacts with other components by referring to the control block variables declared within the `#include`'d header files.

© 2012 GSTF

Each instance within a populated DOM requires a control block, mainly to represent the external port bindings. Given the aforementioned constraints of embedded systems however it is important to share control block data between instances whenever possible. In order to achieve this the approach taken was to generate common ROM based control blocks, then share these between instance specific RAM based data whenever possible. In order to maximize sharing opportunities any control block data which refers to other control block data does this via relative offsets. The intricacies of why this helps are difficult to explain in such a short paper, but the use of relative offsets means that it is possible to share a much higher proportion of ROM based control block data than would be possible if absolute values were used.

The typical generation pattern used to create embedded system code from a populated DOM model is shown in Fig. 5. A single `system.c` file is generated which contains the initialized control block values for a particular system. This ensures that the code generated for each individual component can be re-used by any system which makes use of those components. In other words, the component code is independent of any particular usage scenario, again promoting reusability even at the generated code level.

C. Mapping the 'Counter' example

In order to help clarify the type of code generated from a populated DOM model, an actual `system.c` file which contains the control block data for the 'Counter' example, is shown in Fig. 6. This has been annotated to identify how relative offsets, rather than absolute values, are used within the generated data. Notice the use of the '`const`' keyword within certain variable definitions. This is to ensure that the linker places the data within a read-only section [14] when generating the final binary, thus placing configuration data into ROM when possible. Although not shown here, a header file for each component is also generated that defines the `struct` types to be populated within the `system.c` file.

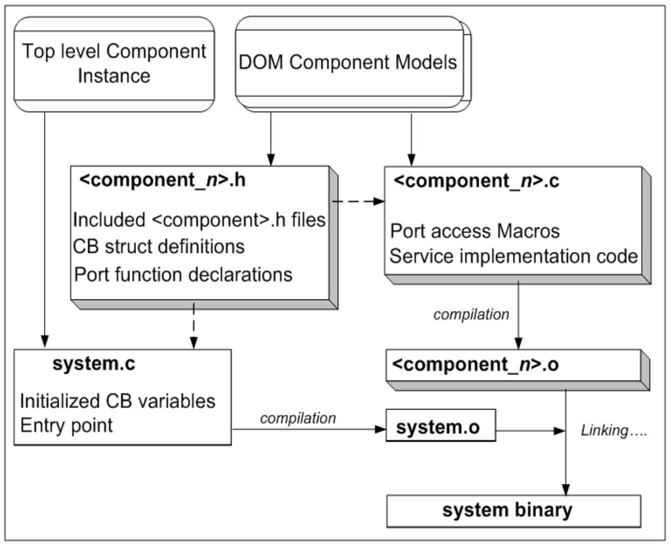
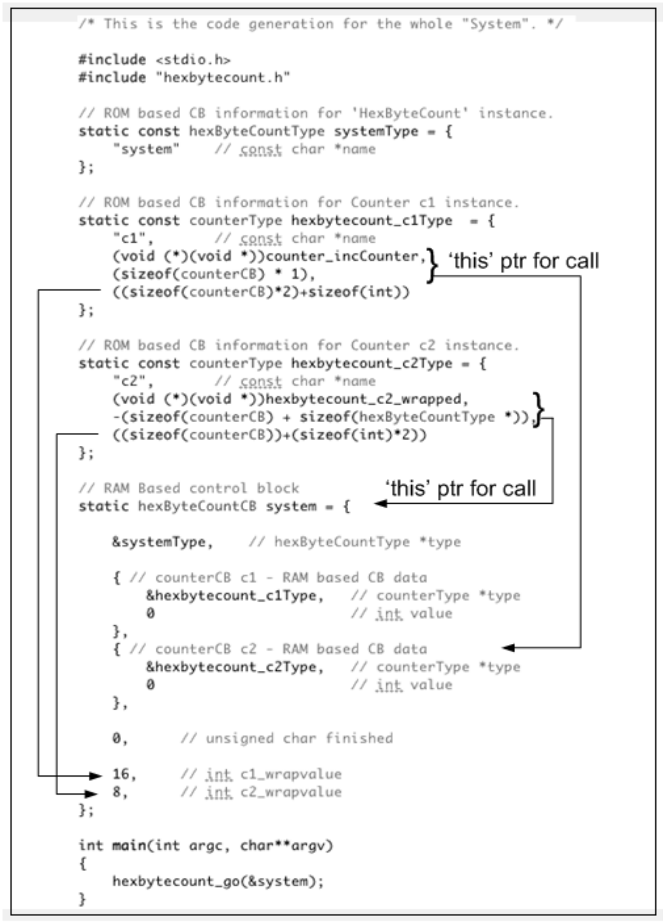


Fig. 5. Generation pattern for embedded systems



```
/* This is the code generation for the whole "System". */

#include <stdio.h>
#include "hexbytecount.h"

// ROM based CB information for 'HexByteCount' instance.
static const hexByteCountType systemType = {
    "system"    // const char *name
};

// ROM based CB information for Counter c1 instance.
static const counterType hexbytecount_c1Type  = {
    "c1",       // const char *name
    (void (*)(void *))counter_incCounter,
    (sizeof(counterCB) * 1),
    ((sizeof(counterCB)*2)+sizeof(int))
};

// ROM based CB information for Counter c2 instance.
static const counterType hexbytecount_c2Type = {
    "c2",       // const char *name
    (void (*)(void *))hexbytecount_c2_wrapped,
    -(sizeof(counterCB) + sizeof(hexByteCountType *)),
    ((sizeof(counterCB))+(sizeof(int)*2))
};

// RAM Based control block
static hexByteCountCB system = {

    &systemType,    // hexByteCountType *type

    { // counterCB c1 - RAM based CB data
        &hexbytecount_c1Type,   // counterType *type
        0                       // int value
    },
    { // counterCB c2 - RAM based CB data
        &hexbytecount_c2Type,   // counterType *type
        0                       // int value
    },

    0,      // unsigned char finished

    16,     // int c1_wrapvalue
    8,      // int c2_wrapvalue
};

int main(int argc, char**argv)
{
    hexbytecount_go(&system);
}
```

Fig. 6. Generated 'system.c' file.

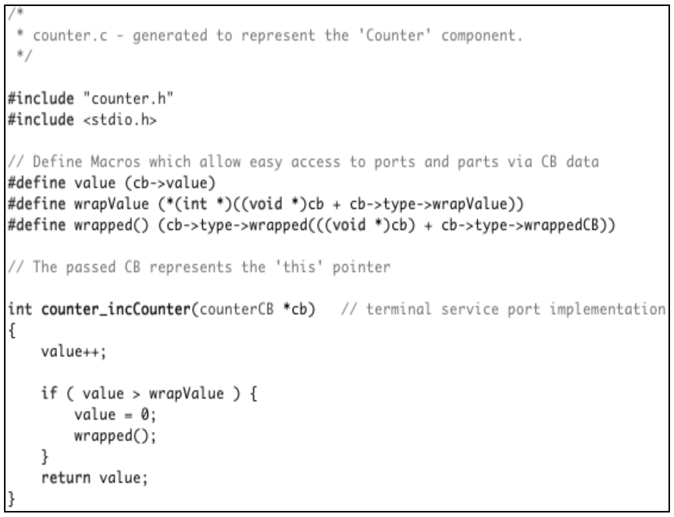
External ports which refer to a service require two entries in the control block. The first is a function pointer to the implementation code, and the second is the relative offset to the control block instance 'this' pointer to be passed as the first parameter.

In this particular example the textual name of each instance has also been included in the control block data. Generally this information would be omitted; however it is useful to produce in cases where a reflective style API is to be supported, or for logging purposes during testing.

D. Accessing External Ports

Providing the control block data is extremely important but it is only part of the final generated system. The service implementation code must know how to use this information in order to access external ports. This could be achieved by generating *wrappers* that provide code that automatically forwards function calls using information obtained from the control block data. This would be fine for many types of system but it does add additional overhead to each function call, which is not a good thing in a resource constrained system. Hence, as an alternative, macro s are generated which provide a more direct route to the external ports. As an example consider the implementation file for the 'Counter' shown in Fig. 7. By examining the defined macros the use of the control block data can be seen. Notice how offsets are added to the local control block value. This is due to the references being stored in a relative fashion.

© 2012 GSTF



```
/*
 * counter.c - generated to represent the 'Counter' component.
 */

#include "counter.h"
#include <stdio.h>

// Define Macros which allow easy access to ports and parts via CB data
#define value (cb->value)
#define wrapValue (*(int *)((void *)cb + cb->type->wrapValue))
#define wrapped() (cb->type->wrapped(((void *)cb) + cb->type->wrappedCB))

// The passed CB represents the 'this' pointer

int counter_incCounter(counterCB *cb)   // terminal service port implementation
{
    value++;

    if ( value > wrapValue ) {
        value = 0;
        wrapped();
    }
    return value;
}
```

Fig. 7. Generated ‘counter.c’ file.

IV. Related Work

This work was originally derived from the development of a configuration tool designed to support the use of an Autosar [15] compliant embedded operating system [16]. Hence this work has its roots within the embedded systems domain. Since then it has evolved considerably however, which has brought it more into line with other component based technologies. These have been well established for many years of course. Technologies such as DCOM [17], Corba [18] and JavaEE [19] tend to focus on higher level distributed components however. Hence, they represent a different layer in the software stack than the RDE which constructs systems from components but does not necessarily deploy them in a distributed fashion. OSGi [20] has a little more in common with the RDE since it is not specifically designed to create distributed systems. However it has not been designed to be deployment agnostic and requires a very specific run-time environment.

In many ways the RDE is similar to the Fractal Project [21]. This too is a component based approach in which models can be graphically defined. An Architectural Description Language [22] is supplemented with C++ or Java in order to build a full system. Although there are similarities they are also differences regarding environment semantics and the graphical notation. Also the bindings between components within Fractal can be representative of higher level network connections etc.

The RDE is in effect an instance of the Model Driven Engineering (MDE) approach. Hence, from a modeling point of view the UML and related MDA technologies [23] cannot be ignored. The model driven paradigm as defined by the Object Management Group (OMG) aims at providing platform independent models which are mapped to platform specific models using transformation rules rather than the direct generation approach taken within this paper.

In terms of purely graphical development of software systems, the MIT App Inventor software [24] and its underlying technologies seem to be based on similar concepts to the RDE. However, the graphical aspects of the RDE are provided to support an architectural level of design, whereas the App Inventor supports graphical design of the user interface along with the procedural aspects of the system. Hence the RDE EDGE notation provides a higher level of abstraction with the lower level imperative code being defined using a 3GL type language (MDL).

The fundamental difference between the RDE and the related work is that its primary aim is to provide a single format that allows for the definition, exchange, and deployment of components which when combined can be used to create software solutions for a diverse set of application domains. In many respects it is synonymous to the philosophy that drove the development of the XML, but instead of being data centric, it is behavioral centric in nature. The tools and techniques developed within the RDE are all specifically designed to provide a realization of this core concept. This paper has focused upon only one of many possible deployment targets.

V. Evaluation and Conclusions

The code produced for the example ‘Counter’ system discussed within this paper compiles and executes as expected. Examination of the generated binaries shows that control block data was indeed placed in a read-only section of the produced binary, as desired. Now the mapping rules for an embedded target have been defined the entire process, including the automated generation of service implementation code, needs to be completed prior to full evaluation.

With regard to the RDE itself, a Java based implementation of the DOM and MDL parser tools has shown the whole concept to be a viable approach. Java was chosen to build the development tools since these have been produced with a commercial strength product in mind. Hence the ability to easily integrate the tools into an industry standard IDE such as Eclipse [25] was seen as paramount.

Although a full implementation of the EDGE notation is not yet complete a prototype has been developed as an Eclipse based GEF [26] dependent plug-in. At the moment this is not yet functional enough to support development of test systems, thus all current evaluation has been done using the purely textual based language (MDL).

The development of several test systems (including the simple counter example described in this paper) has gone some way to validating the supported constructs along with the mechanics of the approach. This work has also shown however that development via the textual based language alone is a fairly difficult process when compared to traditional OO based programming in languages such as Java or C++. This appears to be due to the fact that the RDE was always designed to be predominantly graphical in nature; hence using a textual language to define the architectural properties of a system sometimes seems counterintuitive.

The development of the supporting tools has identified that there is a large amount of complexity involved in ensuring that bindings between compound ports are configured correctly during target system generation. The ability to pre-examine models and produce optimized compiled code is likely to reduce this problem in the future.

Once a full set of development tools are available the system will need to be more thoroughly evaluated via the production of some industrial strength solutions. The available set of tools will be applicable to the development of all Razor based systems irrespective of the target platform. Only the deployment specific compilers or run-time environments need to be target aware.

© 2012 GSTF

VI. Future Work

There needs to be more work undertaken on creating both generic and deployment specific components. When considering embedded systems the production of a typical TCP/IP network protocol stack [27] is an initial priority since such components are not only commonly required, but are also likely to work well in a component based environment due to their *stack* based design philosophy. i.e. a software stack is effectively a vertical set of components connected via well defined interfaces. The development of components to represent other aspects typically present within embedded systems, such as vector tables and interrupt handlers, is also an area to be worked upon in the near future.

Once an appropriate library of components is available the tool can be released to a wider audience in order to gather feedback. The creation of an open source tool chain, along the same lines as the GNU GCC project [28], would help maximize availability of the proposed approach and provide a low cost of entry for prospective developers.

The compilation of Razor compliant components into native machine code via ARM based assembly language [29] is currently being worked upon. The ARM architecture has been chosen due to its high levels of popularity within the embedded systems market along with its powerful addressing modes. These abilities make it possible to manipulate control block information in an efficient manner, thus reducing overhead.

References

[1] B.W. Kernighan and D. Ritchie, The C Programming Language, 2nd Edition. Prentice Hall, 1988.

[2] B. Meyer, Applying "Design by contract," *Computer (IEEE)*, vol. 25, issue 10, October, 1992, pp. 40–51, doi:10.1109/2.161279.

[3] K. Beck, *Test-Driven Development by Example*. (The Addison-Wesley Signature Series), Addison Wesley, 2002.

[4] B. Meyer, *Object Oriented Software Construction*. Prentice Hall, p 23, 1988.

[5] L. Mikhajlov and E. Sekerinski, “A study of the fragile base class problem,” *in Proc. ECOOP'98 - 12th European Conference on Object-Oriented Programming*, Brussels, Belgium, 1998, pp 355-382.

[6] B. Liskov and J. Wing, “A behavioral notion of subtyping,” *ACM Transactions on Programming Languages and Systems* (TOPLAS), vol 16, issue 6, November, 1994, pp. 1811 - 1841.

[7] R. Martin, *The Interface Segregation Principle*, C++ Report, August, Available: www.objectmentor.com/resources/articles/isp.pdf, 1996.

[8] R. Martin, *The Dependency Inversion Principle*, C++ Report, May, Available: www.objectmentor.com/resources/articles/dip.pdf, 1996.

[9] M. Fowler, *Inversion of Control Containers and the Dependency Injection Pattern*, Available: http://martinfowler.com/articles/injection.html, Jan 2004.

[10] E. Gamma, R. Helm, R. Johnson and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

[11] D. C. Schmidt, “Model Driven Engineering,” *Computer (IEEE)*, vol. 39, issue 2, February, 2006, pp. 25-31.

[12] J. Rumbaugh, I. Jacobson and G. Booch, *The Unified Modeling Language Reference Manual*, 2nd Edition. Object Technology Series, Addison Wesley, Chapter 12, 2004.

[13] M. Dixon, “Supporting component oriented development with reusable autonomous classes,” *ARPN Journal of Systems and Software*, vol.1, no.5, August 2011, pp. 182-193, ISSN 2222-9833.

[14] M. Barr, *Programming Embedded Systems in C and C++*, O’Reilly,1999, Chapter 3.

[15] AUTOSAR Partnership, Automotive Open Systems Architecture, Available: http://www.autosar.org, 2003.

[16] K. Tindell and M. Dixon, ‘Scalios’, A scalable Real-Time Operating System for resource-constrained embedded systems, computer software. Published by JK Energy Ltd. 2008. Available: https://github.com/jkenergy/scalios/

[17] T. L. Thai, *Learning DCOM*, O’Reilly Media, 1999.

[18] Object Management Group, *Common Object Request Broker Architecture (CORBA)*, Available: http://www.omg.org/spec/CORBA, 1997.

[19] J.Farley and W.Crawford, *Java Enterprise in a Nutshell*, O’Reilly Media, 3rd Edition, 2005.

[20] OSGi Alliance, “OSGi Service Platform Release 4,” OSGi Alliance Specifications, Available: http://www.osgi.org/Specifications/HomePage, 2009.

[21] OW2 Consortium, The Fractal Project, Available: http://fractal.ow2.org, 2009.

[22] M. Leclercq, A.-E. Ozcan, V. Quéma and J.-B. Stefani, “Supporting heterogeneus architecture descriptions in an extensible toolset,” *in Proc. 29th International Conference on Software Engineering (ICSE 2007)*, Minneapolis, MN, USA, May 2007.

[23] A.Kleppe, J.Warner and W.Bast, MDA Explained: *The Model Driven Architecture: Practice and Promise*. Object Technology Series, Addison Wesley, 2003.

[24] D.Wolber, H.Abelson, E.Spertus and L.Looney, *App Inventor*, May 2011. O’Reilly.

[25] The Eclipse Foundation, *The Eclipse IDE*, Available: http://www.eclipse.org/, 2012.

[26] The Eclipse Foundation, *The Graphical Editing Framework (GEF)*, Available: http://www.eclipse.org/gef, 2012.

[27] K. Fall and R. Stevens, *TCP/IP Illustrated: Volume 1*, Second Edition, Addison Wesley, 2011.

[28] Free Software Foundation, *GCC*, the GNU Compiler Collection, Available: http://gcc.gnu.org/, 2011.

[29] A.Sloss, D.Symes and C.Wright, *ARM System Developer's Guide: Designing and Optimizing System Software*, Morgan Kaufmann, 2004.

About the author:

Mark Dixon achieved a B.Sc. in Computing majoring in Software Engineering (1994) followed by a Ph.D. in Computer Science (1997). His employment history includes working as a software engineer for Dabs Press (now Dabs.com); an embedded software engineer at Taylor Nelson Sofres (TNS - London); a real-time systems software engineer at Live Devices (York); and as a consultant in software engineering and model based development. Current research interests include software engineering, model based development and embedded systems development. Dr. Dixon is currently employed at Leeds Metropolitan University in England and is responsible for Ph.D. supervision along with various teaching and research activities.

© 2012 GSTF