ERTSAL: A Prototype of a Domain-Specific Aspect Language for Analysis of Embedded Real-Time Systems

William Sousan
Technical Support Inc.
wsousan@mail.unomaha.edu

Victor Winter, Mansour Zand, Harvey Siy
University of Nebraska at Omaha/University of Nebraska at Omaha/University of Nebraska at Omaha
{vwinter, mzand, hsiy }@mail.unomaha.edu

Abstract

A primary characteristic of Embedded Real-Time Systems (ERTS) is the fact that they are resource constrained. Such constraints present unique challenges to the embedded systems programmer who must develop software satisfying a given set of functional requirements while simultaneously addressing the limitations of available resources and dependability concerns.

This paper describes ERTSAL – a domain-specific aspect language suitable for use by embedded systems software developers that is comprised of domain specific instructions for use in the monitoring, evaluating, and debugging of ERTS. ERTSAL abstractions shield developers from the intricacies of AspectC++ and the idiosyncrasies of an underlying RTOS. The semantics of ERTSAL is defined in terms of AspectC++. ERTSAL aspects are automatically transformed to corresponding AspectC++ aspects using the transformation system HATS.

Categories and Subject Descriptors D.2.3 [*Software Engineering*]: Coding Tools and Techniques

General Terms Performance, Design, and Reliability

Keywords ERTSAL, Domain-Specific Aspect Language, Program Transformation, HATS,Embedded Real-Time Systems

1. INTRODUCTION

The embedded system market represents the dominant application domain for microprocessors. Of the eight billion microprocessors manufactured in 2000, 98% went into embedded systems [9]. Presently, embedded systems span a broad spectrum of applications ranging from anti-lock braking systems, flight control systems, medical systems, as well as numerous household appliances.

The hardware used in embedded systems is typically heavily constrained. Economic forces demand that the most economical hardware configuration be developed. Beyond this, physical limitations place strict bounds on various hardware attributes such as volume, weight, and power usage. As a result, the computational capabilities of hardware found in embedded systems are typically task specific and narrowly focused. In other words, embedded systems hardware typically do not provide the type of general purpose computing capabilities found on a PC. The software development cost function is further amplified by strenuous dependability constraints that oftentimes accompany embedded applications.

Aspect-Oriented Software Development (AOSD) [5], which is based on the separation of cross-cutting concerns in order to remove tangled and scattered code into their own components, has shown to be helpful in the development of embedded real time systems (ERTS). Some examples of this are in the use of AOSD in embedded databases [12], Operating Systems [9], and Real-Time Component Modeling [1]. All these systems make use of a general programming language (GPL) that is enhanced for AOSD. Languages such as AspectJ [8] and AspectC++ [11] are examples of GPL's that are modified to include support for aspect definition and weaving. However, these languages lack support for domain specific features of the problem domain. This presents the need for a Domain-Specific Aspect Language (DSAL) [10] which can help encapsulate the characteristics unique to a particulcar problem domain. Specifically, a DSAL that focuses on the ERTS development domain could help improve the development phase by offering the ability to monitor power consumption, execution time, memory usage, error handling, and reliability.

In this paper we introduce a prototype language, known as ERTSAL, which is a domain-specific aspect language for the development of ERTS. As of this writing, we believe our DSAL to be the first of its kind to be used for ERTS development during the implementation phase. Other systems such as VEST [7], are used for Real-Time systems during the design phase. The ERTSAL language provides domain specific aspects for ERTS that are expressed in less lines of code as compared to their AspectC++ counterparts. In addition, the ERTSAL instructions are easier to understand and are customized for the monitoring, evaluating, and debugging of ERTS.

The remainder of this paper is as follows: Section 2 explains the steps leading towards the development of a DSAL for ERTS along with the tools and methodologies for its development. Section 3haks summarizes the syntax and semantics of ERTSAL as well as its implementation. Section 4 describes the results of using ERTSAL aspects to monitor and constrain embedded system software. Section 5 discusses areas of future work, and Section 6 concludes.

2. Background Research

The research was initiated by reviewing the difficulties of ERTS development by evaluating a wide spectrum of embedded systems with varying degrees of constrained resources. This included high-end distributed Real-Time systems all the way down to small deeply-embedded systems that are extremely resource constrained. By doing so, it allowed for the characterization of the challenges

DSAL '07 March 12, 2007 Vancouver, British Columbia, Canada.

that are unique to ERTS development as well as the research methods for solving those challenges

The development of ERTSAL is motivated by three goals: The first goal concerns itself with the identification of a set of domain specific cross-cutting concerns S_A central to ERTS software development. The second goal is to provide a language (i.e., ERTSAL) in which the cross-cutting concerns in S_A can be described in a straightforward fashion. The third goal concerns itself with developing an infrastructure in which ERTSAL programs can be incorporated into the ERTS development process. General purpose aspect oriented languages such as AspectC [4], AspectC++, and AspectJ with Real Time extensions can address the third development goal, but do not inherently address our first and second goals. This suggests that a general purpose aspect oriented language could be used as a back-end of an ERTSAL prototype. This observation has led to the development of an architecture in which AspectC++ serves as the general purpose aspect oriented language comprising the back-end of ERTSAL. AspectC++ was chosen because it seemed best suited to ERTS development due to its availability and low overhead. After the selection of AspectC++, we reviewed several case studies describing how aspects are used with embedded systems such as with RTOS configuration [9], WCET analysis [1], database configurations [12] and others. In order to address the first two development goals of ERTSAL, the advantages of Domain Specific Languages were examined with the goal of integrating domain specific ERTS concepts into the aspect oriented idiom. We analyzed a select group of DSAL's [3], [13], [6], that appeared to have beneficial features and characteristics for ERTS development. This provided a foundation to proceed with a prototype DSAL tailored for ERTS.

3. Prototype System

We have developed a prototype DSAL called ERTSAL in which the benefits of a DSAL for ERTS can be evaluated. This system allows us to analyze the lines of code savings and expressiveness of the language as compared to general purpose programming languages with aspect extensions. Since the focus of the research is more on the development of the actual domain specific language, AspectC++ is used for defining the aspects and weaving to avoid the overhead of creating a new aspectual language. By doing so, it limits the work to defining the DSAL and developing a compiler, which is preprocessor based, that converts the DSAL code to AspectC++ code and then finally to ANSI C++ that can be compiled with a standard C++ compiler. To summarize, the front end of the prototype consists of a BNF grammar describing ERTSAL together with an ERTSAL-to-C++ compiler. The back end of ERTSAL is provided by AspectC++. Our prototype provides a translator from the ERTSAL syntax to a corresponding implementation.

ERTSAL aims to provide C++ software developers with a set of abstractions appropriate for expressing a variety of cross-cutting concerns central to embedded real time systems. The abstractions provided shield the developer (to some extent) from the intricacies of AspectC++ and the idiosyncrasies of the underlying real time operating system. ERTSAL encompasses the majority of the pointcut language (e.g., all forms of wildcard symbols and algebraic operators) of AspectC++ [11]. ERTSAL aspects are categorized by the type of their behavior (**monitor** or **constrain**) and by the type of their concern (dependability or RTOS). Aspects whose behavior is of type **monitor** are used to formulate crosscutting concerns that monitor specific behavioral quantities and report results. Reporting is accomplished using the built-in ERTSAL operator **LOG**. Aspects whose behavior are of type **constrain** are used to formulate crosscutting concerns that alter the behavior of program execution. In its present form, ERTSAL is probably best classified as a library of domain specific aspects. Aspects belonging to this library can be viewed as function-like entities whose formal parameters include pointcuts and whose syntax has a command-line flavor. Concrete examples of ERTSAL aspects are given in Table 1.

Presently, the transformation from ERTSAL to AspectC++ is accomplished in a single rewriting step. That is, no intermediate forms are generated during the transformational process. The transformation from ERTSAL to AspectC++ is performed using the transformation system HATS [14]. Figure 1 shows the system build process that should be followed to produce an executable program.

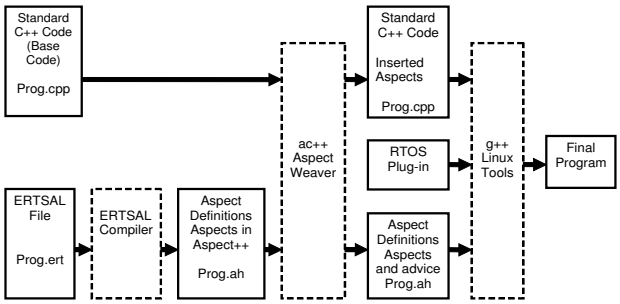

Figure 1. ERTSAL Build Process

Note that a framework is provided in conjunction with the ERTSAL compiler to create an RTOS independent system. This framework provides for RTOS plug-ins that allow for the specification of an interface file which describes mappings from the ERTSAL RTOS functions to their corresponding RTOS specific functions. By doing so, the ERTSAL compiler can be used with any RTOS that can provide an interface file describing its function mappings to the corresponding ERTSAL compiler functions. In addition, the interface file also supports an option for defining the type of communication channel used for reporting the output generated from the ERTSAL instructions. This allows for any of the common communication channels to be used such as serial, parallel, USB, or Ethernet.

4. Evaluation

In order to evaluate the effectiveness of ERTSAL, a test program was created that included enough functions and methods so that each of the ERTSAL commands could be exercised. Likewise, an ERTSAL file that contained a line for each possible aspect was written so that each instruction could be tested. Next both files were put through the build system in order to create the final executable. Finally, the executable was run on the target system and the evaluation continued with verification of the ERTSAL commands. Note that a few iterations of this process were required so that the base and aspect code could be modified enough times in order to verify the operation of each ERTSAL command. In its current form, the main advantage of ERTSAL is that it allows for the specification of aspects that are domain specific to the ERTS domain. This facilitates the ease of defining domain specific aspects that otherwise have to be defined by a general purpose language that requires more lines of code. Table 2 highlights the amount of the code savings by showing the lines of code in AspectC++ that are needed for selected ERTSAL instructions. Figure 2 displays the corresponding AspectC++ code that is used to implement the cpu_time aspect.

There are restrictions in the tools ability to indicate overloaded functions. For example, if the function specified in a target function parameter belongs to a group of overloaded functions, the corre-

monitor cpu_time (2000) execution(“% C::TimeTest(...)”); This aspect monitors the amount of time a function takes to execute. The parameter (2000) denotes the maximum time, in microseconds, allowed for the execution of functions matching the pointcut “% C::TimeTest(...)”.
monitor intra_cpu_time (2000) execution(“% C::IntraTimeTest(...)”); This aspect monitors the amount of time that passes between sequential occurrences of function executions matching the pointcut execution(“% C::IntraTimeTest(...)”). The parameter (2000) denotes the maximum allowable time, in microseconds, between sequential occurrences of the selected function executions. If this threshold is exceeded an error will be logged.
monitor power_level (75) execution(“% MinPwrClass::%(...)”); This aspect monitors the power level of the system. The parameter (75) is a threshold denoting a power level percentage. For every join point matching execution(“% MinPwrClass::%(...)”) the system power is checked. If the available power drops below the threshold (i.e., 75%) an error will be logged.
monitor power_level LOG execution(“% PwrClass::%(...)”); This aspect monitors and reports the power level of the system for every join point matching execution(“% PwrClass::%(...)”).
monitor memory_usage LOG call(“% C::MemoryWatch(...)”); This aspect monitors and reports the memory usage of the system for every join point matching call(“% C::MemoryWatch(...)”).
monitor object_count LOG construction(“C”) || destruction(“C”); This aspect monitors the construction/destruction of join points respectively matching construction(“C”) and destruction(“C”). A global count is maintained of the number of these objects in existence and is reported whenever a join point matches construction(“C”) || destruction(“C”). At present, the object count can be monitored for only one class per execution.
constrain return_time (-1, 3000) execution(“int C::TRTest(...)”); This aspect constrains the execution of every method whose jointpoint matches execution(“% C::TRTest(...)”). In this example after the target program has executed for more than 3000 microseconds, the execution of all methods matching execution(“int C::TRTest(...)”) will return the value -1.
monitor joinpoint LOG call(“% C::TraceMe(...)”); This aspect monitors and reports calls to methods whose join points match the pointcut call(“% C::TraceMe(...)”).
monitor null_ptr LOG call(“% C::PointerTest1(...)”); This aspect monitors the actual parameters that are passed to every method whose join point matches the pointcut call(“% C::PointerTest1(...)”). For every actual parameter whose value is NULL and error report is logged.

Table 1. ERTSAL Commands

ERTSAL Concern	LoC in AspectC++	eLoC in AspectC++
cpu_time	12	6
intra_cpu_time	17	9
memory_usage	7	3
joinpoint	7	3
object_count	18	10
return_time	11	5
null_ptr	21	11

Table 2. Lines of Code Comparison between ERTSAL and AspectC++

sponding ERTSAL instruction will be applied to the entire group of overloaded functions. Presently there is no means to select an individual function from a group of overloaded functions.

5. Future Work

In the present version of ERTSAL, the developer is provided with a fixed set of pre-defined aspects that are parameterized on pointcuts, thresholds, and reporting logs. We are currently expanding ERTSAL to allow developers more flexibility with regard to the definition of aspects. The goal is to extend ERTSAL in a manner that remains mindful of and consistent with the principles underlying domain specific languages.

Another area of interest is providing support for the comprehension of the interaction between ERTSAL aspects and the target C++ application. Presently there is no support with the ERTSAL system for indicating which aspects have been applied to which methods/classes. An aspect viewer allowing for a visualization of the links between the aspects and their corresponding join points would be helpful in this regard.

```
constrain cpu_time (2000) execution(“%Cbclass::TimeTest(...)”);
⇒
aspect MaxCPUTime2
{
   advice execution (“%Cbclass::TimeTest(...)” ) : around()
   {
      long start_time = get_elapsed_microsecs();
      tjp - > proceed();
      if ((get_elapsed_microsecs() - start_time) > 2000)
      {
         output(“Timeout > Method-Function Called: %s \n”,
            JoinPoint::signature() );
      }
   }
} ;
```

Figure 2. ERTSAL cpu_time to AspectC++ translation

6. Conclusion

The contribution of this research is a new DSAL that is tailored for ERTS development. It provides customized aspects that are domain specific to ERTS development for the use of monitoring, evaluating, and developing ERTS. In addition, it simplifies the process of defining domain specific aspects by providing them in the form of instructions within a DSAL. This allows for the reduction of aspect definitions and thus less code is needed to describe these aspects. Note that reducing code also helps in minimizing the chances of error as well as requires fewer lines to write or understand, and, according to Boehm’s [2] maintenance cost prediction model, low-

ers the maintenance costs. In addition, the ERTSAL language is easier to read in comparison to its non-DSAL version. In conclusion, our vision is to add more domain specific instructions and enhancements to grow this new language for commercial use of ERTS development.

References

[1] D. N. C. N. A. Tesanovic, J. Hansson and P. Uhlin. Aspect-Level WCET Analyzer: a Tool for Automated WCET Analysis of a Real-Time Software Composed Using Aspects and Components. In *3rd International Workshop on Worst-Case Execution Time Analysis (WCET 2003) in connection with 16th IEEE Euromicro Conference on Real-Time Systems (ECRTS04), Porto*, July 2003.

[2] B. Boehm. *Software Engineering Economics*. Prentice-Hall, 1981.

[3] M. Bruntink, A. van Deursen, and T. Tourwé. Isolating idiomatic crosscutting concerns. In *Proceedings International Conference on Software Maintenance (ICSM 2005)*, pages 37–46. IEEE Computer Society, 2005.

[4] Y. Coady, G. Kiczales, M. Feeley, and G. Smolyn. Using AspectC to Improve the Modularity of Path-Specific Customization in Operating System Code. In *Joint European Software Engineering Conference (ESEC) and 9^{th} ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE-9)*, pages 88–98. ACM Press, 2001.

[5] T. Elrad, R. E. Filman, and A. Bader. Aspect-oriented programming: Introduction. *Commun. ACM*, 44(10):29–32, 2001.

[6] P. Fradet and M. Suedholt. An Aspect Language for Robust Programming. In *AOPWorkshop at ECOOP '99*, 1999.

[7] R. P. C. Z. Y. M. H. J. Stankovic, R. Zhu and B. Ellis. Vest: An aspect-based composition tool for real-time systems. In *IEEE Real Time Technology and Applications Symposium*, pages 58–69, May 2003.

[8] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. Getting Started with ASPECTJ. *Commun. ACM*, 44(10):59–65, 2001.

[9] D. Lohmann, W. Schröder-Preikschat, and O. Spinczyk. On the Design and Development of a Customizable Embedded Operating System. In *Proceedings of the International Workshop on Dependable Embedded Systems, 23rd Symposium on Reliable Distributed Systems (SRDS 2004)*, October 2004.

[10] K. Mehner and A. Wagner. An Assessment of Aspect Language Design. In *Position Paper Young Researcher Workshop, GCSE'99.*, 1999.

[11] O. Spinczyk, D. Lohmann, and M. Urban. AspectC++: an AOP Extension for C++ . *Software Developer's Journal*, pages 68–76, May 2005.

[12] A. Tesanovic, K. Sheng, and J. Hansson. Application-Tailored Database Systems: a Case of Aspects in an Embedded Database. In *Proceedings of the 8th IEEE International Database Engineering and Applications Symposium (IDEAS'04), IEEE Computer Society*, July 2004.

[13] Y. Usui and S. Chiba. Bugdel: An Aspect-Oriented Debugging System. In *Software Engineering Conference, 2005. APSEC '05. 12th*, pages 790–795, Dec 2005.

[14] V. Winter and J. Beranek. Program Transformation Using HATS 1.84. In *Generative and Transformational Techniques in Software Engineering (GTTSE)*, volume 4143 of *LNCS*, 2006.