Glue Code Generation: Closing the Loophole in Model-based Development

Dionisio de Niz and Raj Rajkumar
dionisio@cs.cmu.edu, raj@ece.cmu.edu

***Abstract*— Embedded real-time systems are tightly coupled with the physical world. This tight coupling imposes para-functional requirements (such as timeliness, jitter, fault-tolerance, and security) that go beyond functional (logical) behaviors. We have developed a Model-based embedded system development tool called Time Weaver [3]. Our tool provides new abstractions that enable the modeling of both functional and para-functional behaviors into separate views whose interactions are automatically handled. These separate views focus on a single concern enabling multiple domain experts (e.g. signal processing experts, control experts, real-time experts, fault-tolerance experts) to focus on the concern of their expertise while the interactions with the other views are automatically handled. However, in order to be effective, model-based technology must take the design decisions down to the running code. As a result, automatic glue code generation to integrate the final running system must be an integral part of any model-driven technology. Furthermore, given that the timing model should reflect the behavior of the running code, both technologies must be properly integrated. In this paper we present the glue code generation framework of Time Weaver and its timing model. We discuss the challenges faced by a code generation framework for embedded real-time system which includes multiple operating systems, multiple communication mechanisms, different hardware sizes (from full computers to microcontrollers), and dynamic structures of the software.**

I. Embedded System: Beyond Software

Distributed real-time embedded (DRE) systems work in close concert with the physical world around them. Examples of embedded real-time systems and their para-functional requirements include: (a) Cruise control systems that need to sample the vehicle speed at regular intervals; (b) Antilock Braking Systems (ABS) whose modules are replicated to tolerate the failure of one or more of the replicas; and (c) video-conferencing systems that need to maintain a regular frame-rate.

The strong relationship between embedded software and the physical world imposes new requirements and semantics to embedded software. For instance, if the system requires an automatic response to a physical phenomenon (e.g., tire skidding in an Antilock-Braking System (ABS)), the time to execute such a response needs to be calculated and verified against the response time required for such interaction. This execution time implies that the system model should also include a model of the hardware (e.g., processor speed, memory size, network connections) where the software is executing along with the binding between the hardware and software elements. Programming languages have traditionally focused on the syntactic description and verification of software logic. Para-functional behaviors are encoded in (spaghetti-like) interleaving fashion into the code. Unfortunately, these interleaving makes the code difficult to verify, modify, or re-use. The clear solution to this increasingly vexing problem is the use of model-based development. In addition, in the development of DREs multiple domain experts focus on the different aspects of the system. As a result, an important factor in model-based development is the separation of these concerns into different view focused on the concerns of the different experts. Furthermore, to preserve the work of each of the experts, the interactions among these views need to be automatically handled.

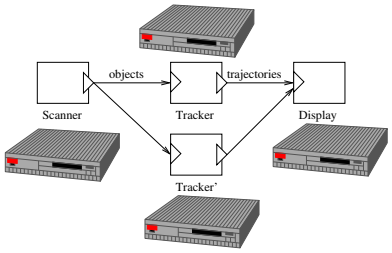

Fig. 1. Replicated Radar System

A. The Need for New Inter-Module Relationships

In traditional programming languages, programs are composed by software modules integrated through an invocation scheme. This scheme builds both an execution order and a data dependency relationship. For instance, in the statement `z = f(g(x));`, the relationships between `f()` and `g()` are two-fold. First, the execution order specifies that `f()` should be executed after `g()`. Secondly, the data dependency specifies that `f()` use the data generated by `g()`. However, an order of execution is often imposed even when that order is irrelevant to the program. For instance, in the statement `z = f( g(x), h(y) );`, whether `g()` is executed before or after `h()` is irrelevant, but a typical compiler generates a specific order.

When para-functional properties are involved, a richer set of relationships other than the data dependency and the execution order offered by the invocation scheme needs to be defined. For instance, consider the example depicted in Figure I-A. In this example, the replication of the `Tracker` module involves new relationships that must be satisfied among these replicas: (a) the replicas must execute in different processors, (b) they must see the same `Objects`, and (c) the `Display` module must see a single consistent set of `trajectories` from the replicas. This set of new relationships must be enforced to provide a specific reliability level for the replicated module. As new para-functional requirements are added, inter-module relationships grow in number and complexity. Examples of other relationships are communication protocols, mutual exclusion, response time, etc. These relationships must all be *simultaneously* captured and properly integrated into the final system.

B. Couplers: Capturing Functional and Para-Functional Concerns

Our framework uses a new abstraction called *Coupler* to capture relationships among components. Components are event-processing modules that communicate through ports and couplers, manipulate code elements inside the components ports to build the different types of relationships.

Inside a component, the application code is localized in an element called an *application agent*. An application agent is structured as a set of functions where each function processes a type of event and generates a set of events. Components contain additional elements

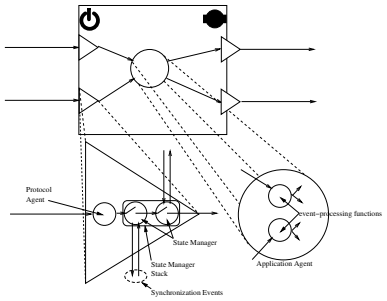

Fig. 2. Component Structure

that support the construction of para-functional behaviors. The organization of the elements inside a component is depicted in Figure 2. Ports contain a protocol agent that is responsible for getting events from one component to another. As a result, communication happens from one protocol agent to another. In other words, protocol agents implement the necessary mechanism to achieve the inter-component communication. For instance, if the components are in the same memory space, they can simply use function invocation. On the other hand, if these components are in different processors, they will need an inter-processor communication such as CORBA or TCP/IP sockets. Protocol agents also encode the concurrency elements of the component. This means that each protocol agent can contain a thread of execution that can run concurrently with any previous one before the event was given to this protocol agent. Whether a protocol agent should contain its own thread is specified with a port property (*active*) at design time.

After events are received through the protocol agents, a series of state managers implement the synchronization protocol that may be necessary for different para-functional constructs. For instance, consider the replication example depicted in Figure I-A. In this example, a state manager in the input port of each replica verifies that both replicas receive an event before giving it to the application agent. This implies that such state managers exchange messages that inform the other of the reception of events so each one can verify that an event has been received by both.

Multiple types of couplers capture the different aspects of the system. A set of basic types are defined in our framework, these types are: (a) event couplers, that capture communication relationships, (b) synchronization couplers that specifies a state manager to synchronize the execution of components, (c) parametric couplers that relates component parameters of different components (e.g. same period for a periodic activation), (d) hardware couplers that define the characteristics of the hardware (speed), (e) deployment restrictions that capture restrictions among components to allow or disallowed its joint deployment (e.g. two replicas for fault-tolerance are restricted not to be deployed on the same processors), and (f) communication scopes that specify what communication protocol to use across different elements of the hardware (e.g. in different networks, such as TCP/IP on Ethernet, CANOpen on CANBus, etc.). These different types of couplers can be composed into hierarchies enabling a problem decomposition scheme similar to the functional hierarchies.

C. Semantic Dimensions: Consistent Separate Views

Having one hierarchy per concern greatly assists in the complexity reduction of developing embedded real-time systems. However, if all the hierarchies are presented together, the density of information would defeat the purpose of such a separation. As a result, each hierarchy should be presented separate from each other.

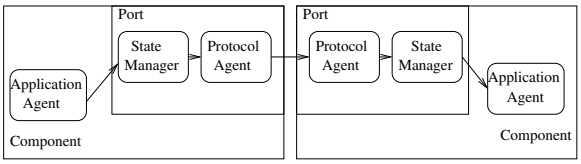

Fig. 3. Application Runtime

Our modeling framework takes a two-fold approach to preserve the value of multiple views in a model. First, a user may modify a view locally without regard to other views. Secondly, the impact of any such changes in other views is accounted for automatically by the framework. In other words, changes in one view are automatically projected to other views. This automatic projection creates stronger semantics than just isolated views. We refer to our views as *semantic dimensions* as we automatically preserve the semantics (intended behavior) of the couplings.

II. Code Generation

Central to our objective of keeping a faithful model of the runtime image is the ability to generate executable glue code. By glue code, we mean the code that combines the application code together to form the final running system.

A. Challenges

Code generation frameworks for embedded real-time systems face multiple challenges. Firstly, in the embedded system arena no single operating system dominates the market. Furthermore, even though OS standards have appeared in this area (e.g. OSEK for the automotive industry) its acceptance would take some time and perhaps the appearance of new standards. Secondly, communications technologies in terms of protocols and links are not standardized and changes depending on the particular application, e.g., CORBA may be used for avionics applications and Flexray may be used in automotive applications. Thirdly, embedded systems are developed in different target languages. For instance, an automotive application is likely to be developed in *C* while an avionics application is developed in *C++* and a robotics application in *Real-Time Java* [2], [1]. Moreover, at different stages of development of an embedded system different languages may be used to evaluate different parts of the system. For instance, signal processing engineers may be interested in generating Matlab® code to evaluate their algorithm while the final system needs to be generated in *C++*. Finally, the behavior of the code must faithfully reflect the timing model used to verify the timing properties of the system.

B. The Code Generation Framework

Our code generation framework is composed of two main parts: an inter-component communication library and the generator of code that performs the inter-component connections and initializes the application.

Our framework generates a runtime application structured as depicted in Figure 3. This runtime structure consists of a network of computing elements connected through a table-driven publisher-subscriber mechanism. In this latter mechanism, one publisher (at an output port) maintains a list of subscribers for each event type it generates. When an event must be generated, the publisher chooses the appropriate list of subscribers based on the type of the event, and delivers the event to all of them.

Three basic types of elements exist within a component: `Application Agents`, `State Managers`, and `Protocol`

`Agents`. The application computation occurs in the application agents which together constitute the overall computations occurring in the system. State managers synchronize these computations by controlling the transit of events through ports. Finally, the protocol agent implements different inter-component communications mechanisms when the communicating components are in different address space, processors, networks, etc.

Paths of execution start at active ports. An active port has a protocol agent with a thread inside. Special *activation* ports can be configured to automatically generate events periodically. Such an event is a system event known as a *tick* that is sent to the application agent to indicate that a new activation event has occurred. The application agent then reacts to this event.

1) Communication Library: The communication library consists of a set of state managers and protocol agents that transport and synchronize the event communications across components. The interface to this library is given by the skeleton of the elements and its publisher-subscriber infrastructure.

The power and extensibility of this library form the basis of the componentization of the para-functional constructs of our framework. In other words, new state managers can be developed to implement new forms of synchronization. Furthermore, the creation of new protocol agents enables new forms of inter-component communication to suit different remote or efficient communications as different networks, middleware, and operating systems are developed.

The publisher-subscriber structure in a protocol agent is updated with the appropriate mechanism used to carry out any communication between components. For instance, if a multicast mechanism is to be used, the protocol agent creates a multicast group for each event type and the receiving protocol agents subscribe to that group. The delivery of the event is then transformed into a single event sent to the multicast group.

2) Code Generation: The glue code consists of two parts: the connection of the subscribers to the publishers, and the initialization of the system.

The publisher-to-subscriber connections are performed as modeled along the different semantic dimensions. These connections build, the data plane to transfer the events associated with the functional model. The connections across the state managers build the control plane to synchronize the concurrent execution of the application agents and to build the para-functional behaviors specified in the model.

The initialization code consists of three parts: setting up the activation parameters of the threads of active protocol agents, creating and starting these threads, and waiting for their completion. The setting up of the activation parameters is done at the creation of the protocol agent objects[1].

3) Supporting Different Execution Platforms: A primary goal of our system is to support multiple execution platforms (e.g. operating system and middleware). Therefore, all the calls to the target platform are hidden inside a library. This library is then changed appropriately when a different platform is desired. As a result, the generated glue code makes calls to such a library that hides the differences among different platforms. However, some execution platforms require programming language extensions. This is the case for the OSEK operating system. OSEK is an OS for embedded systems that can be sized to match the needs of the application. This sizing implies that the number of operating system objects such as processes, semaphores, and messages queues, are limited to only those needed by the application. The end result is that the footprint[2] of the operating system is minimized.

To achieve the sizing of the OS, an OSEK configurator requires a set of declarative statements from the application to calculate the sizes of the OS structures. These statements are given in the target language (e.g. *C*). As a result, the dependencies from this operating system cannot be hidden in a library and therefore requires direct support from the code generator. For such cases, a specialized code generator is used.

4) Multi-Language Code Generation: Our framework supports multiple target languages. The selection of the target language can be different for different parts of an application. In other words, one part of the system can be generated in one language, say `C`, while the other part can be generated in another language such as `Java`. Because the communication across components happens through a well-contained element (protocol agent), the only condition to enable this mixture is to provide cross-language protocol agents.

The multi-language code generator is organized as a collection of code generators that process the components attached to a `Process` deployment coupler. As such, one target language is selected for each of the processes in the system. Given this selection, a main generator traverses the hardware description hierarchy calling different children generators for each process. Code generators can be easily added to the system to extend the set of supported target languages.

To support multiple execution platforms, the selection of these platforms is done on a per processor basis. That is, each processor coupler in the system can have a different selection of the execution platform. As with the target languages, the requirement is the availability of cross-platform protocol agents.

C. Optimizations

Multiple optimizations are possible for the generation of glue code. A simple optimization is the elimination of dead-code elements (state managers and protocol agents) that do not add any value. For instance, state managers for mutual exclusion among components run by threads that never run concurrently can be eliminated. Perhaps the most interesting ones are those that have a compact code representation. Among them we discuss the optimization for Data Flow Components, since dataflow models are commonly used in many application domains such as signal processing.

1) Dataflow Glue Code Optimization: In dataflow systems, components operate over a stream of input data that is transformed into an output stream. This data stream is structured as a sequence of data tokens that are processed one batch at a time. The typical application domain that uses dataflow semantics is signal processing. In this domain, signals are represented as a stream of discrete samples of a continuous signal in the real word, e.g. video. As the signal is being processed, it is passed through several components in a sequence. The processing paths built with these components are typically static. As a result, a static sequence of execution is typically followed and variations to the control flow are seldom used.

a) Execution Sequencing of Dataflow Components.: A dataflow component is executed once all of its expected data parameters arrive. The main input is the sample of the signal being processed and a transformed sample is normally produced. This produced sample is then handed over to the next component which repeats a similar execution pattern.

The execution sequencing of the dataflow component is encoded in our framework with an inter-port coupler. This coupler uses a

[1]Object-like structures are used in non-object oriented target languages.

[2]memory needed by the operating system

state manager that synchronizes the reception of the component input parameters before executing the function. This synchronization is reflected both in the timing model and in the generated glue code. However, the synchronization state manager of the input parameters is eliminated by the *Dataflow* generator. Instead, the natural statement sequence of the target programming language is used. In other words, if component C receives as input data generated by components A and B, then component A and B are called before C in the generated code.

b) Data Stream Representation.: Data streams are represented as pointers to memory locations where data is read or written. The implementation of these pointers depends on the target programming language but the size of the data structure is necessary in all cases. With this information, appropriately sized memory is allocated and its location is passed over to each of the processing functions of different components to read from or write to them.

c) Dataflow Across Communication Scopes: Both the execution sequencing and the data stream representation presented so far assume that the components are located on the same communication scope, i.e., same process. However, when such components are located in different communication scopes, a different approach should be used.

To enable dataflow across communication scopes, the application is structured as a two-layer run-time generation. The top layer generates an event flow run-time with publisher-subscriber connections across components, and the lower layer generates pointer-based data flow connections with an execution schedule that sequences statements appropriately.

The embedding of pointer-based connections inside a composite component with publisher-subscriber connections enables the use of the communication infrastructure designed for event flow. At the same time, the efficiency of pointer based communication is achieve to the maximum extend inside a communication scope.

III. Code Generation Validation

Three experiments were performed to validate our code generation framework. Firstly, we developed two generators for small microcontrollers with embedded operating systems. Secondly, we modeled a large system and use a generator with optimizations to generate the runtime code. Then we compared the efficiency of the generated code against the legacy runtime code. Finally, we use a single software model with three different target languages.

A. Targeting Small Processors

To evaluate our code generation framework for small targets we chose two platforms: an ARM 7 with μCOS-II OS and an MPC555 with an OSEK OS. For the former, we were able to use the same *C* generator that we use in a previous generator for the Linux OS. In other words, all the modification required due to the change of the OS from Linux to μCOS-II were done inside the runtime library.

On the other hand, for the MPC555 running with the OSEK OS, we not only made modification to the runtime library but also created of a new generator. This is because OSEK requires special statements in the program body that need to be customized for each application to be able to define statically static structures of the operating system.

B. Efficiency of Code Generation

To evaluate the efficiency of the generated code we model a software radio[3] system with 700 components and 764 connections across components. We then generated C++ code running on Linux using the optimizations for Dataflow discuss in Subsection II-C. The execution time of the generated code was then compared with the legacy runtime. The generated code showed an improvement of 16.77%.

[3]A system that processes radio signals (modulation, demodulation, etc.) through software.

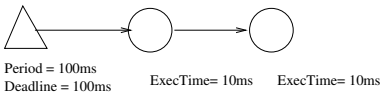

Fig. 4. Simple Response Chain

C. Multiple Target Languages

The software radio system modeled for the code efficiency experiment was also use to evaluate the capacity of the model to generate multiple target languages. In this case we generated not only the C++ code but also Matlab code that the signal processing engineers used to verify the functionality of the system (its ability to decode specific frequencies and encoding schemes). In addition, an XML configuration for a special simulation environment was also generated.

IV. Timing Model

Given that embedded systems can include automatic responses to external phenomena, such as opening a valve based on a pressure reading, deadlines for these responses are common. A response chain model [4] of the system is constructed to verify these deadlines. This model is used by real-time analysis tools such as TimeWiz® [7] to verify that the timeliness requirements are met. The model describes the software as a set of sequences of actions executed by a trigger. Each action has an execution time and whether it needs access to a shared resource (e.g. a mutex). A trigger defines the periodicity (or aperiodic pattern) with which the actions are executed and the deadline, from the beginning of the period, by which their execution must complete. Figure 4 shows one such a response chain.

The hardware is modeled as a graph where the vertices are processors and networks and the edges the connection among them. In this model, networks are considered message processors. In order to target heterogeneous platforms, the execution time is encoded in cycles, and the execution speed of processors is encoded in cycles per second. This encoding enables the analysis of the response time of task sets composed of actions that can be deployed on different platforms.

Interactions among action chains are modeled by merging chains with a special element called an *internal event*. Internal events have preceding action chains and a subsequent action chain. Internal events can specify whether they wait for all incoming action chains to complete (AND) or the completion of a single one triggers the subsequence chain (OR).

Real-time analysis tools use this model to build a task set that can be analyzed with GRMA (Generalized Rate Monotonic Analysis) techniques [4].

A. Supporting Dynamic Systems

Dynamic system structure variations, such as the modification, creation, or deletion of components are commonly used. This is because at different stages, the system may be doing different things. For example, during take-off, an avionics system may have one set of active components, another during cruising, and yet another during landing. These variations need to be modeled to enable analysis at design time. A common model of system structure variations in the real-time arena is system modality [5]. System modality is

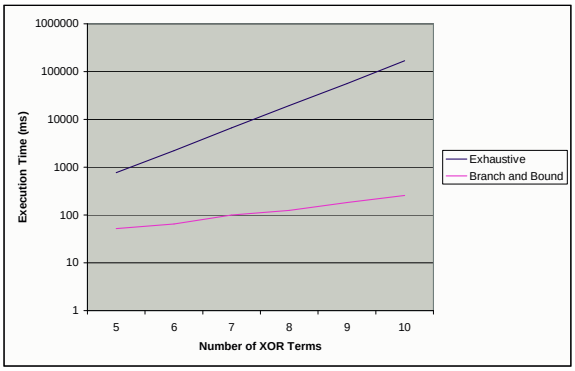

Fig. 5. Branch and Bound Execution Time Comparison

constructed as a set of modes and transitions between them. A system mode is a computation DAG. A transition between modes will create and/or delete nodes and edges in the DAG to convert it to another mode.

Our modality model partitions the transition system into subsystems that are allowed to perform mode changes independent of one another. In the general case where components in a subsystem can call components in other subsystems, the actual execution of the largest mode in one subsystem can depend on the operating mode in the other subsystem. One way to determine the worst-case load is to try all combinations of modes in all the components. In this case, we first organize the mutually-exclusive modes into XOR terms that group all the modes where only one of them can be active at any given time. A set of XOR terms is then defined to group all the independent XOR terms, i.e., all the terms would have at least one mode active. This is formally expressed as: $System = (m_1 \oplus m_2 \ldots \oplus m_i) \ldots \wedge (m_l \ldots \oplus m_m)$ where each mode m_i is a boolean variable that represents whether or not the mode is active. With this representation and the dependency graph, one can explore all the combinations that are possible, and determine which one would be the worst case. However, the search space of this representation is exponential in size. Specifically, given m modes in each term and t XOR terms, the size of the search space is m^t. As a result, an exhaustive search of the combinations needs to be avoided when possible. We solved this problem using a branch-and-bound technique.

A branch-and-bound algorithm [6] organizes a search space as a tree whose root represents the initial state with no exploration so far. A branch is added to the tree when a step is taken into some direction of the exploration, e.g., activating a mode in our modal system. The objective of the search is to find a solution with the best value, which in our modal system means the set of active modes with the largest load. A solution must include the selection of one mode for each term. As new combinations are tried, partial assignments and all the potential assignments derived from it would be discarded if a calculated bound has no possibility of getting better than the best solution found so far. This discarding is known as *pruning* and helps to avoid exploring all the possibilities.

d) Experiments.: We performed an experiment to explore the benefit of our algorithm. In this experiment, we compare the execution time of our Branch-and-Bound algorithm with the exhaustive search.

Figure 5 plots the results of our experiments. To smooth out the randomness of the execution time of the branch and bound algorithm we use an average over a hundred runs. This experiment illustrates the expected performance gains of our branch and bound scheme. We can observe that as expected, the exhaustive search follows an exponential increase in the execution time, while the branch and bound algorithm performs significantly better.

V. Conclusion

In this paper we presented the glue code generation framework of our model-based software development tool Time Weaver. We show how this framework supports multiple OS, middleware and target languages, and even the generation of multiple target languages from a single model. We discussed the experiments we performed to validate the extensibility of the code generation framework and the efficiency of the generated code. We then discussed how the timing model is built to have a faithful representation of the running code. Finally we presented a modality model that support variation to the structure of the software and a branch-and-bound algorithm to evaluate the worst case load that a modal system can impose along with experiments to evaluate its efficiency.

References

[1] Greg Bollela, Ben Brosgol, Peter Dibble, Steve Fur, James Gosling, David Hardin, Mark Turnbull, Rudy Belliardi, Doug Locke, Scott Robbins, Pratik Solanki, and Dionisio de Niz. *The Real-Time Specification for Java*. Addision-Wesley, 2001.

[2] Dionisio de Niz and Raj Rajkumar. Chocolate: A Reservation-Based Real-Time Java Environment on Windows/NT. In *Proceedings of the Real-Time Technology and Applications Symposium*, Washington D.C., May 2000.

[3] Dionisio de Niz and Raj Rajkumar. Time Weaver: A Software-Through-Models Framework for Embedded Real-Time Systems. In *Proceedigs of ACM Language Compilers and Tools for Embedded Systems Symposium 2003*, San Diego, CA, June 2004.

[4] Mark H. Klein, Thomas Ralya, Bill Pollak, and Ray Obenza. *A Practitioners' Handbook for Real-Time Analysis: Guide to Rate Monotonic Analysis for Real-Time Systems*. Kluwer Academic Publishers, 1993.

[5] Lui Sha, Ragunathan Rajkumar, and John Lehoczky. Mode Change Protocols for Priority-Driven Pre-emptive Scheduling. In *The Real-Time Systems Symposium*, 1989.

[6] Douglas R. Smith. Random Trees and the Analysis of Branch and Bound Procedures. *Journal of the ACM*, 31(1):163–188, 1984.

[7] TimeSys Corporation. TimeWiz: A Modeling and Simulation Tool, February 2004.