Usage Scenarios for an Automated Model Compiler

Ken Butts[1], Dave Bostic[1], Alongkrit Chutinan[2], Jeffrey Cook[1], Bill Milam[1], and Yanxin Wang[1]

[1]Ford Research Laboratory, Dearborn, MI, USA
{kbutts1, dbostic, jcook2, wmilam, ywang} @ford.com
[2]Emmeskay, Inc., Plymouth, MI, USA
alongkrit@emmeskay.com

Abstract. This paper is meant to motivate tools and methods research in the field of model-based embedded software development. In particular, we include usage scenarios to describe how an automated model assembler called a *model compiler* could support automotive embedded control systems development. We describe some desired characteristics and features of the envisioned model compiler and place particular emphasis on support for model compatibility checking. Finally, we describe characteristics of model components that are commonly used in practice.

1 Introduction

The automotive manufacturers' need to reduce cycle times and costs has resulted in a shift toward model-based design and development. Such processes offer the opportunity to generate and validate robust designs with minimal expenditures on physical prototypes. However, early adopters of model-based development often find that model creation is a time consuming and resource intensive task [1], especially when a large number of system and vehicle level models is required to support typical vehicle and technology variations.

We believe that, just as automotive vehicles are efficiently assembled from mass-produced parts and sub-assemblies, it should be possible to cultivate an environment in which subsystem models can be automatically validated for compatibility and assembled into system or vehicle models for model-based design and development. We strive for such a modeling environment to facilitate model reuse over product life cycles, across product families, and across development organizations. Experience has shown that, given sufficient attention to model compatibility, significant reuse can improve quality, reduce development cost, and reduce time-to-market. See [2] for a case history in applying software factory approaches to industrial control software development.

The automated model assembler, or *model compiler*, is the subject of this paper. A model compiler is a tool that automatically composes a model from a set of sub-

models and an architectural description of the arrangement of the sub-models. It will also ensure full connectivity of all control flow and data flow signals between sub-models, proper sequencing of sub-models, and compatibility of sub-models. In the following we use the term *component* to mean an element, whether part, subsystem, system, or vehicle, of a composition set that is submitted to the model compiler for assembly. To convey the scale and complexity of a typical assembly level model, let us consider a modeling environment used for energy analysis [1]. This model is composed of nine major systems; seven of these include controller, sensor, and actuator subsystems. There are forty-five locations in the environment where the user can select [3] between 129 component offerings. Within this framework, analysts can readily evaluate various vehicle technologies by assembling the components into alternative configurations.

Another interesting example is the powertrain control application. Here 218 algorithm model libraries, each with 3 to 30 component variations, are used to service 130 vehicle applications in the product-line. A typical vehicle application is composed from 75 to 105 components and requires more than 2000 signal-flow interconnections between the components. Ultimately, we would like to be able to link these two example environments by managing the vehicle specific powertrain control application as a single (albeit complex) component in the energy analysis environment; hence the need for automation. We believe the model compiler will be used recursively to assemble complex components and then to create systems of such component assemblies.

We describe our notion of a model compiler in section 2 and emphasize component compatibility analysis in section 3. Our application focus for this work is the on-board electronic control system domain, including the powertrain, chassis, and energy management systems. Given this domain, we describe how a model compiler can be used to support system component selection and requirements generation, facilitate system evaluation studies, and support large-scale software development factories in section 4. We conclude with a call for research collaboration in section 5.

2 Model Compilers

In order to understand the concept of a model compiler, we must first set the context within which one would use a model compiler and define what kinds of models are involved. While we feel the model compiler concept has a broad range of applications and should help to ease the use of many types of models, our scope for this paper is restricted to the Matlab®, Simulink®, Stateflow®[1] modeling domain.

Consider the powertrain control example mentioned in the introduction. Using median numbers, there are 218 different elements. Each of those has a median of 16 variants

[1]Matlab, Simulink, and Stateflow are Registered Trademarks of The MathWorks, Inc., Natick, MA.

leading to 3,488 possible components. From this domain we need to select 80, which would make up a median application. If we assume 25 input/output ports per component we now have 2000 connections to make to create a fully composed application model. We need to ensure that the signal characteristics actually make sense for each of these port connections. A simple signal name match does not necessarily imply that the connection is valid. We also need to ensure that the format of the signal matches at both functional and non-functional levels. At the functional level, we have to ascertain that the two signals are of compatible type: for example both should be scalar types or vector types of the same size. Non-functional characteristics that must be checked include verifying that the units of the two connections are correct. If, for example, an input-output pair attempts to connect "radians per second" to "degrees Kelvin", an obvious (non-functional) error can be detected and flagged to the user. If however, we are trying to match radians per second and revolutions per minute, the possibility exists to insert a conversion to correct the units mismatch. A warning would then be given to the user that this less onerous mismatch has occurred, but the units may interact correctly.

The model compiler requires the existence of component data dictionaries because the internal behavior of the model component may not be exposed. In the Matlab®, Simulink®, Stateflow® world, this is true if the component is an S-Function using C or Fortran. In the case where the component contains yet more Simulink® or Stateflow® components it is possible to build directed graph representations of the internal data and control flow of each component in order to look at optimal sequencing of components in a particular subsystem. The result of this analysis might be a Stateflow® component that deliberately sequences the components to ensure the proper sequence of operations.

Now that we have presented some of the tasks the model compiler has to accomplish, how do we envision it working? There are two necessary items that must be defined in order for the idea to work. First we need to define what constitutes a component. Second, we need to define how we select components and how we describe the hierarchical organization of the model.

2.1 Model Component Attributes

Some of the key component properties, which can be derived from a model compiler compliant component model, are:

- Sample Time (if the controller models are designed to be fixed step.)
- Solver (if one is used.)
- Set of inputs
  - Data dictionary definition for each input
    - Name.
    - Type.
- Set of outputs

  - Data dictionary definition for each output
    - Name.
    - Type.
    - Source component.
- Set of workspace parameters (typically, calibration parameters)
  - Data dictionary definition for each parameter
    - Name.
    - Type.

Once we have this information for each component to be composed, we can begin to determine whether the given components actually can be composed.

2.2 Assembly Model Structural Specification

One solution to the problem of model organization may be to combine the model component concept with a *formal language* [4] that describes the architecture of the model created by the components. Consider a simple example based on three simple components as shown in figure 1. Each component, X, Y, and Z, has some number of inputs and at least one output.

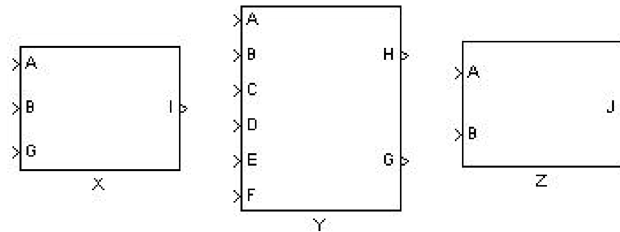

Fig. 1. Simple composition example

Each input and output is assumed to be of a scalar type. In addition, each input and output is named with a signal name. For our example these are just simple single character names, in a real model however they might be more descriptive names such as RPM, Coolant Temperature or Throttle Position. Now, define a component to have a set of inputs and a set of outputs and walk through the following scenario.

Accept, for the moment that the statement: Q { X,Y,Z} means that a component Q is to be created with members X, Y and Z. (Q might look a lot like figure 1.) The set of all inputs to Q can be found by taking the union of the sets of inputs from the three components:

$$X.in = \{A,B,G\}$$

$$Y.in = \{A,B,C,D,E,F\}$$

$$Z.in = \{A,B\}$$

$$Q.in = X.in \bigcup Y.in \bigcup Z.in = \{A,B,C,D,E,F,G\}$$

So *Q.in* represents the set of all inputs to Q. The same operation can be used to determine the set of outputs for Q. Figure 1 shows that the input G to component X is actually an output from Y. This can be determined by finding the intersection of *Q.in* and *Q.out*.

$$Q.out = X.out \bigcup Y.out \bigcup Z.out = \{G,H,I,J\}$$

$$Q.inner = Q.out \bigcap Q.in = \{G\}$$

So *Q.inner* represents the set of signals that originate and terminate inside Q. Another reasonable action might be to remove G as an element of *Q.in* and *Q.out*. If G originates within Q, it certainly should not also be a member of *Q.in*. It may, or may not, also be a member of *Q.out*. G could be removed from *Q.out* by default. Of course it may be possible that G is used by another component, which is a peer of Q. In that case, the model compiler would need the ability to search a model to find instances of G that are a match in both name and type to the G that is needed.

Our example demonstrates that, by applying simple operations on component attributes, we can 'wire together' the components to form either larger groups of components or a full model. As these functions are recursive, it is now possible to define a language to support this task of combining model components according to some rules. The use of rules implies that errors can be identified and flagged. This would be similar to the function of a High Level Language (e.g. ANSI C) compiler that does matching between prototype function declarations and instantiations of the function call. This leads to the idea that the model compiler will consist of two processes: one process will parse an architecture language that describes relationships between components. The other will parse the components to create mappings of relationships. The architecture language then becomes a control language to the output generator that will take the parsed components and create a new component as directed by the architectural description. The model compiler should also perform analysis on the components to determine if the components being combined are compatible.

3 Compatibility Relationships

The model compiler must ensure compatibility among the components being composed into the overall target model. There are two levels of compatibility that the model compiler must consider. Locally, as described in the previous section, the model compiler must ensure interoperability of the components in the same subsystem. Globally, the model compiler must ensure that only components that are designed to operate together are integrated in the same model. The following subsections describe the nature of these compatibility requirements in greater detail.

3.1 Structural Compatibility

The model compiler must first ensure the interoperability of the components at each Simulink® subsystem level. While the model compiler may not be able to fully determine whether the components in the subsystem are interoperable semantically, it can at least ensure structural compatibility of components in the following areas.

Signal Type. When connecting an output signal of a component to an input signal of another component, the model compiler must ensure that each input signal is of a type compatible to that of its source output signal. The type information for each signal may describe both *static* and *dynamic* [5][6] properties of the signals. Examples of static properties of a signal are data type (real, integer, boolean, trigger, enable, function call, etc.), units, and dimension. Dynamic properties describe how the signal evolves with time or how it is updated possibly both in simulation and real hardware. Examples of these properties are:

- Continuous-time vs. discrete-time (with sampling rate.)
- Data transfer protocols such as Controller Area Network, Time Triggered Protocol, or direct connection (signal values are always identical at both ends of the connection at any time.)

The signal compatibility may be defined in terms of the so-called *lossless convertibility* [5][6] where the output signal can be converted into the format of the input signal without loss of information. For example, an integer output can be converted into a real input. A discrete-time signal with can be up-sampled to a higher sampling rate provided that the new sampling rate is a multiple of the original sampling rate. The model compiler must ensure compatibilities of both static and dynamic properties of the signals being connected and notify the user when type incompatibility is detected.

Simulation Properties. Simulation properties may be specified for each component in the subsystem. Examples of such properties are ODE solver type, fixed or variable time step size, and tolerances. These properties essentially specify the preferred execution method for the component. The model compiler must ensure that the simulation properties for all components in the model are compatible. This assurance may be provided, again, through the concept of lossless convertibility similar to the one discussed in the previous discussion. For example, a model component that uses a fixed step ODE solver can be simulated with another component that uses a fixed step solver with a smaller time step, provided that the larger time step is a multiple of the smaller time step. The aggregate component now uses the smaller time step for the solver, i.e. the most general simulation property among all components is used. If no common simulation property can be found that all components can be converted into, the model compiler must notify the user of the conflict.

3.2 Component Compatibility

We discuss structural component compatibility earlier in this section. Here we attempt to address functional component compatibility. Our basic idea is to capture and exploit the knowledge that model components have been designed to satisfy certain product requirements or that they have previously been used successfully with other components. To refine this idea we introduce the concept of a *model domain*. Model domains are loose classifications that serve to sort model components and other domains according to some functional attribute. A *Customer Vehicle Character* domain could be used to classify vehicle level customer attributes such as *green* (environmentally friendly) and *fun-to-drive*. *Application* domains corresponding to actual automotive products (e.g. *2.0L, lean burn engine*) will be useful in tracking previously validated model assemblies. A *System* domain hierarchy could be used to replicate product system hierarchies (e.g. *Electronic Throttle Control*) to aid in model component organization and selection. It is often useful to aggregate model components that are considered alternatives in some sense. For example, models representing two types of electronic throttle control (ETC) sensors should reference the *ETC sensors* attribute of the *Component Choices* domain. Similarly, a *Parameter Choices* domain could aggregate model parameter choices for a single model component. We present usage scenarios associated with these notions of model domains in the next section.

To identify compatibility we assign *Member of* relationships between domains and components. If two components do not have a common domain attribute (reachable by tracing *Member of* relationships) then they are potentially incompatible and their combined usage must be validated. For example, the usage of two components that trace only to *green* and *fun-to-drive* respectively should be investigated since there is likely to be a conflict. Moreover, two elements that are members of the same choice domain attribute are, by construction, incompatible for a given instance. The *Member of* relationship can be specialized to the *Necessary member of* relationship when a fixed relationship exists (e.g. an *ETC system* always contains *ETC Controller*, *ETC Actuator*, and *ETC Sensors* components.) This specialization admits completeness compatibility analysis.

A component manager will be tasked with assigning domain membership relationships to each model component and its associated parameterization sets when it is checked into the model compiler database. A model component may be a member of multiple domains. The accuracy of the domain compatibility checking largely depends on the level of detail that is included in the domain models and the maintenance of the membership relationships. It may not be practical to model the domain structure to the fullest detail in general. Nevertheless, one can benefit from this type of compatibility checking. Even with partial domain information, the model compiler can still lessen the chance for the modeler to incorporate nonconforming components into the same model.

4 Usage Scenarios

In this section we describe how the concepts presented in the previous sections can be applied to automotive control system development. The powertrain challenge models in the Model-Based Integration of Embedded Software project [7] are representative of the types of model components that could be assembled into automotive system models. We have plant components that model the behavior of the physical system under control. In this case, the engine, transmission, vehicle, sensors, and actuators are managed as components whose hybrid-dynamics [8] evolve in the continuous and crankshaft-position domains. The signal-flows between these components represent physical quantities such as pressures, torques, and velocities. Controller-function models form a second component type. The dynamics of the controller-function components are event-driven. Thus the controller-function component has two interface types: the signal-flow for representing inter-function signal-flows and the function-call event for passing initiation directives from function to function. The third component type is the scheduler type that serves as the control-flow interface between the physical and controller function types. The scheduler observes the physical system and starts threads of initiation directives in the controller functions controller.

Figure 2 shows how the three basic components interact in a control system model. Note that the controller-input models are a special controller-function (event-driven) type that input continuous physical signal-flows and output event-driven controller-function signal-flows. Conversely, the controller-output models are a special plant type that input controller-function signal-flows and output physical signal-flows.

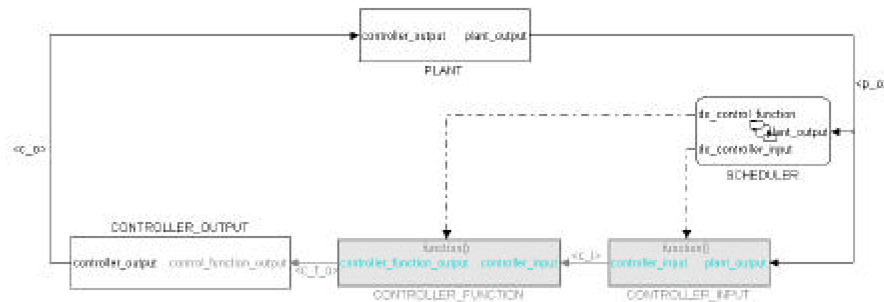

Fig. 2. Component model types

Let us consider the task of developing an embedded control system for a new vehicle application. Given the appropriate level of abstraction, the engineering processes and corresponding model applications are similar for the powertrain, chassis, and energy management electronic control domains. The remainder of this section describes the role of the model compiler in an advanced electronic controller development process.

4.1 System Definition

Once the requirements for a vehicle are known, a system must be assembled to satisfy those requirements. As discussed in the introduction, component reuse can have a significant impact on quality, development cost, and time-to-market. However in practice, we experience three difficulties with component-based development at the system definition stage. First, how does the system designer sift possibly thousands of component models to identify the appropriate set of components for this system? Second, how does the system designer know that the identified set of components is consistent in terms of interface and system architecture? Third, how does the system designer know to request new component development? And what are these new component's requirements in terms of interface and system architecture?

Sifting the Components (Creating an Assembly Model Build List.) Let us show by example that the previously described notions of component compatibility are applicable here. A vehicle team is charged with developing a *green* and *fun-to-drive* vehicle and the system designer decides to specify a *2.0L, lean-burn engine* to accomplish this task. The control designer typically assembles a system model from previously used components to validate this design choice.

Consider the (incomplete) component relationship model shown in figure 3. This model contains valuable information that can guide the creation of an appropriate assembly model build list. By selecting the *2.0L, lean-burn engine* from the *Application* domain the control designer can readily see that this engine is a plausible choice for a *green* vehicle because it is a member of the *low emissions* and *low fuel usage* attributes. Moreover, the designer learns that this engine must employ an *engine management system (EMS)*. The designer could choose to include the optional *electronic throttle control (ETC)* system since it further contributes to *low emissions* and has been previously applied in the *2.0L, lean-burn engine* application. Thus the control designer can sift through the relevant application domains and their component membership relationships to create a complete build list for the assembly model of interest.

We remark here on an implicit requirement of the model compiler technology. Please notice that the component relationship model includes the ability to create component and parameter choice relationships. As described in [3], there are several design scenarios in which the control designer may wish to iteratively switch between predefined choice selections without incurring the cost or system architecture modifications associated with a system model re-compile. Thus we require that the model compiler and Large Scale Model [3] model management technologies be complementary.

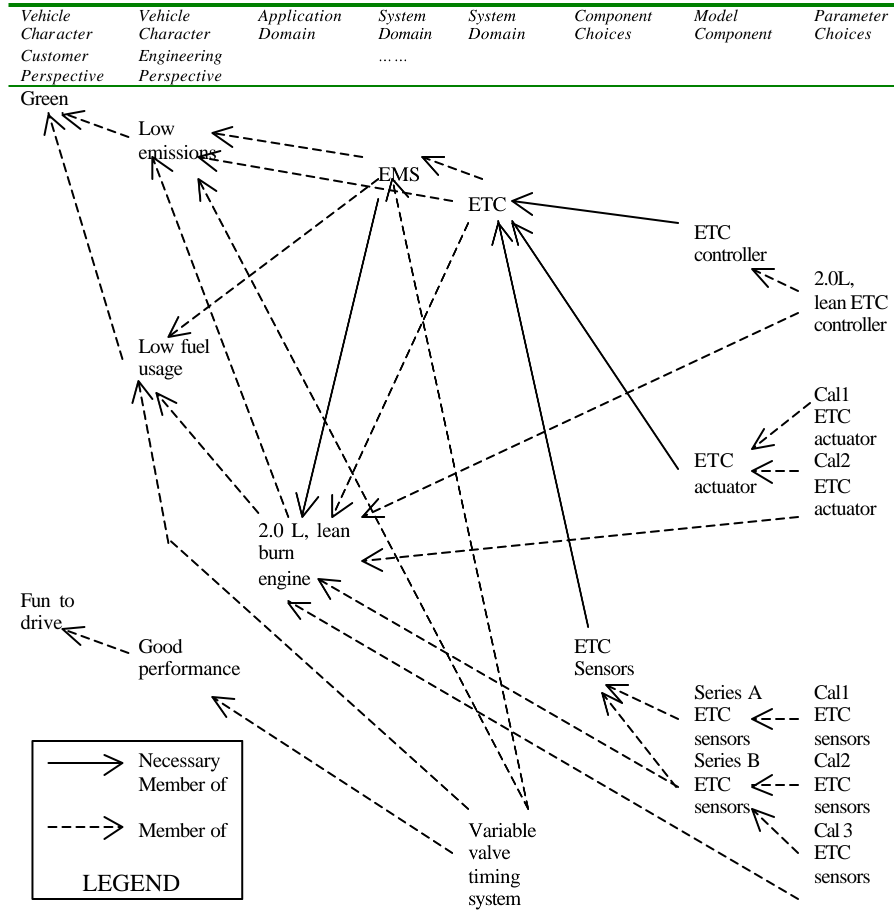

Fig. 3. Component Relationship Model

Analyzing the build list. Once the control designer has generated the assembly model build list, the analysis engine of the model compiler can be used to check the quality of the build list specification. We envision that the following validation analyses will be particularly useful:

- That the assembled model will satisfy the subsystem compatibilities described in section 3.1.
- That all inputs are either generated by a component model or defined as assembly model inputs.
- That all unused output signals are either terminated or defined as assembly model outputs.

- That all component compatibility relationships are satisfied or explicitly overridden.
- That the build list is complete in the sense that all "necessary member of" relationships are satisfied.
- That the build list is unambiguous in the sense that component models with choices employ only one component selection per location.
- That the specified hierarchical structure follows (user-defined) architecture rules, for example:
  - Plant component models are grouped with other plant components.
  - Controller-function component models are grouped with other controller-function components.
  - One and only one scheduler component model is associated with each controller-function group.

Generating New Component Requirements. Let us continue the example from section 4.1. Assume that inadequate *fun-to-drive* characteristics are generated from the assembled vehicle system model that includes the *2.0L, lean-burn engine* with *electronic throttle control* system. Under investigation, the component relationship model (figure 3) will show that the *variable valve timing (VVT) system* is a member of *good performance*, *low fuel usage*, and *low emissions* attributes. Thus it is a good candidate for inclusion in the vehicle system build list.

However, in this case the designer learns that a *VVT system* has not been previously used in the *2.0L, lean-burn engine* application. Consequently, new component development must be initiated. The model compiler can generate a set of interface and composability requirements that arise when the *VVT system* components are included in the system build list. These requirements can then be used to guide a work task to:

- Determine if a new air management coordination controller (between the *ETC* and *VVT controllers*) is warranted.
- Resolve any interface and composability issues that arise when the new controller and *VVT system* components are added to the build list.
- Generate a complete set of parameter component choices for the new controller component and system application.
- Update the component relationship model upon design validation of the new components and system application.

We believe that the creation and maintenance of the component relationship model will help in the system definition and new component creation phases of development. Design iterations between system and control designers will be significantly enhanced via efficient reuse of compatible components.

4.2 System Validation

In today's practice, it is a year's major accomplishment to assemble a powertrain control application model as described in the introduction. Thus full systems

integration models typically do not support the design stage of development. Usually, our downstream engineering customers verify systems integration and performance in the target vehicle application. Of course, this tends to damage inter-departmental relationships, reduce engineering efficiency, and increase time-to-market.

The ability to create large system models in a timely and efficient manner is a critical benefit of the model compiler technology. We believe that systems validation during system and component design (prior to component implementation and subsequent systems integration) will be possible. The assembly models created with the aid of the model compiler provide a complete description of the system dynamics. The functionality of the full system can thus be evaluated using simulation. Here again, Large Scale Model technologies [3] that allow the control designer to switch between model component choices within a given model structure would be useful. Pre-compiled, interpreted, empirical, or first-principles versions of a model component could be made available to manage the simulation speed versus model complexity trade-off.

Full system models are particularly useful for studying the system's:

- Robustness to parameter variation.
- Performance under various environmental conditions and usage scenarios.
- Performance in response to failures and faults.

We believe these studies should be made in the context of validating the system requirements. Requirements test procedures should be embedded in Test Scenario model components that have traceability to the product requirements database.

We intend to provide full system assembly models (complete with Test Scenario components) to our internal and external suppliers so that they can develop a better understanding of our requirements for end-product application. We will also ask the suppliers to contribute component models of their subsystems and devices. Requirements for these requests should be consistent with the compatibility relationships discussed in section 3. Intellectual property protection mechanisms need to be developed to facilitate model sharing across organizations.

4.3 Software Development

The development process and tools that comprise the model compiler can have a significant impact on the development of embedded software implementations of the compiled models. In today's powertrain controller development process, we use the validated control-function component model as a template for software development. Unfortunately, this process is labor intensive and time-consuming due to its inherent reliance on the software developer's ability to analyze the control-function model, write embedded software that is intended to behave as specified in the model, and verify that the embedded software and control-function model have the same behavior.

Model compiler technology, coupled with code generation technology can improve this situation dramatically. Newly available tools now allow us to automatically generate embeddable software directly from the model specification. Consequently, the model compiler should capture data flow interaction among model components in a format that is compatible with today's automatic code generators.

The compatibility relationships previously described in section 3 are important when using the models for software development and automatic code generation. Our internal control-function modeling guidelines address discrete-state behavior, software-tasking architecture, and distinct data flow and control flow partitioning. Thus they provide the determinism exhibited in our implementation language: ANSI C. These characteristics, when embedded in the component object, ensure consistency between the generated code and the control-function model without need for additional configuration.

The consistency of the dataflow interfaces between components will provide the single biggest benefit to the automatic code generation phase. This is because managing the different interfaces of each component or connection is generally time consuming. Certainly, the interfaces on archived components may not align with traditional C boundaries, which are function interfaces. Fortunately, this is not a necessity. Control-function components verified for inclusion in the component library may have some superstructure that naturally maps subcomponents into functions, but the interface at the component level should save the software developer from having to trace every data source and destination for verification.

The specification of each model component as an independent entity will have significant positive impact on the code generation process as well. This process calls for functional validation prior to software development necessitates rigorous up-front requirements capture: a critical quality attribute in the software development process.

The ability to connect multiple components has been acknowledged as beneficial for simulation, but can have many advantages for embedded software development efforts as well. In our current software development process, each element is typically specified and designed in isolation. The interfaces to these elements are defined, and static test inputs are used to evaluate the correctness of the implementation. Using the model compiler technology to replace this approach can have multiple benefits. For example, if multiple components are joined (especially where there is strong data flow connectivity among the components), a single test vector can generate inputs for blocks further down stream. Furthermore, an environment with linked components can be used to test the runtime realities of component interaction more comprehensively. In concert, these two characteristics could dramatically reduce the time it takes to verify software.

In a similar manner, the model compiler's ability to operate on joined components can be used to classify and optimize various software performance metrics. ROM, RAM, and CPU usage are of particular concern when the software is destined for an embedded controller. In the embedded environment, computing resources are

constrained, so it becomes important to be able to track the allocation of those resources. Once characterized in a particular execution environment (specific CPU, compiler, test vector, etc.) these data can be assigned to the component. Granted, this method will be more useful for data flow oriented components, which have more deterministic schedules, than for control flow oriented components that are less timing consistent. With information detailing memory usage and performance now available at the component level, intelligent, performance-conscious analysis and optimizations can be made during the model compilation process.

5 Conclusions

We have captured the need for an automated model compiler, described some of its desired features, and discussed its application in automotive control systems development. Our aims are to clarify and express our needs (both for our partners and ourselves) and motivate a dialogue with systems researchers and tool providers. We wish to express our apologies to the research community for not conducting an extensive literature search to support the paper. Perhaps solutions similar to those proposed in this paper already exist. In any event, we look forward to a dialogue on model compiler technology development and application. For example, the maintainability and scalability of the component compatibility notions presented in section 3.2 are un-quantified and must be verified for practicality.

Acknowledgements

We would like to acknowledge support by the Defense Advanced Research Projects Agency under the DARPA MoBIES contract number: F33615-01-C-1841.

References

1. Jennings, M., Tiller, M., Butts, K., "Defining a Flexible Modeling Methodology for Design and Development of Automotive Powertrain Systems," to be published in the Proceedings of the ASME 21st Computers and Information in Engineering Conference, Pittsburgh, PA, September, 2001.
2. Cusumano, M.A.,"Chapter 5 - Toshiba: Linking Productivity and Reusability," Japan's Software Factories, A Challenge to U.S. Management, Oxford University Press, Inc., New York, 1991.
3. Sivashankar, N., Butts, K., "A Modeling Environment for Production Powertrain Controller Development," Proceedings of the 1999 IEEE International Symposium on Computer-Aided Control System Design, Hawaii, August 1999.
4. Sudkamp, T. A., Languages and Machines, An Introduction to the Theory of Computer Science, Second Edition, Addison Wesley, 1998.

5. Lee, E. A., Xiong, Y., System-Level Types for Component-Based Design. Technical Memorandum UCB/ERL M00/8, University of California at Berkeley, February 2000.
6. Lee, E. A., "What's Ahead for Embedded Software?," IEEE Computer, September 2000, pages 18-26.
7. Model-based Integration of Embedded Software, Information Technology Office, Defense Advanced Research Projects Agency, http://www.darpa.mil/ito/research/mobies/.
8. Butts, K., "Analysis needs for Automotive Powertrain Control," Proceedings of the 7th Mechatronics Forum International Conference, Atlanta, Georgia, September 2000.