In Proceedings of the Automotive Software Workshop on Future Generation Software Architectures in the Automotive Domain, San Diego, CA, Jan. 10-12, 2004.

Two-tiered Architectural Design for Automotive Control Systems: An Experience Report

Kevin Steppe, Greg Bylenok, David Garlan, Bradley Schmerl,
Kanat Abirov, Nataliya Shevchenko
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15221
{ksteppe,gbylenok,garlan,schmerl,kanatknt,san}@cs.cmu.edu

Abstract

An attractive approach to architecture-based design is to structure the development process into two tiers. The top tier represents the abstract design (or architecture) of a system in terms of abstract components. The bottom tier refines that design by making specific implementation decisions, such as platform, middleware, and component implementations. While attractive in principle, there has been relatively little industrial-based experience to shed light on problems and solutions involved in such an approach. In this paper we describe our experience in developing tools to introduce a two-tiered model-based approach to the design of Ford Motor Company's automotive control systems, highlighting the principle challenges, and evaluating the effectiveness of our solutions for them.

Keywords: Model-driven architecture, architecture-based design, architecture design tools, software development environments.

1. Introduction

Over the past decade industry has become increasingly aware of the value of architectural models and architecture-based design. Architectural models define a system at a high level of abstraction – typically in terms of a system's interacting components – where major design tradeoffs can be analyzed before committing to a particular implementation. Architecture-based design typically starts with an abstract architecture and then refines that model to the point where the system can be directly implemented.

For many classes of system it is helpful to structure the process of architecture-based design into two phases. In the first phase system architects design a system in terms of abstract capability, postponing implementation decisions, such as execution platforms, communications infrastructure, and component implementations. In the second phase implementation commitments are made in a structured and systematic way. For example, abstract components can be assigned to platforms, middleware can be chosen to support distributed interaction between components, and component libraries can be used to select implementations for the abstract components.

While attractive in principle, a two-tiered approach to model-based development raises a number of interesting issues that have yet to be fully understood. What kinds of notations are best used to represent the two levels? What kinds of architectural features should be modeled at each level? What parts of the refinement process can be automated? How well does the approach scale to realistic systems? To what extent does the application domain influence the process and artifacts?

In this paper we offer insight into these issues by describing our experience of developing a tool to support two-tiered design methods for automotive control systems at Ford Motor Company. Currently, Ford has detailed component specifications in the form of Simulink [9] models. However, they are currently only able to perform component level analyses. Composition of the Simulink models is currently performed manually, and even constructing small subsystems can take a matter of weeks; composing an entire automotive software system is infeasible with this approach. Our goal was to provide automated assistance that introduces a two-tiered modeling process and dovetails well with Ford's current practices. In particular, Ford engineers desire to initially design their systems at a high level of abstraction in terms of abstract entities such as controllers, sensors, and schedulers. These components could then be associated with specific component implementations and their Simulink models, and used to automatically produce a more detailed composition on which detailed design analysis could take place.

As we will illustrate, architecture description languages and their associated tools can play a central role in supporting two-tiered architectural design. However, there were a number of difficult technical hurdles that we had to surmount – hurdles that we suspect will be characteristic of many other domains in which this kind of support is required. Section 2 discusses some related work. In Section 3 we discuss the approach we used, give an overview of our tool (called Synergy), and present an example

problem to provide context. Section 4 & 5 discuss the notational and technical issues encountered and the solutions we used. We conclude with a review of the implementation status and evaluation of results.

2. Related Work

The use of two-tiered approaches to software development extends back to early specification languages, such as Larch [6], which advocated the separation of abstract capability from implementation details. More recently the Object Manage Group has been promoting model-based design using a two-tiered approach that they refer to as "Model-Driven Architecture." (MDA) [10]. MDA is motivated by similar concerns to ours, but attempts to advance the state of understanding about how to carry out such an approach in the context of real systems, complementing existing development methods, and leveraging special features of a product domain (in our case automotive control systems).

Within the area of architectural design, many people have advocated the importance of multi-view approaches [3][8]. A two-tiered architectural method can be viewed as a specialization of such approaches, focusing on two specific architectural views: an abstract and a concrete view. The specialization allows one to consider general questions of multi-view consistency, and requirements for multi-view tools in a more limited, but tractable, context.

There has been considerable recent interest in model-based approaches to embedded control systems, such as automotive and avionic. For example, the DARPA-sponsored MoBIES Project specifically focuses on this area, and has developed a number of techniques, notations, and tools [2]. Our work fits within that general category of research, but explores the specific consequences of using architecture description languages as the carriers of embedded systems designs.

The ISIS group at Vanderbilt has been working with Ford on a similar project [7]. They have so far focused primarily on handling constraint satisfaction within a large design space through BDD trees. Their solution uses a single abstract view and presents the acceptable solutions to the given constraints. The single view model, however, prevents the user from fine-tuning or validating the selections.

3. Two-tiered Architectural Design

In this section we describe a motivating example that is typical of the work of a Ford engineer designing an automotive software system. We cover the architectural styles for background. We describe in general terms a scenario of what engineers need to do, and then continue with how to think about this example in architectural terms. We then briefly introduce an architectural tool, AcmeStudio, which is a typical software architecture environment. Following this, we show how Synergy, an augmentation of AcmeStudio, supports the scenario.

3.1. The Problem

Ford Motor Company, like many in the automotive industry, build software systems for all their car models. The software used in these systems is life critical, where the failure can cause loss of human life. To address this, Ford develops Simulink models of their componentry so that they can conduct rigorous analyses of these components to help ensure reliability. These models are a detailed specification indicating all component interfaces in addition to properties supporting simulation. Component models are reused and iteratively changed across projects.

Despite having the ability to analyze individual components, producing assemblies of these components to be analyzed is problematic and does not scale. Currently, these compositions, if they are built at all, are constructed manually. Because components typically have dozens of interfaces each, manually connecting them is tedious and error prone. One of Ford's main needs is to determine whether all input ports in a model are connected. If any input port has not been connected the final system will not work. However, for a typical six component subsystem, Ford engineers report that construction takes approximately two weeks. The handful of large (50 component) vehicle control subsystems developed have taken six months to produce.

Factor in that there are multiple choices for each component (e.g., it is possible to use one of several wind shield wiper servos), and the combinatorial explosion of possible combinations quickly makes manual construction absolutely infeasible.

However, this problem presents an ideal opportunity for automated tool support. Typically, engineers think of constructing their software in terms of abstract *system architectures*. Detailed compositions, called *assemblies*, are only necessary when performing detailed analysis. Thus, we introduced a two-tiered modeling approach that reflected this, and allowed assemblies and composed Simulink models to be automatically generated.

3.2. Overview of Approach

Thinking about a software system in terms of its components and interactions can be represented with software architectures [1][11][14]. A software architecture represents a system in such a way, and is amenable to automatic analysis. To model an architecture in a specific domain, it is common to use an architectural style [4]. An architectural style is a vocabulary of the possible types of components, connectors, and interfaces that can be used

in a particular domain, in addition to rules governing the correct composition of these elements.

To address the problem described above, we introduced two levels of architecture representation to be used by Ford: the high-level *System Architecture* and the low-level *Assemblies*. These levels of abstraction are represented by two related architectural styles, which will be discussed in Section 3.3. In this section, we discuss a typical scenario for which Ford engineers desire tool support.

An engineer starts designing the software for a car by creating a high level architecture of the system. At this level, the engineer is only concerned with the high-level vocabulary of Servos, Managers, etc, rather than particular implementations or specializations of these. For example, at this level the engineer may only be concerned with putting together a system architecture for the cruise control aspect of the car, and not be concerned about the low level details such as particular connections, timing requirements, or memory footprint of particular implementations of a cruise control manager for a particular car model. At this level, the engineer is concerned with whether the system is well-formed and consistent, and also which subsystems of the car interact with other subsystems. Because the system architecture is abstract, it can be reused in other products, both within Ford or its subsidiary companies.

After creating the system architecture, the engineer associates the abstract components with component models that are stored in a repository and which may be related to particular products within the automotive lines. They are reused for new abstract architectures.

In order to manage the complexity of the architecture, the engineer may divide the architecture into sub-systems and compose them in a hierarchical fashion. In such cases the user must associate the deepest components of an abstract component with component characterizations.

After associating abstract components with characterizations of those components, the engineer needs to produce an assembly from this abstract design. This assembly is based upon platform-specific information provided by the component characterizations, and involves choosing among multiple alternatives for components.

The final step of the process is translating assemblies into Matlab/Simulink models of the entire subsystem. The assemblies can then be imported into Matlab, and pre-existing analyses and simulations of the system can be conducted.

3.3. Architectural styles

Although each tier of our approach is closely related, we developed two architectural styles to allow modeling at each level in Acme [5]. When working at the system architecture level, Ford engineers use the *Ford-System Architecture* style to create abstract architectural models. These models are then translated by the tool into architectural models in the *Ford-Assembly*R style.

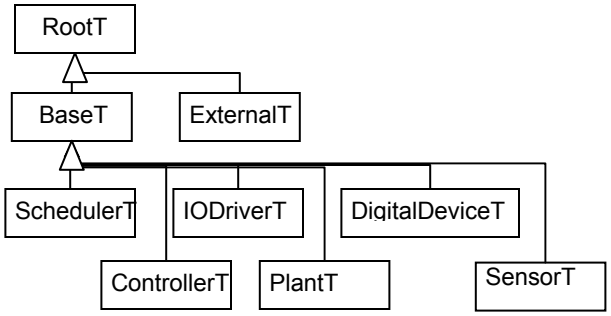

Figure 1. The Component Types of the System Architectural Style.

To design these styles, we analyzed documents, Matlab models, and papers provided by Ford researchers. We then created Acme families based on these artifacts by defining element types and creating rules. The high-level vocabularies of these styles consist of component types, their expected forms of interaction in the form of connector and interface types, and a set of constraints (heuristics and invariants) on how components should be assembled into abstract or low-level systems.

The Ford System Architectural style consists of the following elements:

- Ten component types organized as depicted in Figure 0. The component type RootT defines a property allowing a component characterization to be assigned to a component. At this level, the components in the architectural style merely interact with other components via input and output ports. These ports are defined in BaseT. ExternalT represents the point of interaction between the subsystem being defined and other systems. For example, if the cruise control subsystem must interact with the brake subsystem, it does so through this component.
- One connector type, called CSignalT that all components at this level must interact through.
- Associated port and role types for input and output, representing interaction points between components and connectors.

Figure 0 illustrates the abstract architecture of our example cruise control system. At this level, the architectural style is not particularly rich. Ford engineers are only concerned about the interconnections between components at this level. Rules specify that all ports must be attached to roles, and that BaseT's contain a single abstract input and output port.

At the Assembly level, engineers are concerned with detailed knowledge of the connections between compo-

nents. Thus, while the component types at this level are the same as at the abstract architectural level they may have any number of input and output ports showing detailed communication. Additionally there are two new connector types:

- ManSignalT, indicating a signal connection added by the user at the assembly level rather than automatically generated from the system architecture.
- Bus, representing a specific publish-subscribe or shared data bus connection.

Because architectures in this style are generated from the higher design, this style does not have any new rules on its connections. Similar rules to the high level are maintained in case engineers manipulate the model at this level.

Generation of the Assembly level from the System Architecture level involves elaborating connections between components in detail. All the possible ports on a component are enumerated. Then all legal connections for those ports to other components are made. This detail is provided through the component characterizations and the connections made at the system architecture level.

3.4. Overview of Synergy

A typical architectural development environment provides support for producing architectural models, and conducting architectural analysis to determine properties of the model, such as performance, quality, and correctness.

Many architectural design tools are written to work with a particular architectural style, and making them work with other or customized architectural styles is difficult. More recently, architectural tools have been developed that allow users to customize the environment based on particular architectural styles. Among these include Unicon [15], Mae [12], and AcmeStudio [13].

We used AcmeStudio as a basis for this project. AcmeStudio is a style-neutral architecture development environment that can use any style written in Acme and tailor the environment for that style. It provides access to the element types in the style, style-specific depictions of elements in the architectural diagram, and support for analysis of rules. Quite recently, AcmeStudio has been retargeted to the Eclipse platform, which enables extension through the notion of plugins. AcmeStudio takes advantage of this feature to allow style-specific analysis to be integrated and used by designers. These extensions allow customization of the user interface, specialized views of the architecture, and access to extra analyses. This meant that it was possible to reuse a large body of code, and concentrate only on the areas specific to Ford.

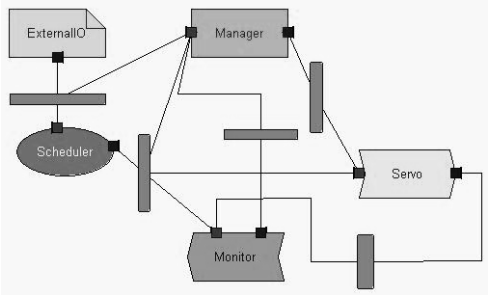

Figure 2. The System Architecture of the Design.

In this experience we extended AcmeStudio so that it supports the two architectural styles that were developed for Ford, the ability to read Ford's Simulink component models, and generation of assemblies from high level system architectures. This tool is called *Synergy*. In this section we give an example of using Synergy to support the scenario described above. In later sections, we discuss the issues that we needed to resolve in order to make this tool useful to Ford engineers.

3.4.1 Creating the abstract design

To illustrate the use of Synergy, and to give an impression of its capabilities, we follow the design of a simple and generic cruise control system. The demo system discussed further in the paper is rather simple.

The system represents a cruise control system that consists of the following components:

- "ExternalIO". This ExternalT component provides a bridge between this system and other outside systems. This component sends parameters to "Scheduler" and "Manager" components such as scheduling periods and cruise settings respectively.
- "Scheduler". This SchedulerT component manages usage of hardware resources of the system and schedules all the components.
- "Manager". This ControllerT component is responsible for controlling the Servo component.
- "Servo". The ActuatorT component manipulates physical devices. The physical devices are not included in this example, but can be modeled with PlantT components.
- "Monitor" is a SensorT type which reads data about the hardware from the servo and sends results to the manager.

The user creates an abstract architecture of the demo system without any platform specific details (see Figure 0).

At this abstract level a component of the architecture is represented with two ports (input and output). These input and output ports are interfaces to other components of the design. Connectors are represented as buses that connect interfaces of the architecture's components.

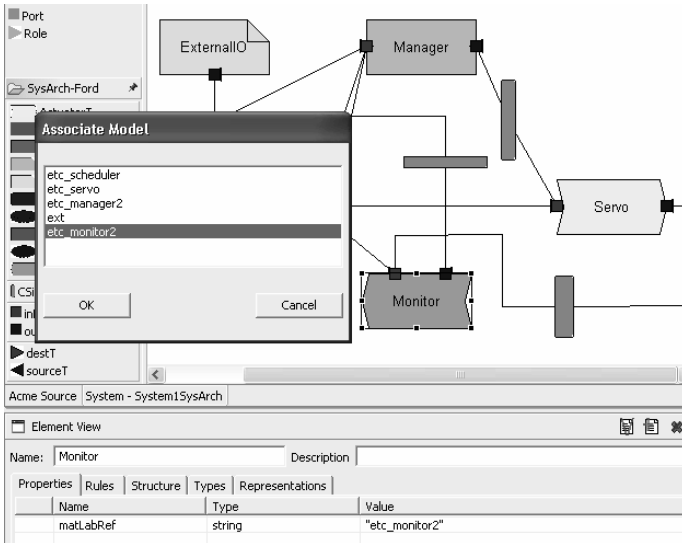

Figure 3. The user links Acme components to Matlab/Simulink models.

After creating the abstract architecture Ford engineers must either associate each component to existing Simulink models or refine the component as a subsystem. Synergy allows the user to browse and select a Simulink model to associate with a component in the project, as depicted in Figure 0. In this version of Synergy, there is currently no check that the right *kind* of Simulink model is assigned to a component. For example, a scheduler model could be assigned to an Actuator component. An error of this sort only becomes apparent in later stages of the design.

Design choices between alternative Simulink models or subsystems can be specified through AcmeStudio's support for multiple representations. For each possible choice, a new representation of a component is created, with subcomponents. A component characterization representing the particular alternative is then assigned to subcomponents.

3.4.2 Creating the assemblies

After the user creates the abstract architecture and associates its components with component characterizations, Synergy allows the user to generate low-level architecture alternatives or "Assemblies". The low-level architecture elaborates details of the abstract architecture and they are described as follows:

- The set of input and output ports are completely enumerated. These sets are detailed refinements of the abstract input and output ports shown in System Architecture.
- Connectors of the design at the assembly level are detailed representations of the buses used at the abstract level. These buses are refined as end-to-end connectors, each having only two roles connecting two neighboring components.

At this time legal connections are defined by a match of input port name and output port name. The buses created in the system architecture are first represented as hashtables. The name of output ports connected to the high level bus is added to the table. Then, for every input port connected to the bus, the corresponding output port is found in the table and a point-to-point connector is added to the assembly. Subsystems are handled recursively, built from the bottom up. Though no data have been collected, this solution should scale linearly with the total number of ports in the system – an important feature as a large system could contain thousands of ports.

For example, onsider the bus in between the 'Scheduler' and 'Monitor' in Figure 2. That bus becomes a hashtable of all the output ports from the scheduler component. The Manager, Monitor, and Servo components all have input ports attached to the bus and thus may search that hashtable for connections. For the 'Monitor', the port named 'trig_etc_monitor_fast' will match with the output port of the same name on the 'Scheduler' component as shown in Figure 4.

Such detailed representations of components and connectors empower the user with the ability to conduct complex analyses using heuristics and constraints of Acme architecture description language and/or custom-developed analyses that could be plugged into Synergy.

We added two analyses that demonstrate this plug-in feature. The first takes a numerical property (CPU usage in our case) and sums across all components, checking if the final total is less than the system wide constraint specified by the user. The second gives a suggested scheduling order, by creating a dependency graph using Apache's Commons project graph code. Cycles are identified and reported, then a possible ordering suggested.

3.4.3 Generating Simulink Models

Ford's primary model analysis tool is Matlab/Simulink. Thus it was vital that Synergy produce Simulink models from the designs. As noted above, all the atomic components are linked to Simulink models. Additionally, all the ports in the assembly view map directly to ports in the Simulink model. Using those mappings Synergy generates a model using the Simulink scripting language. An example of the Simulink model generated by Synergy is presented in Figure 4. Once run through Simulink the model can be re-imported into Synergy as a single component, allowing for iterative development of ever larger models.

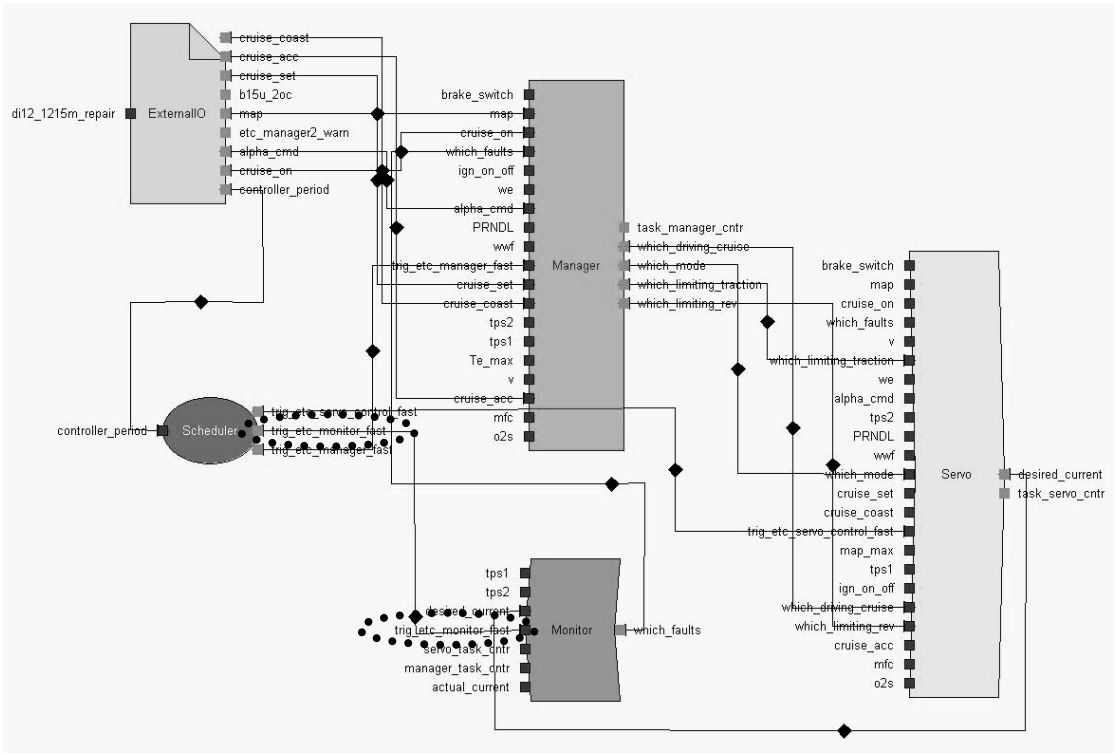

Figure 4.The Assembly for the Design.

Summarizing this example – we have given Ford architectural styles tailored to their domain modeling problems. AcmeStudio provides an effective tool for developing abstract architectures of vehicle control systems. The Synergy extension to AcmeStudio automatically generates a more detailed view of the 'assembly'. The user is then able to review, analyze, and modify the details of the design. Finally the entire assembly can be converted to Simulink for further analyses.

4. Representation of component architectures

Although in the example above, the use of Synergy seems straightforward (because it was designed that way), we had to deal with a number of representational issues. Other architectural approaches to multilevel design are likely to encounter similar issues.

4.1. Styles of multiple representations

The most obvious issue in multilevel design is how to represent each level. We needed two design levels, the more abstract *System Architecture*, and the detailed *Assembly*. As our intent was to simplify component compositions, the abstract level needed to be easily created by the user. We also wanted the ability to do some analysis at the abstract level before generating detailed assemblies. The detailed level had to support the many ports and connections present in the system as well as the properties to be analyzed. In our case we used the Acme ADL for both levels. To better represent the features of the two levels, we used different architectural families for the two levels. The family provides and requires the attributes and design restrictions appropriate to that level. While it would be possible to use different languages for each level, using the same one as in our case means that engineers don't have to learn to two different tools. And since both views and manipulations are architectural, it made sense to use the same language.

4.2. Hierarchy

Ford's vehicle control models are likely to include one hundred different components. Managing that scale requires breaking the system into modules and subsystems via architectural hierarchy. In fact, Ford engineers generally work with only five or six components at a time, constructing large systems from these subsystems. Additionally, hierarchy provides a mechanism for re-using assemblies. Thus large architectures can be built in a bottom-up

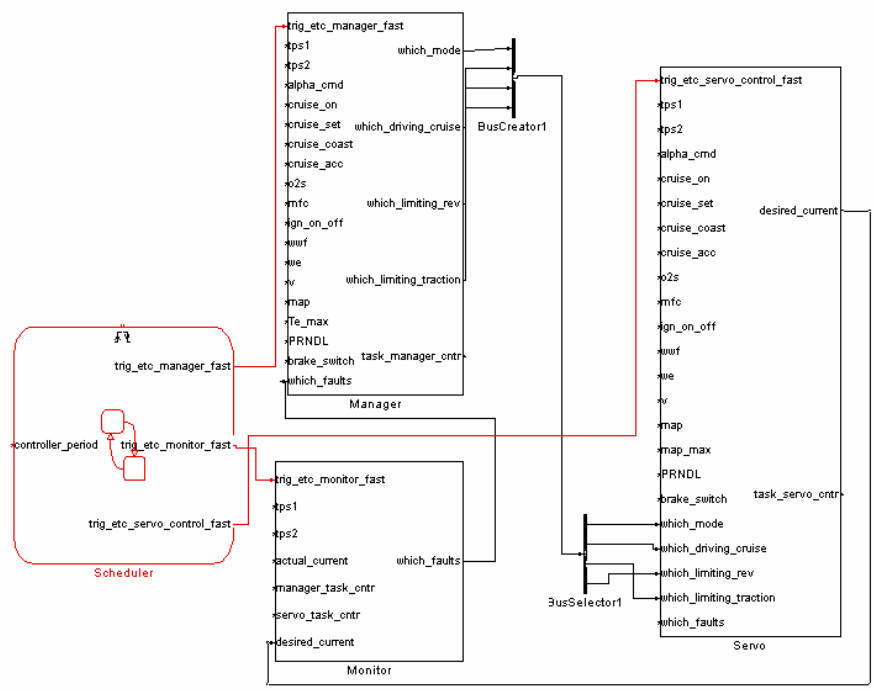

Figure 5. The Simulink Model generated by Synergy

approach from previously built assemblies. In our case hierarchical design was supported through Acme representations. In representations, the abstract component is a place-holder for the underlying sub-structure, a scheme which matched our intended use of hierarchy.

The interface to the subsystem must be clear for the engineer to make use of the subsystem. At the system architecture level a single input and output port is sufficient, however the details must be identified somewhere. This interface must also be mapped to ports within the sub-structure. In some cases the mapping from external port or ports to internal structure will have higher than 1:1 cardinality – for example an internal source 'A' may supply data to two components outside the subsystem. In these cases a mechanism for one to many mapping is needed which can distinguish the different possible meanings applicable to the domain. One to many mappings could be fan-in, fan-out, a priority arrangement, a rotation of usage, or some other meaning depending on the domain. Some method is required to indicate this choice in the model (so that it can be used in analysis, for example). Unfortunately, no mechanism currently exists in Acme to make this possible, and so handling different kinds of one to many mappings was not supported.

4.3. External interfaces

Any design will need to interface with other systems, requiring that the system representation includes facilities for identifying that interface. The interface is also crucial to using hierarchy and building progressively larger architectures. The question becomes how that interface is determined. Options include explicit specification by the designer, automatic generation from the available ports of the system, or leaving unbound ports. Explicit specification enables the system to check for completeness, but requires a lot of extra work for complicated interfaces. Automatic generation is easier on the user, but could mask incompleteness by generating additional inputs. Leaving unbound ports allows the user to review the incomplete system to pick needed inputs, but prevents full analysis. One of Ford's main needs is to determine whether all input ports in a model are connected. If any input port has not been connected the final system will not work. Thus we chose to have the interfaces be explicit. Having explicit interfaces allows us to check and verify the completeness of the system, both for inputs and expected outputs.

However, within hand built sub-systems, we generated the interface from unsatisfied input (or required) ports, and any output ports used by the rest of the system. We did not allow the user to specify or restrict which components (or ports) within a sub-system would map to its interface. For the highest level of the System Architecture the user must specify the interface through the use of special components representing external systems. This is not an especially intuitive mechanism. A more intuitive model would make a single choice between generated and specified interfaces. However, the method does allow for a designer to maintain control of the interface to be ex-

posed to other systems, while making modularization within the system easy.

4.4. Expression of alternatives

For Ford, exploration of design alternatives is an important feature. With a large collection of components there are often multiple choices which fulfill the required interface yet still have different properties. Ford often uses a single architecture across several car makes and models, the differences coming in the choice of components to fill in that architecture. The abstract component representation must be able to express these alternatives. Since the number of alternatives grows exponentially, the scalability of constraint checking across the possibilities may be heavily dependant on representation. Even after constraint checking, a single system architecture can generate a large number of assemblies. Each of these will need to be stored and presented in some navigable manner. The user may also wish to review how constraint checks eliminated certain alternatives, requiring the software to present easily searched reports on the fate of all choices.

Vanderbilt used a special alternative component type to indicate choices. We used Acme's allowance for multiple representations to express choice points. Each representation for a component represents a potential choice. This mechanism is fairly heavy for a choice between two components. However, it does allow for choice points between entire subsystems – which may contain their own choice points.

4.5. Component linking

When the design must link to an existing set of components or models, the architecture description needs to support that link. Component linking is especially important when the architecture must integrate with external tools as discussed in 4.6. We used a property on each component to indicate which model to link to the architectural component. Every component to be used is placed within the project directory making selection via a browser easy. Unfortunately we did not find a way to prevent erroneous linkages – such as mechanical models with software models. Using a property rather than naming, allows the user to construct the architecture using a naming scheme appropriate for that design while linking to models named with a different scheme.

4.6. Analysis support

Additionally various properties are needed to support analysis of the system. We supported analysis on three different kinds of properties. The first kind is checked by AcmeStudio and includes architectural constraints, as well as properties added by the user directly to the architecture description. The second set of properties is pulled from the Simulink models – either analyzed through Simulink directly or added to the architectural description automatically. The component linking mentioned above enables Ford to take full advantage of Simulink's power on assemblies generated by Synergy. Lastly, our tool uses its own component characterization file. Properties added to the characterization can be analyzed through an analysis plug-in framework in Synergy.

We presented a number of questions at the beginning of the section. The solutions we chose to these issues heavily influenced the development of Synergy. Similar but separate styles make automatic conversion easy, while supporting the difference in detail. We used AcmeStudio's built in representations to support hierarchy and thus enable generation of large modules in an understandable fashion. The external interface for a design is specified by the user through use of a special component. Internally, Synergy automatically generates the interface between hierarchical levels. Design space is also supported by Acme representations, considering them to be mutually exclusive alternatives. Linking to existing models is provided by Acme properties and a convenient browser dialog. Lastly, an analysis plug-in framework is part of both Synergy and AcmeStudio. We felt that our architectural representation is rich enough to support many more analyses than we could develop. Thus we built Synergy to make it simple for the customer to add more without having to address representational issues.

5. Tooling Issues

Usable multilevel design software requires more than just appropriate representation solutions. The user needs to be able to construct architectures easily, view them, and analyze them – often using other software. In particular the developer must provide tool support for graphical editing, layout of generated views, and integration with other analysis tools. Lastly, as there are multiple views representing the same design, consistency checking is also important.

5.1. Visualization & usability

Software architectures can be described using an architectural description language, but a textual description may be difficult for users to read and understand. It is more natural to deal with the architecture graphically, where relationships between components can be made clear. Since we had adopted the Acme architectural description language, we could utilize the AcmeStudio design tool to provide a graphical visualization of the model. AcmeStudio also provides a way for users to

compose models visually. To create a new model, users simply drag-and-drop new components from a palette of pre-defined types. Beginning users will likely stick to the graphical user interface, however, advanced users may still edit the underlying Acme architectural description directly.

5.2. Layout Tools

Rather than generating source-code from a model, our modeling tool generates one model from another. Once generated, the user must be able to visually inspect the new model. The newly generated model contains significantly more detail than the source model, and care must be taken to keep the model readable. Components should be laid out intelligently so that connections between components remain clear. Clarity remains a challenge within these highly detailed models, where each component may involve twenty to thirty individual connections.

The right layout algorithm can help reduce the visual complexity of the generated diagram. Existing graph layout algorithms generally try to optimize a particular aesthetic quality of the diagram, for example, minimizing the number of connection bends and overlaps. Choosing an algorithm is generally a tradeoff between these different aesthetics, as well as running time and development effort. In the end, the team settled for a heuristic-based approach. The layout algorithm used here takes the original system architecture as a starting point, adjusting the positions of components to meet a minimum acceptable level of readability. This helps promote some correspondence between the system architecture and generated assembly. It provides the user some influence over the generated layout, but it relies on the user to place components intelligently within the system architecture.

5.3. Integration with analysis tools

The tool fills a small but critical niche in the model-based development cycle, so it should not be considered a stand-alone tool. Rather, it must work in cooperation with the other tools at the modeler's disposal. Foremost among these is the Mathwork's Simulink modeling and analysis package. Simulink provides the individual component models that serve as building blocks within our tool. Simulink also provides advanced analysis capabilities for verifying completed assemblies. Our tool must therefore be able to translate models to and from Simulink.

Unlike in the object-oriented world, the embedded systems community has yet to adopt a standard format for exchanging models between tools. In the meantime, tool developers must choose from one of several candidate formats. Acme's roots as a generic description language makes it a suitable candidate for model interchange. Translating the models between Simulink and Acme was fairly straightforward.

Once settling on the Acme ADL, the team was able to make advantageous use of the existing Acme toolset. Foremost among these was AcmeStudio, a tool for composing and analyzing Acme models. AcmeStudio is itself built upon the Eclipse open-source development environment. Eclipse provides a ready-made framework for integrating individual development tools under a common user interface. The cornerstone of this framework is a well-developed plug-in mechanism for expanding a tool's capabilities. The Eclipse framework allowed the team to deliver the core functionality in one plug-in, saving advanced forms of analysis for a later plug-in. Several more plug-ins supporting additional data formats and analyses will expand the tool's usefulness and reach.

5.4. Consistency Issues

With multiple models involved in the development process, consistency between models becomes an important issue. The tool gives the developer free-reign to tinker with the design at both the high and low levels of abstraction, so keeping everything synchronized remains a challenge. Changes to the system architecture or individual component models must be carried forward into the generated assemblies. Today, this is accomplished automatically by regenerating the assemblies. Reverse engineering is not supported, because developers are expected to make at most minor targeted changes to the assemblies.

We used AcmeStudio to provide graphical editing of architectures as well as the previously mentioned representational facilities. Automatic layout will always be difficult for larger, more complicated systems. To simplify the problem we took advantage of the user constructed system architecture as a starting point. Integration with other tools was a major requirement from Ford, which led us to select Eclipse and AcmeStudio as a base for plug-in development. Ford is focused on the architecture and pushing for automatic generation, thus Synergy maintains consistency in a feed-forward mechanism, regenerating the detailed model when requested.

6. Implementation status

Synergy has been built and delivered to Ford. Synergy is best considered a prototype at an alpha or beta development status. Ford is currently evaluating the tool, and has expressed interest in further development.

Synergy was created by five students in the Master of Software Engineering program at Carnegie Mellon University. Our experience report comes from one calendar year of work as part of the masters program. Synergy is

an augmentation of AcmeStudio and uses AcmeLib, both developed by the ABLE group at Carnegie Mellon. As of writing, Synergy contains about 160 source files using Eclipse, AcmeStudio and AcmeLib. To support Ford's interest in further development, another group of masters students will be working on Synergy over the next year. Areas for future work include:

- Improving support for hierarchy through interface specification or restriction.
- Development of more architectural design rules and heuristics.
- Improved component layout generation.
- Integration with Ford's enterprise wide repository of components, possibly including automated search.

7. Evaluation

While Ford has had considerable success in simulating individual software components, the complexity of building assemblies of components has prevented larger scale model development. We took a two-tiered approach to solving the problem. The user is able to work with enough abstraction to design a large system. Synergy then automates the time consuming process of connecting the components and checking constraints across alternatives. As discussed earlier, Synergy also handles issues such as component linking, consistency, and hierarchy.

The initial reaction from Ford has been very positive. In a demo we were able to build a system in twenty minutes that would have previously taken two weeks. They estimate that large systems which took six months previously, would now take about two weeks.

The Synergy project illustrates a number of lessons. There are several notational and tool support issues which any similar endeavor will encounter. To be useful the tool must support graphical construction of architectures and layout of any generated diagrams. If editing is allowed at all levels, consistency becomes a particularly difficult problem. The notation of the solution must contain representations for hierarchy or else the designs will be too complex to be understood. A means of specifying the system's interface is needed as well as a way of linking to existing component characterizations. For domains where a large number of components exist, expression of design choices and constraint satisfaction are particularly important issues. An extensible analysis framework will greatly enhance the system's value. For example, Ford and Volvo have similar design and architecture needs. However, the components and analyses they use are different. Synergy allows both companies to capitalize on the same tool by adding analyses specific to their needs rather than building separate tools.

Not all of these issues are challenging, but a successful project must have considered and decided upon a solution for them. The more challenging issues will be difficult to solve if they are not considered upfront.

Acknowledgements

The authors wish to acknowledge funding from Ford Motor Company for this work. In particular, we thank Ken Butts and William Milam for providing feedback on our work.

References

[1] Bass, L., Clements, P., and Kazman, R. *Software Architecture in Pra*ctice. Addison-Wesley, 1999.

[2] Bay, J.B. Model-Based Integration of Embedded Software. http://dtsn.darpa.mil/ixo/programdetail.asp?progid=38.

[3] Clements, P., Bachmann, F., Bass, L., Garlan, D., Ivers, J., Little, R., Nord, R., and Stafford, J. *Document Software Architectures: Views and Beyond.* Addison-Wesley, 2002.

[4] Garlan, D., Allen, R.J., and Ockerbloom, J. Exploiting Style in Architectural Design. *Proc. SIGSOFT '94 Symposium on the Foundations of Software Engineering*, New Orleans, LA, 1994.

[5] Garlan, D., Monroe, R.T., and Wile, D. Acme: Architectural Description of Component-Based Systems. *Foundations of Component-Based Systems*. Leavens, G.T., and Sitaraman, M. (eds). Cambridge University Press, 2000 pp. 47-68.

[6] Guttag, J.V., and Horning, J.J. (Eds) *Larch: Languages and Tools for Formal Specification*. Springer-Verlag, 1993.

[7] Karsai, G, Sztipanovits, J., Ledeczi, A., and Bapty, T. Model-Integrated Development of Embedded Software. *Proceedings of the IEEE* 91(1):145-164, 2003.

[8] Kruchten, P.B. The 4+1 View Model of Architecture. IEEE Software, 2(6):42-50, 1995.

[9] The Mathworks. Simulink 5.1. http://www.mathworks.com/products/simulink.

[10] Object Management Group. MDA: The Architecture of Choice for a Changing World. http://www.omg.org/mda.

[11] Perry, D.E., and Wolf, A.L. Foundations for the Study of Software Architecture. ACM SIGSOFT Software Engineering Notes, 17(4):40-52, 1992.

[12] Roshandel R., van der Hoek A., Mikic-Rakic M., Medvidovic N., Mae - A System Model and Environment for Managing Architectural Evolution, Submitted to ACM Transactions on Software Engineering and Methodology (In review), 2002.

[13] Schmerl, B., and Garlan, D. Exploiting Architectural Design Knowledge to Support Self-repairing Systems. Proc. 14th International Conference on Software Engineering and Knowledge Engineering, July, 2002.

[14] Shaw, M., and Garlan, D. Software Architectures: Perspectives on an Emerging Discipline. Prentice Hall, 1996.

[15] Shaw, M., DeLine, R., Klein, D.V., Ross, T.L., Young, D.M., Zelesnik, G. Abstractions for Software Architectures and Tools to Support Them. *IEEE Transactions on Software Engineering*, 21(4):314-335, 1995.