On Component Reuse in Embedded Systems

Hans Otto Trutmann
Swiss Federal Institute of Technology Zürich ETH
Computer Engineering and Networks Laboratory TIK
ETH Zentrum, Gloriastr. 35, CH-8092 Zürich, Switzerland
trutmann@tik.ee.ethz.ch

Abstract – **Real-time requirements, space constraints, and the tight interlock of functional and interface parts in embedded systems often degrade software reuse to scavenging components from previous work. The presented strategy separates concerns, supports development on a high level of abstraction, and relies on the assistance of code generators. This paper points out the benefits of this approach to software reuse. An architectural view separates functional aspects of a system from their connection to the environment. The functionality is modeled as a collection of cooperating extended state machines; it is attached to a physical environment using access routines in a static schedule. Appropriate tools favor the creation of detachable, self-contained components. Functional models that reflect the behavior of a component in the environment are especially fit candidates for finding generic solutions to recurrent problems. For example, a mechanical switch simply corresponds to its counterpart in the model, which will be used with the same confidence as its match. Elements needed to attach a functional solution to specific environments are difficult to reuse. Together with a code generator that creates efficient, deterministic and balanced schedules, such components can be included unaltered in various environments. These measures open up new possibilities for reuse, and because they help to raise the abstraction level for the development work, both software quality and developer productivity improve.**

I. SOFTWARE REUSE

In a broad sense, software reuse has been practiced since the beginning of computer programming, either by understanding and reproducing other people's ideas or by copying pieces of code directly. Today's systems have grown in complexity and size, but to some extent they all share components that are similar or even identical. The primary goal in reuse technology is to enhance productivity by reducing the number of unnecessary reinventions and reimplementations, which at the same time improves software quality. In addition to quality and cost considerations it is advantageous when a developer's familiarity with her tools can be extended to include a collection of often used components and their behavior.

Apart from a number of other classifications, technically software reuse is said to be either compositional or generative. Compositional reuse refers to bottom-up development using low-level components from repositories. Depending on the environment, these components can be functions, modules, classes, etc. Lack of generality or insufficient documentation often impede their unaltered use. For the same reason, reuse on a small scale is easier and much more wide-spread than reuse of complete systems, which also necessitates additional organizational and other non-technical measures. The application of generative techniques, on the other hand, is often restricted to specific environments, where a standard reference architecture is connected to equally standardized component interfaces [1]. Often a two-level approach is proposed with one layer describing what an application does, and the other one how it is accomplished. On the first level an abstract machine describes the fundamental operations for a specific application domain, and a program defines the use of these operations on the second level. The benefits are, that the abstract machine remains the same for similar types of applications, and that the expressiveness of the language used to describe a particular machine can be limited to what is needed within the specific application domain [2, 3].

If reused components are black boxes, the contents cannot be seen or altered by the reuser. In the case of white-box reuse the internal structure is not only visible, but it is usually also adapted to the new context. Clearly, both these solutions are unsatisfactory. The information available about the interface and the behavior of a black box do not answer all the questions a reuser might have, especially if the component has internal states, where the permissible traces in an input stream are determined by the structure and interoperation of a receiver. White boxes, on the other hand, are no different from mere copying, and gamble away the benefits of a single component to maintain and carry on with. Glass-box reuse with components that are used like black boxes, but with their internal structure revealed, is therefore a necessity in embedded system development.

II. CODE GENERATION

The difficulty of maintaining the original design and the emerging implementation synchronized presents a major problem in current development techniques for embedded systems. Code generators ameliorate these matters, as they operate from specifications, and in that way tightly link an abstract system representation to the target installation. Generators are also indispensable if one has to choose a logical architectural structure according to the problem at hand, and not just aligned with a momentarily envisaged implementation structure. Many abstract structures that are helpful in finding isolated problem areas and dealing with them during the development process are not implemented advantageously the same way. Consider for example separate processes that were introduced as a means of structuring. If required to run on a single processing entity, they can be efficiently cast into a combined single task, which avoids the overhead and possible non-determinisms associated with run-time scheduling methods. The explicitly specified connections between such components are used to keep dependencies under control, but the connectors themselves will not be present as active elements in the implementation. These rather contradictory views can only be reconciled with the assistance of tools. Only tools let developers

work with the abstract categories they are at ease with and which they consider for reuse. If abstraction is the prerequisite to software reuse [4], then code generation closes the gap between an abstract model and its implementation. However, all the advantages of generated code are only of practical use, if such code can compete with manually written variants, i.e., it ought to be of small size, execute fast, adapt easily to vastly different hardware and operating environments, etc. [5]. Additionally, the utilized methods and supporting tools must make work easier, instead of being an additional burden on the developers. They can only be justified if they further the entire development process, with benefits for the system architect as well as for the specialists working on the details of functionality and integration [6]. Because appropriate abstraction rules always reflect the purpose of a model, the specific demands of different problem areas are best met with dedicated models and tools.

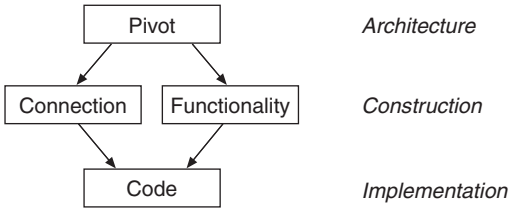

Fig. 1. Problem areas in embedded system development

In the context of software reuse application generators are investigated mostly because of the productivity gains offered in comparison with component-based reuse methods. The emphasis is on the generator itself and the elements it contains to create reusable patterns, similar to object-code patterns a compiler substitutes for high-level language elements. Compositional reuse on the other hand is based on the notion of parts, that are themselves made up of elements on a lower level. A combination of the two approaches yields the benefits of high productivity (because no coding errors occur), and problem oriented development with components on a high abstraction level, expressed in appropriate notations [7]. Carefully built components are perfectly suited for reuse, and because the number of errors left decreases with each reuse, confidence in them rises accordingly.

III. ARCHITECTURE AND CONSTRUCTION

Established engineering disciplines benefit a lot from the clear division of different phases in the development process, as for example the process of building a house is completely detached from the work on the architectural drawings. Software engineers have also begun to adopt this approach, with some disagreement on the role of modeling, the form of the models, and the means of relating architectural designs to the construction of its elements. Just like in the example, specifying the architecture of an embedded system precedes the realization of its components, whether this implies a mere sketch with boxes and lines, or a design based on a formal model. Furthermore, architecture and design are top-down activities in both cases, whereas the construction of the required elements is more favored by compositional, i.e., bottom-up approaches.

Ideally, embedded system development is problem-oriented, so that it abstracts completely from any envisaged implementation. An architectural view therefore unites logically related components into groups, and defines the necessary interconnections between these groups. With an eye on a feasible implementation it is advisable, however, to strive for a finer-grained subdivision of large groups. This top-down view of the system to be built is the *pivot* which forms the basis for the (compositional, i.e., bottom-up) construction work. The construction is undertaken in two separate problem areas, one of them dealing with all the functional aspects, the other uniting elements concerned with the actual embedding into a specific environment. Because the pivot is less prone to frequent changes than the problem areas themselves, development may progress independently on the grounds of the previously defined architectural elements.

The *functional problem* comprises all the behavioral aspects of a control system. The interactions with the controlled processes themselves are accomplished via shared phenomena (events and actions in an event driven environment). The appropriate abstraction for solving this problem is to leave out any details not pertaining to the relation between the course of events in the environment and the inner states of a model. The *connection problem* unites input, the extraction of events contained in the new data, the calling of the functional reactive block, and the output. The functionality of the connection that must be built is implementation-oriented with a regular sequential structure. A useful abstraction for this problem concentrates on information transport within a generalized form. Based on these precepts development takes place with problem-oriented methods, that offer separate solutions for the functionality and the connections to the physical processes. The firm basis of the architectural pivot guarantees, that the interfaces in any generated implementation will match.

A. The Pivot

Complex systems are built as a collection of a number of different more or less independent blocks. It is common practice to define their interrelation only implicitly in the interfaces of the active components. As a result, the system structure is hidden, and component reuse is complicated by embedded references to other components. A clearer picture of system architecture requires concepts that treat not only *components*, but also the *connectors*[1] between these components as distinct entities [8]. Various languages have been developed that not only cover syntactic aspects of components and connectors, but also permit to perform dependency and consistency analysis by incorporating behavior as formal annotations [9, 10]. For analysis, formal descriptions of the components and connectors are used to reason about system behavior. In the construction of a working system, functional models (the foundation for any implementation) are associated with the components of the architecture. Systems developed in this way are based on an explicit structural skeleton, which helps to maintain consistency among the various elaborated models.

[1] cf. architectural element *connector vs. connection problem*

The pivot specification does not need the expressiveness of some of these architectural modeling languages, considering the declared separation of functional from connective aspects. This separation also permits to hide the internal refinement (the local architecture, as it were) of components on the architectural level. Therefore, the pivot contains only a minimum of different component types as well as ways of component interaction, hence connector types.

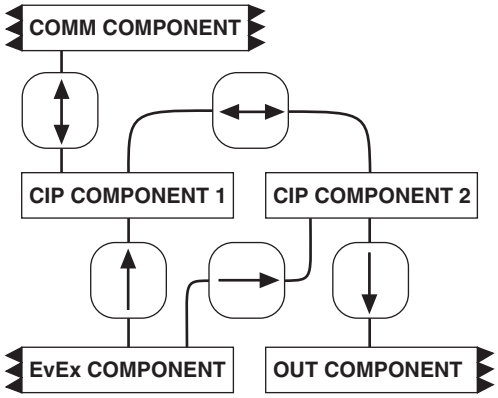

Fig. 2. Example pivot

The emphasis in the presented approach is on its applicability to the development of real-world systems. For this reason, implementation details, such as data type definitions have been introduced into the notation, rather than complete formal specifications. Channels are the interfaces of components and connectors. They define syntactic properties, but may also contain architectural constraints based on the element's internal specification, for example protocols, sets of messages or even traces that can be accepted and/or produced.

1) Component: Components are elements that must later be refined as either parts of the functional or connection problems. The functional components can interact symmetrically via asynchronous channels, and they can be attached symmetrically or asymmetrically with connective components. The channel properties of functional components are derived from the behavior of their internal refinement. It depends, for example, on the transition structure of the process connected to a channel, if this channel can operate according to a specific communication protocol.

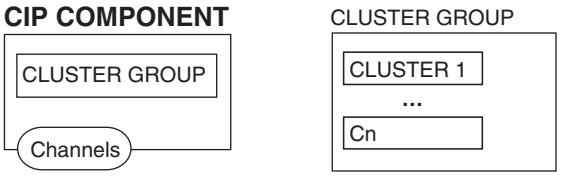

Fig. 3. Functional model component

Connective components accomplish the chores of communication message handling, data input and event extraction, or data output. With the exception of the properties of communication components, which depend on the chosen communication paradigm, the channel properties of input and output components remain structurally stable, only their types and sets of messages vary.

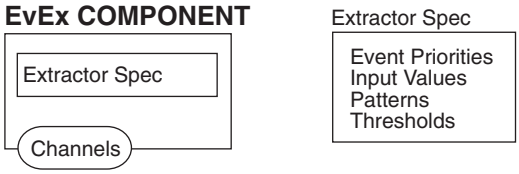

Fig. 4. Event extractor component

2) Connector: Connectors mediate the interaction among components, i.e., they restrict the behavior of the attached components with protocols, sets of messages or permissible traces. On a conceptual level the roles of a connector are seen as independent processes. Actually, they guarantee syntactic and semantic compatibility between the components they relate, and will not be present as active elements in a final implementation.

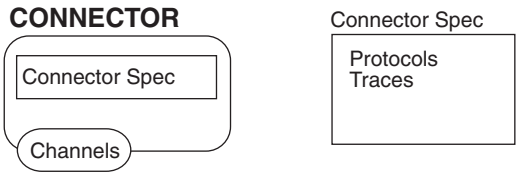

Fig. 5. Connector

B. The Functional Problem

To solve the functional problem means to refine the relevant components of the architecture. Within the constraints of the component's channel properties, the refinement can be structured in a problem-oriented way. The functional reactive behavior of the system is modeled with CIP Tool in the form of extended cooperating state machines [11, 12]. CIP models are built with a notation of synchronous processes enclosed within asynchronous clusters. Clusters are the coarse-grained elements combined in the cluster group of an architectural CIP component. In order to have a comprehensive view of the system to be built, all the cluster groups defined in the pivot can be dealt with simultaneously, and their interrelations defined on a functional level of abstraction. This is especially important for implementations that may be placed into a multiprocessor environment. The method permits to deal with both event-based and hybrid systems with integrated quasi-continuous processes. Event and communication messages from the environment cause a transition in the receiving process which activates other processes for his part. Transitions may generate pulses to other processes, as well as action and communication messages for the environment, and they can also contain additional code for data manipulation and control functions. Hierarchical system decomposition is further supported with a powerful master-slave concept.

Chain reactions initiated by a message cannot be interrupted; their execution depends on the message itself and the internal states of all activated processes. A local architectural description (interaction net) together with explicitly specified execution cascades set upper bounds to these chain reactions and prevents the specification of cycles. Code generation for complete models is straightforward, and produces efficient and deterministic implementations.

A system is built with only a few essential elements (processes and channels embodied within clusters), which are put together with appropriate glue parts (interaction net and pertaining cascades with pulse translation, message translation, inspection and master-slave assignments). The basic elements are self-contained and keep their interface properties even when detached from the surroundings, else they could not be integrated into reusable components.

Experience has shown, that well-built process models can reflect the behavior of components in the environment so well, that developers often do not need to examine their internal structure in order to work with them; components such as switches, drives and so forth are used with the same confidence as their mechanical counterparts.

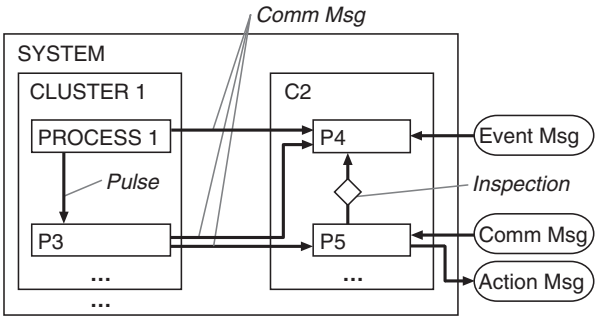

Fig. 6. CIP system structure

1) Process: Processes are the building blocks of any functional model. Their internal state machine can be triggered by other processes through synchronous pulses or by messages received asynchronously from the environment. Process interfaces consist of ports and inspection panels. Ports contain the sets of pulses and messages a process can consume and produce, whereas traces are defined by the process' transition structure. Message and pulse ports can be hooked up to ports which have the same specific type attributes. Optional inspection panels define how state information is made available for reference to other processes in the same (synchronous) cluster.

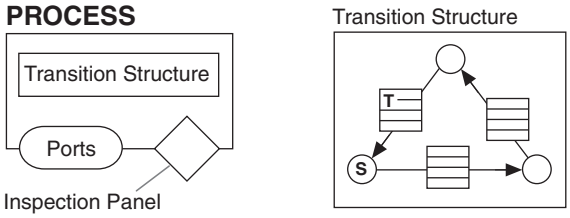

Fig. 7. Process element interface and internal structure

2) Channel: Channels provide asynchronous links across the cluster and system boundaries. If their respective type attributes match, their ports can be attached to process ports and to the enclosing shell. The properties of those channels, that are attached to the shell, must match the interface definition at the system architecture level. The transition structure of a process hooked up to a channel defines the traces that this channel can accept or produce. This information could be used for consistency checks on the architectural level, but this has not been attempted so far.

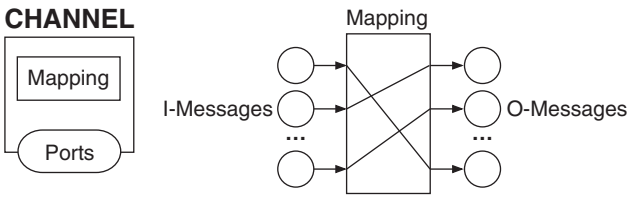

Fig. 8. Channel element interface and internal structure

3) Cluster: Clusters unite sections of a system with synchronous internal behavior. In addition to processes and channels, clusters contain translation elements to accomplish the required interoperation. Master-slave relations among processes can be established, where various modes of a slave process with different transition structures are associated with the actual state of one or more master processes. Clusters are not self-contained entities, but serve as brackets to indicate the synchronous interaction of the processes they accommodate.

Because the interface properties are defined within the process and channel elements, their ports can be readily connected with pulse and message translators. It can be argued on the same grounds as in the case of the architectural pivot, that connectors must be distinct objects also in the functional model. Even if they do not appear as boxes, they are nevertheless well-defined entities, accessible via translation browsers.

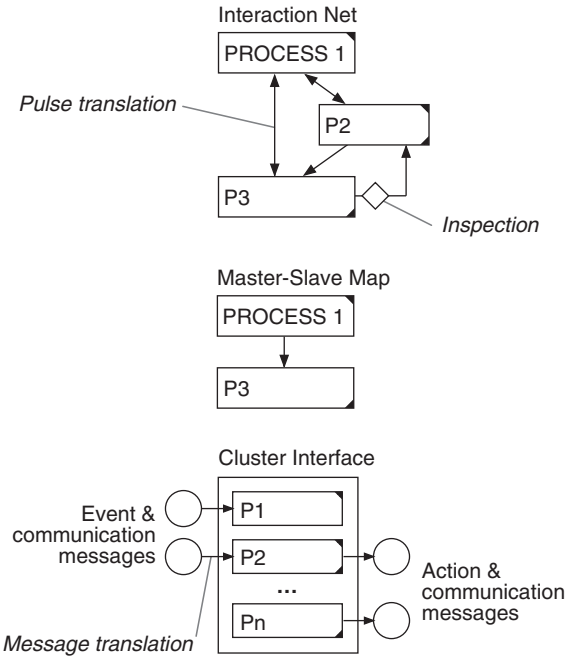

Fig. 9. Translations and master-slave map

C. The Connection Problem

Interfacing functional code to the existent conglomerate peripheral hardware is a difficult task. Reuse in this context is usually limited to the actual access routines (i.e., code), and it is only possible if the hardware platform remains the same. To make peripheral access routines work together or side by side in a notoriously heterogeneous environment requires experience and much labor, and the expansion of a hitherto stable solution can easily end in a new beginning.

The main reason is, that input and output operations on many devices are fairly complicated. They must be made with multiple accesses, possibly also requiring delay periods in between, or they may introduce conflicts concerning the use of common resources. In addition, considerable variations with regard to data width, channel number, multiplexer setup, and possibly filter and amplifier settings must be accounted for.

1) Input: Clearly, an input signal is associated with a respective processing element. This connection is important on a conceptual level, but tends to complicate matters when it shows in a system's implementation. Instead of using individual tasks and dynamic scheduling for peripheral device accesses, access routines of several devices are interleaved, and cast into a single task with a statically determined schedule. Apart from introducing abstraction into the construction of input systems, this approach offers the benefits of reduced runtime overhead, minimal processor idle time, and practically equal processor load for each invocation. The generator used to create this solution operates on a specification, using a device's timing requirements and its low-level functions for initialization, setup, access, conversion, verification and filtering. Output signals are specified similarly.

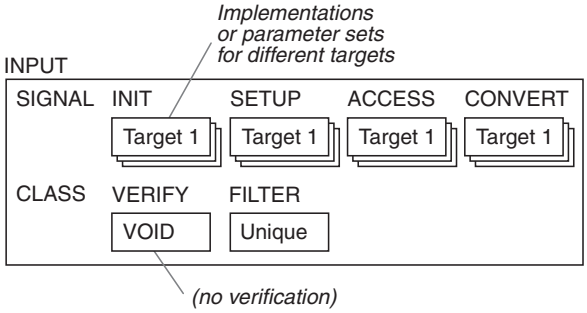

Fig. 10. Input signal specification

2) Event Extraction: Another chore of the input system is the extraction of events that are contained in the newly acquired data. In the implementation, an optimized algorithm works on tables generated from a specification. The specification contains events and the conditions they are associated with. Discrete events (i.e., those related to sensors with binary type signals) are generated on the appearance of specific patterns. Other events are linked to threshold levels in quasi-continuous signals. For these events, the extractor maintains the correct sequential order. Events are placed in different priority lists, one of which will be selected dynamically at run-time.

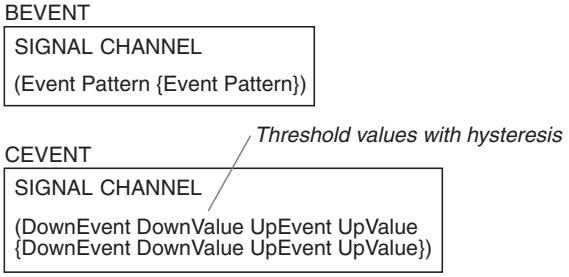

Fig. 11. Event specification

IV. REUSE COMPONENTS

Different kinds of self-contained, detachable components have been identified within the two problem areas. Because of the high abstraction level, the reusable components do not differ structurally from the ones created specifically for an application. Nevertheless, are these really the components to consider for reuse? From a developer's view, there exists a connection between various elements, a connection that crosses the problem area borders and even involves the architectural pivot. A small example with the constituents of a simple switch illustrates this.

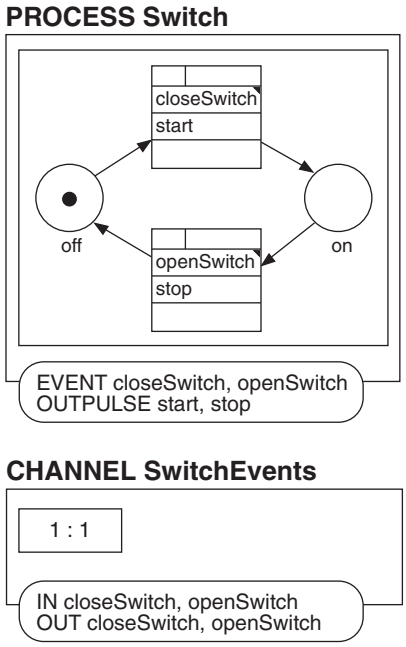

Fig. 12. Functional switch elements

The simple functionality of the switch is appropriately expressed as a process with two states. The transitions, triggered by event messages *closeSwitch, openSwitch,* respectively, produce the outpulses *start* and *stop.* The outpulses remain within the scope of the cluster the switch process will be placed into, whereas its event message input must be fed from a channel element that provides a link across the cluster boundaries.

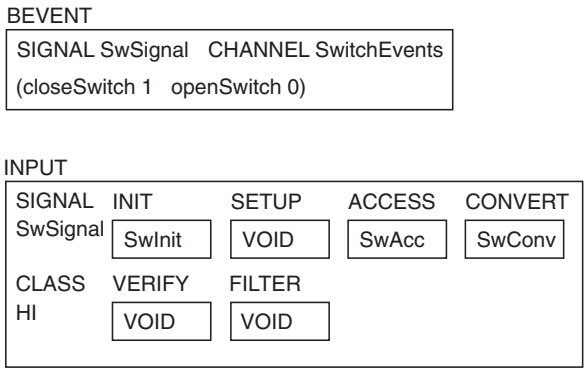

Fig. 13. Connective switch elements

The components defining the connection of the functionality to a particular environment consist of the specifications for the events and a suitable input signal.

A prospective reuser of the switch could look for all these elements and reassemble them as required, but it is more helpful to preserve their interrelation right from the beginning. For this reason, reuse components must be aggregates that unite elements meant for joint application. An aggregate component is structured along the lines of the architectural view. It contains connectors, or at least connector fragments that specify the architectural reuse conditions for an aggregate.

As the low-level input or output signal specifications are generally hardware-oriented and thus only loosely coupled with the other components (in the switch example, any binary device could be substituted), they are replaced in the aggregate by a list of references to previously used specifications. If a fresh reuse cannot be realized with one of the recommended sources or sinks, it causes an additional entry in the list.

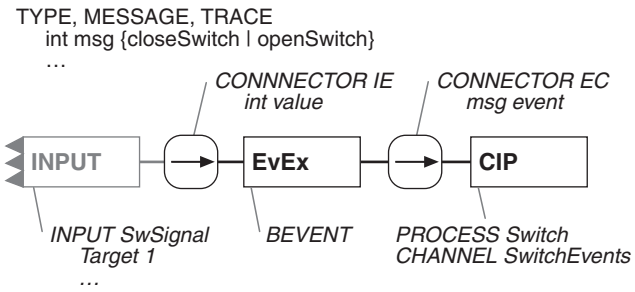

Fig. 14. Switch aggregate component

Aggregate components must unite elements from different modeling and specification environments. For reasons of compatibility, they are saved in textual form, at the same time making them accessible to conventional repository management. On reuse, their constituents must be taken apart, converted and placed into the respective environments again. This is possible within any architecture that can satisfy the reused component's connector requirements.

V. CONCLUSIONS

The presented approach emphasizes the usefulness of distinct problem areas in embedded system development. The separation of concerns permits the application of problem-oriented methods and tools, tuned to the specific needs of each area. Solutions on a high level of abstraction offer the opportunity to isolate elements with well-defined behavior and interfaces, a prerequisite for software reuse.

The functional models are built with and retain a close connection to the real processes in the environment. This relation makes it easier to find generic solutions for recurrent problems. The tedious chores of dealing with peripheral devices are alleviated by specifying all connective issues instead of programming. Thus, low-level routines can be reused as parts of generated input schedules for different implementations.

Automating the link between a high level of abstraction and the implementation eliminates situations, where an otherwise fitting component cannot be used because of the ways it has been implemented. Namely, if abstract component definitions are associated with manually written code fragments, it is not sufficient to meet the architectural constraints. The code must be appropriate for the envisaged context as well, or only the component's structure could be reused. Code generation greatly increases the chances for component reuse without adaptations; it thus helps reduce the number of variants to monitor and maintain.

A number of real-world applications have been realized on the basis of separate problem areas. However, in these cases the presented methods and tools were used side by side, burdening the system architect with the details of cooperation. The introduction of the pivot specification as an architectural basis does not only aid in integrating the different problem areas, but it also allows for a comprehensive definition of reusable components. Not only do such aggregate components unite all the elements required for reuse, they also preserve their order and interrelation. Reuse is made easier, with immediate benefits for software quality and productivity.

Further work is required to improve the integration with available repository management utilities. Instead of subscribing text files that must be imported in an additional step, subscription from within the modeling environment is envisaged. Whereas black box reuse is not attracting developer groups working within the same organization, it may well become an issue, if components are to be sold for reuse by someone else. To this end, suitable means to extract the essence of a component without revealing proprietary information must be investigated.

Acknowledgement: The author wishes to thank Hugo Fierz and Samarjit Chakraborty for their comments.

REFERENCES

[1] J. Sametinger, *Software Engineering with Reusable Components,* Springer-Verlag, 1997.

[2] S. Thibault and C. Consel, "A Framework for Application Generator Design," in *Symposium on Software Reusability SSR'97* (M. Harandi, Ed.), Software Engineering Notes 22(3), 1997, pp. 131-135.

[3] U. Thoeni, *Programming Real-Time Multicomputers for Signal Processing,* Prentice Hall International, 1994.

[4] P. Wegner, "Varieties of Reusability," in *Tutorial: Software Reusability* (P. Freeman, Ed.), IEEE Computer Society Press, 1987, pp. 30-44.

[5] HO. Trutmann, "Well-Behaved Applications Allow for More Efficient Scheduling," in *Proceedings of 24th IFAC/IFIP Workshop on Real Time Programming WRTP'99,* Schloss Dagstuhl, 1999.

[6] P. Naur, *Computing: A Human Activity,* ACM Press/Addison-Wesley, 1992.

[7] HO. Trutmann, "Generation of Embedded Control Systems," in *Proceedings of 23rd IFAC/IFIP Workshop on Real Time Programming WRTP'98,* Shantou, 1998, pp. 99-104.

[8] M. Shaw, "Procedure Calls are the Assembly Language of Software Interconnection: Connectors Deserve First-Class Status," in *Studies of Software Design ICSE'93* (D. A. Lamb, Ed.), Lecture Notes in Computer Science Vol. 1078, 1996, pp. 17-32.

[9] R. J. Allen and D. Garlan, "A Formal Basis for Architectural Connection," *ACM Transactions on Software Engineering and Methodology,* 1997.

[10] D. C. Luckham, "Rapide: A Language and Toolset for Simulation of Distributed Systems by Partial Orderings of Events," *DIMACS Partial Order Methods Workshop IV,* Princeton University, 1996.

[11] H. Fierz, "The CIP Method: Model-based Construction of Real-Time Components," submitted to *European Software Engineering Conference ESEC'99,* 1999.

[12] *CIP Tool User Manual,* CIP System AG, Solothurn, 1996/99.