Flexible Small Scale Parallel System in the Development of a Hybrid Vehicle

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

Abstract *It is common in the development of complex mechanical systems to break up a machine into its components, recombining them later. Often these transformations happen repeatedly in the process and of course the control system experiences these changes along with the mechanical components. In such cases control system design has to take into account two very different modes of operation: components running separately on their own test beds during development on the one hand or united to form the final installation on the other hand. Transformations between modes must be straightforward and with as few changes as possible to the control system. To this end a flexible multiprocessor solution that mirrors the machine structure is proposed. Standardized data input and output are supported by software generators easing the task of adapting to ever changing environments. The entire functional program code is generated automatically from specifications and fits into an equally standardized runtime environment. Designers of the mechanical elements work with a dedicated graphical software tool and are thus able to concentrate on their actual problems instead of getting involved with control system details. The concept has been proven in the ETH project Hybrid III, targeting the fuel efficiency of private vehicles.*

Keywords: automotive, distributed system, embedded system, real-time, specification.

1 Introduction

The development of any complex mechanical system suffers from various uncertainties concerning the control system: the number and variety of inputs and outputs is largely unknown as are their data rate requirements. Elements of such systems are likely to be developed by different persons in separate locations. In addition to being operated in composition, they must also run on their own on separate test beds. Transformations between combined and separate modes must be straightforward and with as few changes as possible to the control system hardware and software.

If space restraints and/or energy consumption are an issue, data input and output facilities that are only partly present at the individual test beds must be pooled in the case of the entire installation.

As designers of the mechanical components are by preference computer users who do not like to get involved with control system details more than necessary, their interest concentrates on an easy to use, general purpose environment that permits close attention to the actual development work. This environment can greatly improve the project engineers' performance and must at the same time hide all but the necessary details from access by such a user, also adding invaluable benefits in overall system security.

This complex situation can hardly be handled without the help of efficient tools. Specification methods and corresponding software tools deal with the functional properties and configuration tools help in creating the low-level structure for the peripheral connections. The complete functional code for a particular implementation is generated from specifications automatically. The high level of automation designed into the tools relieves the developers from the tedious chores of mere programming and allows them to devote their attention to the functional issues.

2 Logical Structure

In just the same way as the mechanical system the control system comprises a number of singular, loosely cooperating blocks. These so called clusters form widely autonomous subsystems which are capable of executing complex operations all by themselves. It is only possible to develop and test subsystems separately because the mechanical components belonging together are united within the control system as well. In the same manner clusters also permit hierarchical system decompo-

sition, restraining any direct communication of leaf elements with each other.
Input and output devices are located within the scope of a specific cluster, namely the one who controls the mechanical component where said devices are installed.

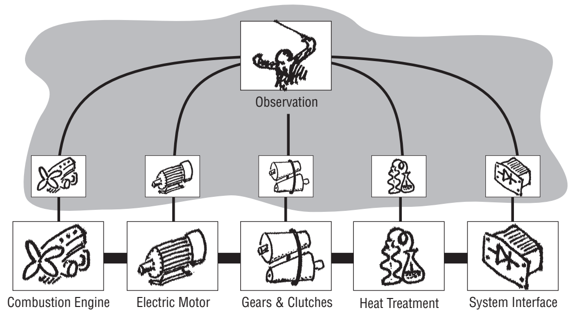

Figure 1: Logical structure[1]

Along with the aforementioned working clusters an additional cluster with higher level functionality supervises and coordinates the individual subsystems. This representation abstracts completely from lower level functionality, but it is this structure that lies at the heart of what specialized engineer groups use in the development of their respective part of machinery and pertinent control system.

With the star topology it is perfectly possible to set up a partial system without any interface changes to the cluster in question and for any specific test bed installation.

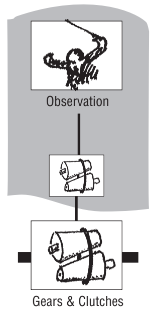

Figure 2: Partial system

3 Implementation Structure

In a functional implementation the cluster partition and interconnections outlined above must be adapted to the features existent in a hardware environment, i.e. the processors and the peripheral interface. Notably the star structure cannot be accomplished for an arbitrary number of working clusters. In the current implementation it is replaced by a double ring, ensuring that all partners can be reached in two ways.

While the logical structure guarantees interface consistency of any partial with the full implementation, assigning separate processor nodes to all the clusters guarantees that performance will equally remain constant in both types of implementation.

It would be costly and inefficient to fit every cluster with its own data input and output mechanism. The supplementary cluster *IO*, hidden from the logical structure, handles peripheral data input and output for the entire system. This cluster is also included in the ring structure and in addition conveys input data to all the other clusters simultaneously through an efficient hardware broadcast mechanism *BC*.

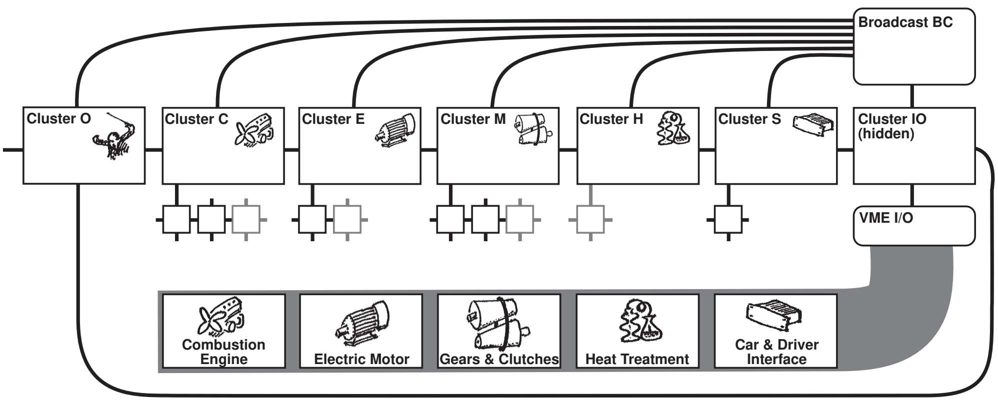

Figure 3: Implementation structure

[1] Cluster names refer to Hybrid III described in section 5

3.1 Cluster Tasks

Global functions (system operating modes, error handling and the like) are controlled by the process *SystemMaster* placed above all the others and situated within the cluster Observation *O*; this process thus handles all the typically slow activities with reference to the whole installation. At the same time the cluster supervises the overall system communication and is in charge of the interface to the developers' work station as long as it is required.

All the working clusters, i.e. the ones associated with a piece of machinery, contain a *SubMaster* to execute all the cluster's local functions. Therefore fast operations remain within the scope of a single cluster where they can make use of locally added resources if necessary.

Sometimes the functional membership contradicts the constructive realities. Because the control system is modeled after the machine construction, all elements are connected with those clusters to which they belong constructively. The imposed structural restrictions require a *RemoteService* mechanism to make elements accessible to those clusters where they belong functionally.

Along with the above clusters contain an arbitrary number of processes for the actual control functions, by far the bulk of a cluster's tasks. All these processes are defined and specified by the specialists' teams with a specification tool.

3.2 Processors

Structural flexibility without adverse effects on performance encouraged an implementation with parallel hardware, while a flat architecture offered simpler solutions regarding common resources. A parallel approach was chosen because cluster performance can be adapted to changing needs with additional local processor nodes. Because input and output are handled centrally the equally variable demands there seemed the smaller problem. The processor nodes are Transputers[2], RISC-processors with all the facilities needed for a small-scale multiprocessor system. The number of nodes differs from one installation to the other, but the link connections are static.

The possibility of dividing a complex task into individual processes is ideally tailored to our system demands; each task block is controlled by a group of processes on an independent computer, which communicates via links with his partners. The hardware directly supports communication and process scheduling, so the application can be built directly on top of the hardware without the need for an operating system.

3.3 Communication

The logical star topology forms the base of all communication in the complete system. The ring behaves identically with the aid of various protocols for point-to-point and broadcast connections. A special *Communicator* process on each cluster handles ring traffic from and to this element and forwards other messages. The ring mechanism is also used for output data and all in all for any kind of traffic in the system.

3.4 Input and Output

Because input and output signals belong to a specific cluster just as the respective sensors and actuators there exists another structural conflict, this time between the logical structure and the realities of the peripheral hardware. Generally these are multichannel devices and it is thus inevitable to distribute the available channels among various clusters. Different speed classes for the input signals make things even more difficult (e.g. 1 kHz, 100 Hz, 10 Hz side by side).

Data input and output are standardized. To balance the work load on the cluster *IO* and on all the other clusters for that matter, carefully chosen groups of input signals are read at a given point of time. Together with the signals of the highest speed class which are read in every cycle, a selection of signals of the other speed classes is processed. With a large amount of signals and complicated restraints on the hardware accesses automatic generation of the program code from specifications is a necessity.

Inputs occur in fixed intervals (i.e. synchronized), outputs immediately after a cluster has produced a new output value (asynchronously). Still a burst of output values is connected to the preceding group of input values; namely are they located together between two ticks of the input system. If the capacity limit of the arrangement is exceeded, this is no longer the case!

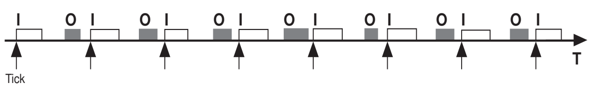

Figure 4: Input and output cycles

An individual cycle begins with the cluster *IO* acquiring a group of input data. The raw data are filtered and converted to physical units, then forwarded to the working clusters. All clusters receive an identical copy of the newly acquired and converted data simultaneously through the broadcast mechanism. Sometime later, after the working clusters have finished their processing and calculated new values, output data from each cluster arrive at the cluster *IO*. These data already show the correct hardware formats and can therefore be output immediately. Input and output algorithms as well as related filter and conversion routines are complemented by sending and receiving mechanisms in the clusters involved. They are standardized and use particular structures in each specific implementation that are generated to measure from configuration data.

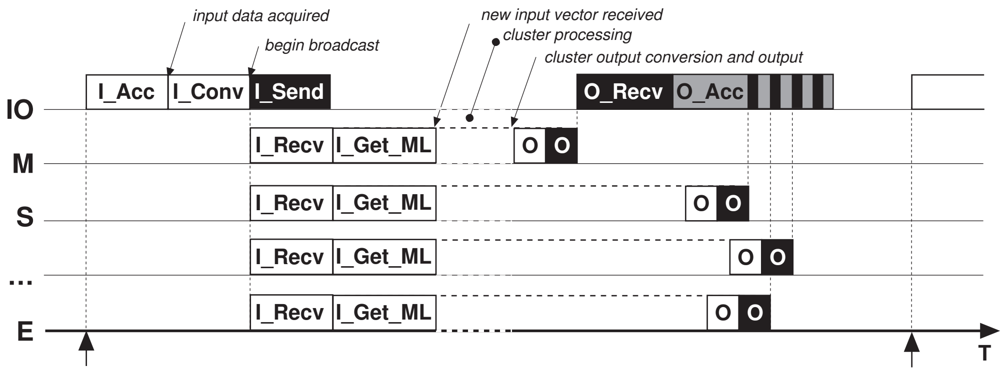

Figure 5: Synchronous input and asynchronous output

4 Development

The final application results from a combination of three different development strands:

- low-level mechanisms such as input and output, communication and the runtime environment for the generated code are written in C and made available through libraries,
- parts that are event driven as well as the overall system structure are specified in CIP Tool, a specification environment with a graphical notation. C code is then generated automatically from these specifications. Solutions can be tested against numerical models before attaching the control system to the mechanical components,
- algorithms for continuous control are developed separately with appropriate tools and integrated as complete building blocks.

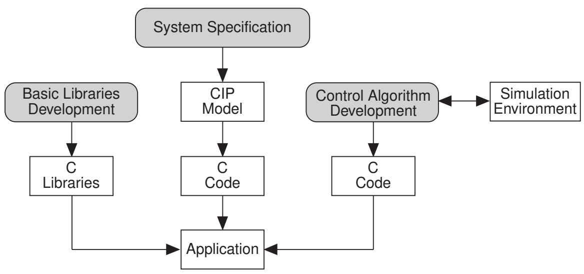

Figure 6: Development strands

The first strand is especially precarious regarding efficiency and system integrity; its internals are not generally accessible. Sensitive parts of the other two are handled by generators for the same reason.

The separation of low-level chores from system functionality is advantageous in general, because in this way only specialists for the particular parts (overall system structure and functionality, control technology, low-level programming) deal with details, responsibilities can be clearly defined and efforts concentrated. Interfaces as well as basic data propagation methods are standardized and laid down beforehand.

4.1 Environment Generators

Software generators control access to low-level facilities, hide unnecessary details and enhance system reliability very much. From a developer's view system design, implementation and testing thus take place in an integrated environment.

Any developer group starts out with configuring their environment, i.e. communication as well as input and output as required for the peculiar case. Number and kind of sensors and actuators vary greatly depending on the installation, just as do their associated data rate, resolution and filter parameters.

The code generated from these configuration data, together with the basic libraries, provides developers with a runtime environment for the functional part that is equally generated, this time from the CIP specification.

While the environment configuration may be modified occasionally from then on, the actual work is done within the specification tool, changing functionality time and again, usually to the last moment of the development process.

4.2 Low-level Mechanisms

The system-wide communication tasks for ring and broadcast are present on all clusters with appropriate processes. Buffers separate global and local sections. Throughout the system all of these processes share the same code. Moreover also all the working clusters are identical, arranged around the one process running the proper functional code generated from the specification.

Considerable manual work went into these processes, because system efficiency depends heavily on their performance. The surprisingly simple structure with only a few different processes meets the requirements of all the working clusters. Code generated from the CIP specification fits into these surroundings regardless of its functionality, with the communicating and discrete event extraction processes depending on structures turned out from configuration data.

Consequently, the low level code does not experience any changes during the development of the system's functional properties. Because of this and because of the limitations enforced by the specification tools unenlightened contributors are less of an issue regarding system reliability.

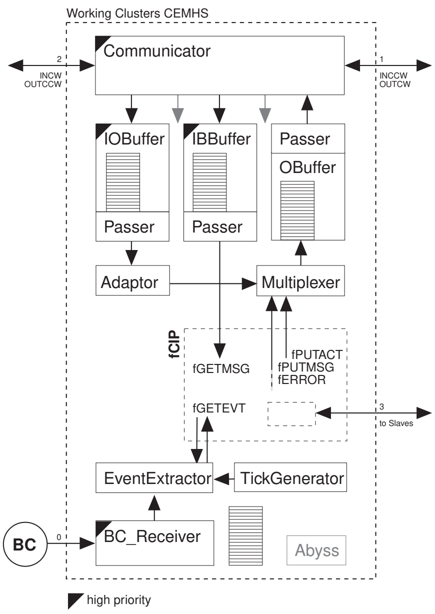

Figure 7: Working clusters: process structure

The cluster *IO* had to be split in three nodes due to heavy processing loads. Node *IO_Acc* performs all the bus accesses, node *IO_I* filters, converts and broadcasts input data, node *IO_O* receives output data from the ring, buffers and forwards them to node *IO_Acc* for output. This node accepts output data only when no inputs are pending. This is the bottleneck of the installation and therefore process priorities had to be carefully balanced to get optimum performance. Structurally, the three processor nodes appear equally simple, yet they perform the big and complex switch statements described in 3.4.

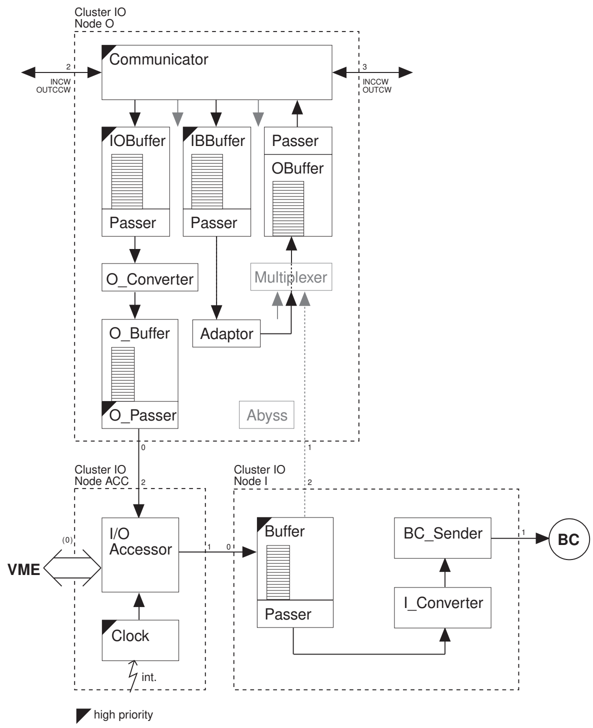

Figure 8: Cluster IO: process structure

The cluster *O* has a similar structure, but with a lot of communication chores instead of its own input and output. System process and link allocations remain the same for all kinds of implementations. Additional working clusters are simply added within the ring according to requirement.

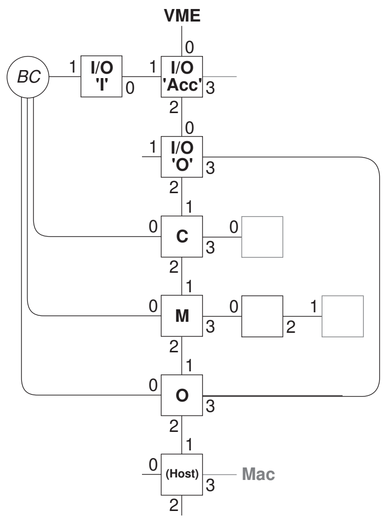

Figure 9: Partial implementation

4.3 Communicating Interacting Processes CIP

Communicating Interacting Processes CIP[4][5] is a method and corresponding tool for developing real-time and embedded systems. It describes control systems in a multiprocessor environment with a notation of synchronous *Processes* and asynchronous *Clusters*, together with the necessary communication mechanisms. Although the model is event-based, quasi-continuous processes, such as control algorithms, integrate well.

CIP specifications of reactive systems evolve constructively as a composition of extended finite state machines. Formally the model is the product of these state machines, forming a single machine in a multidimensional state space. In a cluster *Events* (Ev) or *Messages* (Co) from the environment cause a transition in the receiving process, which can activate other processes for his part with synchronous pulses (*Outpulse* Op and *Inpulse* Ip). Static cascades propagate these pulses from one process to the other. Transitions can generate *Actions* (Ac) or send messages to influence the environment. C code for added functionality may be placed in any transition, a convenient way to incorporate algorithms obtained from other sources.

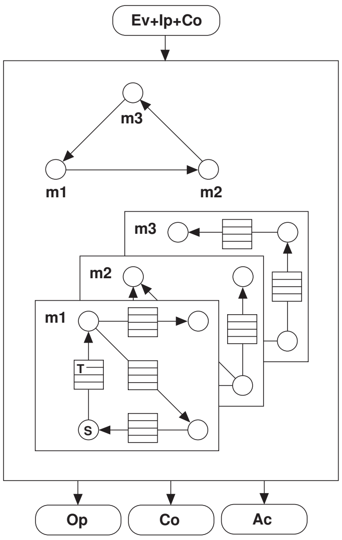

Figure 10: CIP process structure

A chain reaction initiated this way cannot be interrupted and defines a single state transition of the cluster; the tool prevents it from being cyclic. While processes located together in the same cluster exchange messages via pulses synchronously most of the time, asynchronous communication inside a cluster is also supported.

So cluster tasks are built with elementary reactive blocks, with either *Model Processes* that mirror parts of the outer system reality or *Functional Processes* which deal with the cooperative aspects of the components. The internal interaction between all these processes defines the comportment of the reactive system in its entirety. In addition to that different *Modes* (m1…m3) permit changes in the dynamic behavior of single processes. The currently active mode depends on the state of a moderating process. Modes therefore add the concept of behavior hierarchies to the simple behavior abstraction.

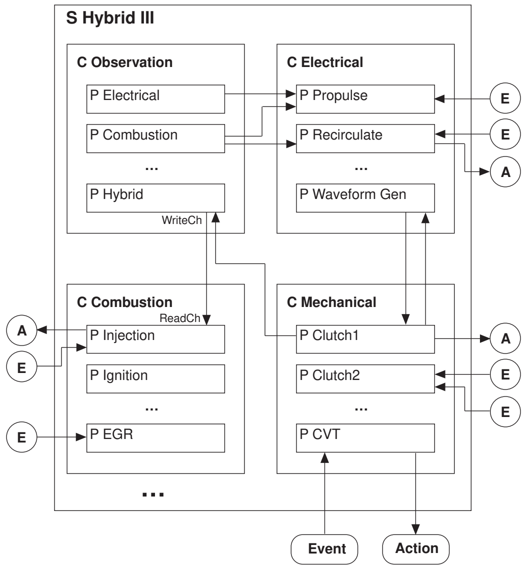

Figure 11: Hybrid III project structure (excerpt)

Composition as a development principle is radically opposed to the common top-down approach, where assumptions about the top level of unknown systems are made at an early design stage. The logical structure of many control systems corresponds ideally with the compositional type of modeling.

Equally important is the separation of the two basically unrelated problem fields in embedded systems:

- correct reaction to events in the environment,
- information transport between this environment and the control system.

To solve these problems separately is not just a matter of reducing a big problem to two smaller ones, but of disentangling two problem complexes that belong to different abstraction levels.

From a complete specification C code is generated. This code can be used for function testing against a numerical model and it is identical to the one installed in the final implementation! Because the model enforces the definition of an explicit control flow, the generated code is determinate and runs efficiently on the target.

5 Hybrid III

The incentive for the project Hybrid III[6] is an urgent need for less polluting private traffic in the industrial capitals of the nineties. The idea consists not so much in developing a completely new kind of vehicle, but mainly in pushing the limits of current drive technology to the extremes[1].

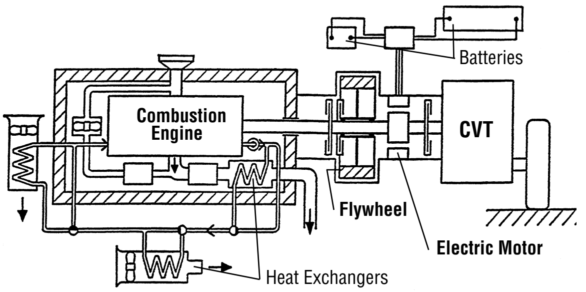

Figure 12: Hybrid III concept

Contrary to current cars where engines operate continuously and within unfavorably broad load ranges, the combustion engine in Hybrid III works at intervals permitting both a restricted RPM-range and optimum load at all times. A flywheel and an electric recovery facility help to make optimum use of the available energy. State of the art exhaust gas treatment and thermal energy management minimize negative impacts on the environment. An electric motor permits vehicle circulation inside pollution-sensitive urban areas. The control system is based on Transputer technology with mezzanine peripheral devices on the VME-bus[1][3]. Test bench measurements confirmed the energy saving potential of some 40% in spite of the more than 300 kg additional weight. The system is currently being installed in a Volkswagen Sharan for road tests.

Hybrid III comprises the clusters:

O Observation
C Combustion
E Electrical
M Mechanical
H Heat Treatment
S System Interface
IO Input & Output.

The hardware of the final installation consists of a 12-slot VME-bus, seven M-Module mezzanine carriers with 28 cards for input and output, one Transputer card with access to the bus acting as master, two TRAM carriers with several Transputers on Transputer Modules, two slave Transputer cards with mezzanine cards for direct I/O, and the broadcast card. Many of these elements are not connected to the bus themselves, but they still share the same shielded card cage and power converters. The interface capacities are:

Input: 80 binary, 64 analog, 8 frequency, 2 phase,

Output: 64 binary, 24 analog.

In view of the great number of peripheral connections and the limited space in a car, elaborate small interface satellites were built. A small, rugged case contains up to 24 miniature cards with 15 different functions like amplifiers, current loop converters and the like. Ten of these satellites are installed close to specific machinery components.

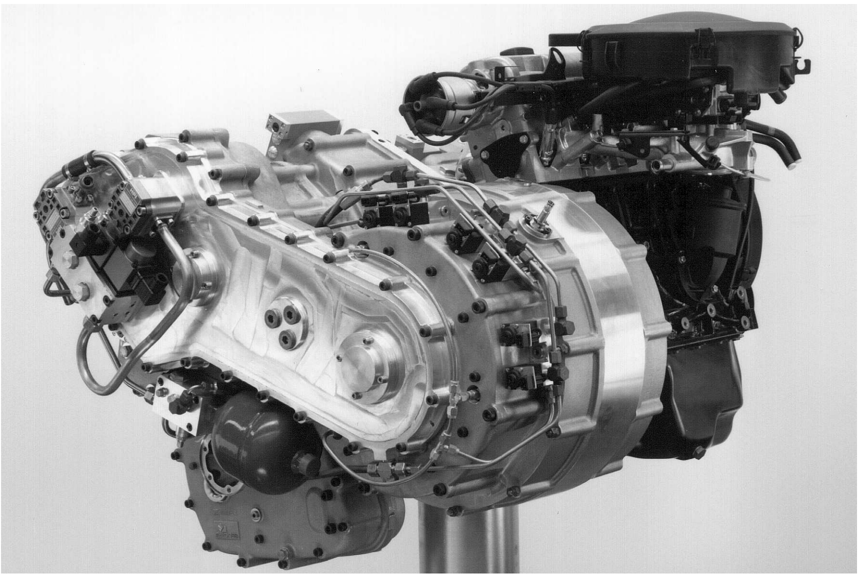

Figure 13: Hybrid III drive train

[1] A joint project of several laboratories at the Swiss Federal Institute of Technology, Volkswagen and other external partners.

6 Summary

The concept of using a small scale parallel system together with a number of advanced tools has been proven in a complex application. The main issues:

- relieve the developers of the machine components of programming tasks,
- support decentralized development work,
- adapt resources according to requirement any time during the development process,
- avoid code alterations when the hardware environment changes,
- cut down size and energy consumption of the control system.

The complete separation of different structural layers is a promising approach to control system technology. Low-level chores may be burdensome, but there is always a solution. Solutions on the reactive layer of a complex system are less obvious, with a mass of possible interactions of many different components. Because the concepts of CIP are based on constructive realities, clear and comprehensible specifications are possible. The background of formal models allows the development of robust and reliable systems. In a complicated case such as Hybrid III it is improbable that there would have been any other possible course. Tools exert a certain discipline and can thus compensate for academic habits. The frequency of fatal errors creeping into the handwritten code made this quite clear.

Sturdiness, small size, compatibility with the harsh environment, the seemingly trivial chores for the installation in a vehicle have been very demanding.

Although Transputers have almost disappeared by now, from a technical point of view the concept remains attractive and especially well suited for complex, embedded systems.

7 References

[1] *The VME-Bus Specification*, VITA 1987

[2] *The Transputer Databook*, inmos 1989

[3] *Mezzanine Bus Mini-Specification*, VITA 1991

[4] H. Fierz, H. Müller et al.: *CIP, Communicating Interacting Processes, A Formal Method for the Development of Reactive Systems,* Poznan 1993

[5] *CIP Tool User Manual 1.2,* CIP System AG, Solothurn 1996

[6] M. K. Eberle, Ed.: Erweiterte Konferenz über hybride Automobilantriebe, IET ETH, Zürich 1996