GENERATION OF EMBEDDED CONTROL SYSTEMS

HO. TRUTMANN

Computer Engineering and Networks Laboratory TIK, Swiss Federal Institute of Technology ETH, CH-8092 Zurich, Switzerland. E-Mail: trutmann@tik.ee.ethz.ch

Abstract. The divergence between the development of the control system and its environment of technical devices in a complex new machine is difficult to handle and usually not addressed by existent development methods. The presented approach considers these questions and at the same time preserves the essential separation of functional and interface domains. Within each domain progress can be made independently using dedicated software tools. Generators provide implementations running with simulation models as well as partial or complete target implementations. The code is statically deterministic and runs in a simple environment with only a few processes. To confirm the real-time behavior of an implementation the generators also provide worst-case traces. Automatic code generation from models and specifications minimizes programming efforts and coding errors. The concept has been proven in a number of industrial and academic projects.

Key Words. Embedded Systems; Specifications; Real-Time Control; Automobile Control

1. INTRODUCTION

Control system design for complex new mechanical systems is plagued by a number of problems inherent in the development process itself. Particularly, the divergence among different versions of test bed implementations can easily become a major source of concern. Along with the obvious real-time issues a well-balanced development environment must address these questions as well.

1.1. Functionality and Interface

Functional and interface aspects should preferably be separated into problem domains, so they can be dealt with by specialists in each field. The boundaries between different domains are semi-rigid, because the number and properties of signals passed to and fro change only occasionally throughout a project. On the other hand, changes within each problem domain are frequent, after all this is where the actual development takes place. Dedicated software tools with appropriate notations are used within each domain.

1.2. Operation Modes

Development of the control system must begin before the machine itself exists, with requirement specifications often remaining vague as to the functionality and the available input and output signals. Individual parts of the new machine may have to be designed separately and put together only after refinement and testing. Work on the components usually takes place in different locations and is carried out by different teams. Developers must rely on simulation models not only because they cannot wait, but also because trials with the actual hardware are too costly and dangerous. Progress is made in loops, so many transitions among the different modes of operation must be expected, where appropriate implementations are often needed at the same time. They must perform equally well with simulation models running on a workstation, with single mechanical components on their own test beds, and with the final assembled installation. Transformations between different modes must be smooth and with as few changes as possible to the control system software.

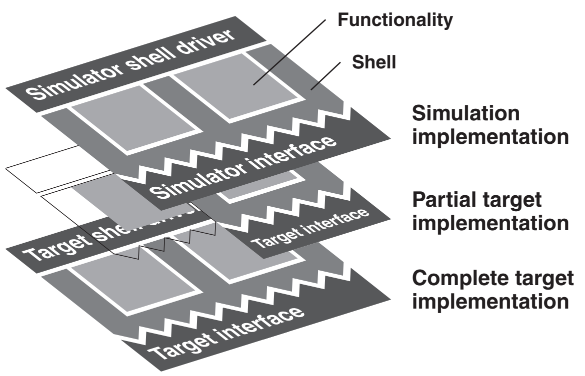

Fig. 1. Problem domains and operation modes

A conflict arises from these two ways of structural division, and specific degrees of freedom for various combinations of problem domains and operation modes can be observed. The driver, shell and interface domains vary with operation modes, whereas the functional domain remains unaltered. Changes within the functional domain, on the other hand, do not affect the driver, shell and interface domains.

1.3. Implementation

The functional model is translated to a true reactive block that must be triggered from the environment. With the exception of this code, which is identical in all operation modes, code for the various operation modes must be generated specifically. This task is greatly simplified by the fact that the shell, interface and driver code for each operation mode is based on the same specification. For target implementations the generators also create worst-case traces of the statically deterministic code; it can then be used to verify the installation's real-time properties.

The concept has been proven in the development of the drive drain for Hybrid III, a Volkswagen Sharan with a novel propulsion system, where partners from academe and industry target the fuel efficiency of private vehicles (Trutmann 1996, 1997). The method has also been used successfully in a number of industrial projects.

2. PROBLEM DOMAINS

Problem domains isolate certain aspects of the entire task to permit separate development within a limited and clearly defined scope.

Functional domain. The functional domain is usually the most complex and at the same time the one where alterations are to be expected until late in the development. It is vital that this domain and its implementation remain unaltered during operation mode transitions. The functionality is modeled in CIP Tool (Fierz *et al.* 1993), an extended state-machine specification environment with a graphical notation. Plain C code can be integrated directly, if appropriate. An example is an algorithm for continuous control, developed separately and used as complete building blocks. The tool generates code automatically from complete specifications.

Shell domain. The shell domain connects the functional domain with the surrounding interface and driver. Communication issues between different processor entities are also handled by this domain. The shell is specified within CIP Tool at the outset of a project and is assumed to be more inert than both the functional and the interface parts. It can be locked for extended periods and thus incorporates one of the semi-rigid joints vital in isolating the problem domains.

Driver and interface domains. These domains connect the shell to the environment. System input is handled in a round-robin fashion for an arbitrary number of signals in different frequency classes, including conversion, verifying and filtering functions. Events used to drive the functional reactive block are extracted from the input data according to specified priorities. System output messages are either converted to the proper hardware formats and forwarded to the output devices or communicated to its receivers. While these domains have simple regular structures, they differ considerably for each of the various operation modes. Specifications for all cases are made in textual form, and the variants generated mechanically. Because they change rarely, they constitute another semi-rigid joint.

3. STRUCTURE

3.1. Logical Structure

The logical structure of a control system must correspond with respective elements in the environment of the final entire installation. It thus comprises a number of singular, loosely cooperating blocks in just the same way as the environment. These functional groups form widely autonomous subsystems that are capable of executing complex operations all by themselves. It is only possible to develop and test subsystems separately when the mechanical components belonging together are united within the control system as well.

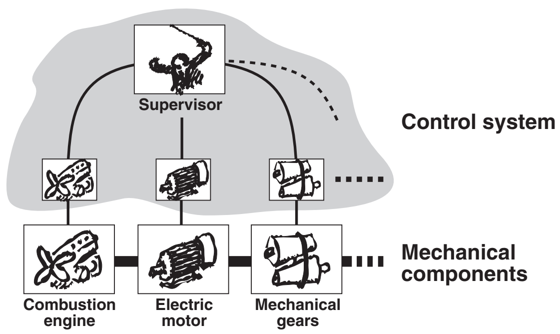

Fig. 2. Logical structure (example)

Often it is convenient to add additional blocks with higher level functionality. They supervise and coordinate individual subsystems and also permit hierarchical system decomposition, thus restraining any direct communication of leaf elements with each other. This representation abstracts completely from implementation issues and lower level functionality and thus constitutes an ideal setting for specialized engineer groups who work within the functional domain on a part of the machine and corresponding control system. Another advantage of the star topology is that it is perfectly possible to set up a partial system without any interface changes to the group in question and for any specific test bed installation.

3.2. Implementation Structure

Depending on the operation mode and the desired real-time performance within a certain architecture, the implementation structure can be varied at will (Fig. 3.). The decision to concentrate all group implementations within a single processing entity, or to distribute them among separate processors is completely detached from the given logical structure. A limitation is that clusters (see section 4.1.) are the smallest assignable entities.

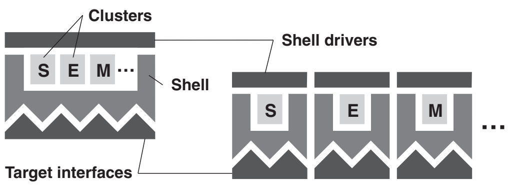

Fig. 3. Combined and distributed implementations

Because functional groups execute their local functions autonomously, fast operations remain within the scope of a single processor. Functions involving several groups typically present slower activities with reference to the whole installation; therefore the added communication penalty in the case of distributed implementations can be tolerated. Hardware resources may be attuned exactly to the required real-time properties using the statically deterministic worst-case traces.

The logical structure guarantees interface consistency of any partial with full implementations. In the special case of simulation the entire functional code is always united in a single implementation. In both the partial and complete modes of target implementations any number of clusters may be assigned to separate processor nodes. Furthermore, clusters can also share input and output devices to achieve cost and space efficient use of peripheral resources.

4. METHODS AND TOOLS

The use of a tool for modeling reactive behavior within a clearly defined functional problem domain is crucial. The notation is adapted to modeling system behavior and complex functional interactions; at the same time it eliminates many unwanted constructs of a general purpose programming language, e.g., recursion and non-determinism. The resulting implementations are free of dynamic language elements, and because the code is serialized and runs with a minimum number of tasks, costly operating system facilities are unnecessary.

Abstract specifications of low-level functions are stated in a textual notation. Input, output, communication and event extraction code is generated, adding the benefits of access control and hidden details. From a developer's view system design, implementation and testing thus take place in an integrated environment.

Any developer group begins with the definition of communication, input and output requirements for their particular case. This information defines the properties of the semi-rigid joints used to isolate separate problem domains. Number and kind of sensors and actuators vary greatly depending on the installation, just as do their associated data rate, resolution and filter parameters. As soon as the joints are locked, work within different domains can progress independently.

Intricate simulation models form the basis for most of the work on complex control systems. Models allow developers to experiment with critical features in safety and to catch up with the constant progress the rest of the machine makes. The delay from assembly of the entire machine to the first test runs thus is shortened considerably.

4.1. Communicating Interacting Processes CIP

Communicating Interacting Processes CIP is a method and corresponding tool for developing the functionality of real-time and embedded systems (CIP Tool 1995-98). It describes control systems in a single- or multiprocessor environment with a notation of synchronous processes and asynchronous clusters, together with the necessary communication mechanisms (Fig. 4.). Although the model is event-based, quasi-continuous processes, such as control algorithms, integrate well.

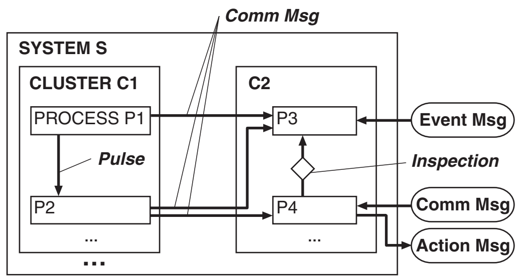

Fig. 4. CIP system structure

Event or communication messages from the environment cause a transition in the receiving process, which can activate other processes for his part with synchronous pulses. Explicitly specified static cascades propagate these pulses from one process to the other. Transitions can generate action or communication messages to influence the environment. In addition to pulses passing from one process to another, passive state inspection of other processes can be used to eliminate non-determinisms in transition graphs. 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. A chain reaction initiated by a communication or event message cannot be interrupted and defines a single state transition of the cluster; the tool prevents it from being cyclic. C code for added functionality may be placed in any transition, a convenient way to incorporate algorithms obtained from other sources.

CIP models of reactive systems evolve constructively as a composition of extended finite state machines. Formally the cluster model is the product of these state machines, forming a single machine in a multidimensional state space. Clusters are built with elementary reactive blocks, with model processes that either mirror parts of the outer system reality or functional processes that deal with the cooperative aspects of the components. The internal interaction of all these processes defines the comportment of the reactive cluster model in its entirety. Additionally, different modes permit changes in the dynamic behavior of single processes. The currently active mode depends on the state of a master process. Modes therefore add the concept of behavior hierarchies to the simple behavior abstraction (Fig. 5.).

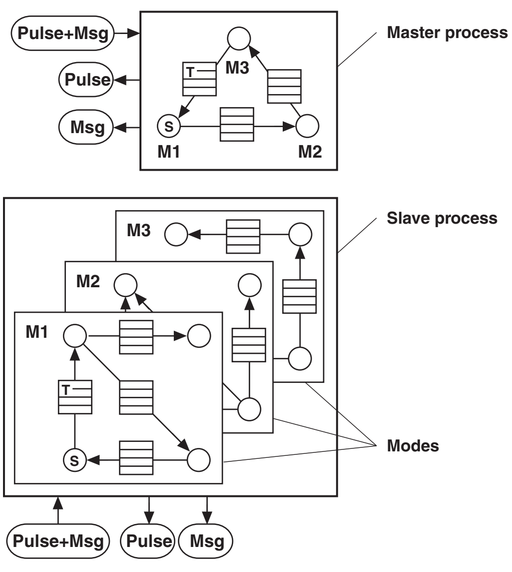

Fig. 5. CIP state machine hierarchy

From a complete specification C code is generated. The code used for testing against a numerical model 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 deterministic. It can be used for performance analysis and runs efficiently on the target.

4.2. Interface Generators

Considering all the different peripheral hardware devices and their access methods, interfacing the functional code to a specific environment can be as challenging as the functionality itself. The absence of structural difficulties encourages simple, linear solutions for the three basic tasks: data input, data output and communication. For practically all devices setup and data transfer can be handled with at most two separate accesses, without using interrupts (see section 5.). The execution and required delay times of the low-level access functions written in C must be profiled beforehand to enable the code generators' static scheduling. In the special case of simulation, where the interface is much simpler, implementations are based on a subset of the specifications. Because the domains' structures are not affected, all variants can be generated by tools, in this case they are operating from textual specifications.

Input. Inputs occur at fixed intervals, with different frequency classes for the individual signals (e.g. F_{high} = 1 kHz, F_{med} = 100 Hz, F_{low} = 10 Hz side by side). To balance the system work load, carefully chosen groups of input signals are read at a given point of time. The implementation essentially consists of a switch with F_{high}/F_{low} cases and an almost identical workload for all cases. Conversion, verifying and filtering functions complete the access procedures, so only reliable and standardized data are present throughout the entire control system.

Event extraction. Events are required to drive the functional reactive block; they must be extracted from the newest input data. While discrete signals are simply checked for changes, the situation for events buried within continuous signals is less evident. The data supplied for use by an extractor algorithm needs a hysteresis for proper switching (Fig. 6.).

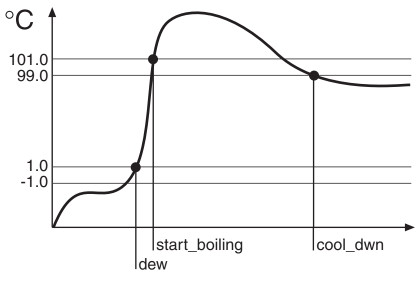

Fig. 6. Event extraction from a continuous signal

The event extractor maintains the correct ordering of subsequent events. Furthermore, it does not only regulate the amount of events the system handles, it is also capable of switching between different priority schemes at run-time. The currently active scheme is chosen from within the functional model, where all information regarding system status and desired system behavior is available.

Output. System output may need similar multi-stage access methods, but is usually simpler to implement. After conversion to the required hardware format the pending accesses are buffered and eventually forwarded to the output devices.

All the properties of the input and output system are stated in a simple textual notation and then used to generate implementations for the various operation modes. This code is also deterministic and complements the functional code in the performance analysis of the entire installation.

5. IMPLEMENTATION

5.1. Simulation Implementation

A simulation implementation[1] maps the specified input and output signals to the simulation environment's data structures. It maintains the desired timing relations and uses the same algorithm for event extraction as in the target implementations.

5.2. Target Implementations

In a target implementation the desired cluster partition and interconnections must be adapted to the features existent in a hardware environment. The key issue of the chosen approach is to avoid scheduling a great number of processes. In a single-processor environment three processes are sufficient (apart from eventual interrupt service routines, if they cannot be avoided). This limitation is possible because the code is statically deterministic and therefore open to performance analysis before installation. The possibility for adapting processing power to changing requirements dynamically is retained, with the additional bonus that these changes remain under control of the functional code. In case of event avalanches, when the number of simultaneous events exceeds the specification, system reactions may be late, but this never leads to a complete failure. If the hardware complies with the required peak processing power for the hard real-time chores, additional processes can exploit the remaining capacity without compromising performance.

Processors. Implementations with parallel hardware offer a number of advantages already during the development process: structural flexibility without adverse effects on performance, as well as adaptable processing power to changing needs with additional local processor nodes. The possibility of dividing a complex task into individual processes is ideally tailored to many system demands; each task block is controlled by a group of processes on an independent processor which communicates with his partners. Processors that support communication and process scheduling directly are ideal, because the application can then be built easily on top of the hardware without the need of an operating system. Single-processor solutions, on the other hand, are more convenient for sharing common resources among different clusters. It is beneficial that this decision need not be made irrevocably at an early stage of a project; the code generators facilitate architectural changes at any time.

Shared I/O. In a distributed implementation it is costly and inefficient to fit every cluster with its own data input and output mechanism. Supplementary clusters, hidden from the logical structure, can handle peripheral data input and output for the entire system. Generally multi-channel devices can be used more efficiently by dividing the available channels among various clusters (Fig. 7.).

A data input cycle begins with acquiring a group of input data. 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 a broadcast mechanism. Sometime later, after the working clusters have finished their processing and calculated new values, output data from each cluster, converted to the correct hardware format is 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.

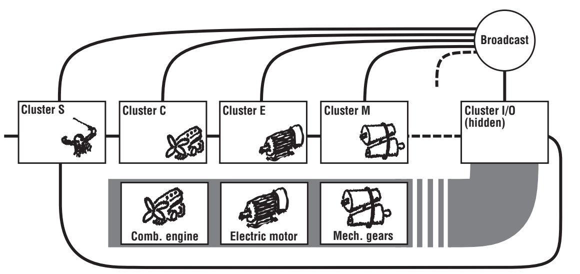

Fig. 7. Distributed implementation using shared I/O

Communication. The logical star topology forms the base of all communication. In a distributed system a ring with appropriate protocols for point-to-point and broadcast connections can behave identically. Communicator processes on each cluster handle ring traffic from and to an element and forward other messages. The same ring mechanism is also used for output data.

The final implementations for all the various operation modes result from a combination of different development strands, combining expert knowledge of the many specialists involved (Fig. 8.):

[1] currently for Matlab/Simulink

1) the functionality is modeled in CIP Tool, a specification environment with a graphical notation, with C code generated automatically,
2) algorithms for continuous control are developed separately with appropriate tools and integrated as complete building blocks,
3) input and output requirements, filtering, event extraction, priorities, etc. are stated in textual specifications. Tailored to the operation mode, C code is generated from these specifications,
4) the basic routines for all low-level mechanisms such as input, output and communication are written in C. Performance measurement data must be supplied for the interface code generators.

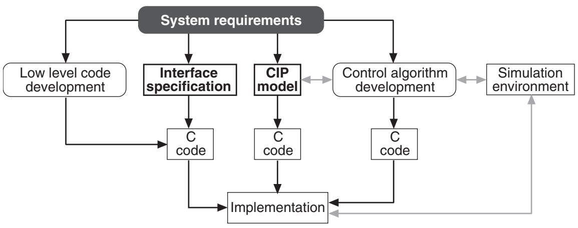

Fig. 8. Development strands

The separation of system functionality (functional domain 1-2) from low-level chores (shell and interface domains, performance analysis 3-4) 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.

6. SUMMARY

Combining problem domain separation with different operation modes has been proven in complex applications. The main issues are:

- relieve the developers of the functional components of programming tasks,
- support decentralized development work,
- adapt resources according to emerging requirements any time during the development process,
- automate code alterations when the hardware environment changes,
- test the proposed solutions against numerical models before attaching the control system to the mechanical components,
- simplify the target implementations: no complicated scheduling, no interrupts, no fancy operating system,
- analyze the performance of the statically deterministic code exhaustively to assure the desired real-time properties.

The reactive behavior of a complex system is less than obvious, with a mass of possible interactions of many different components. Because the concepts of CIP are based on constructive realities, clear and comprehensible models are possible. A formal background allows the development of robust and reliable systems. 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.

Specifying low-level chores combined with code generation minimizes programming efforts and coding errors. Automatic generation of the program code from specifications becomes a necessity whenever a large amount of signals and complicated restraints on the hardware accesses are involved.

The complete separation of different structural layers in the form of problem domains remains the crucial point in the development of embedded control systems; correct reaction to events in the environment on the one hand, information transport between this environment and the control system on the other hand. 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. The different operation modes necessary during development tend to disrupt this separation.

Efficient tool support eases the task of switching operation modes without disturbing the problem domains; it greatly improves the project engineers' performance and adds invaluable benefits in overall system security.

7. REFERENCES

CIP Tool - User Manual (1995-98). *CIP System AG.* Solothurn.

Fierz, H., H. Mueller and S. Netos (1993). CIP, Communicating Interacting Processes. *A Formal Method for the Development of Reactive Systems.* Poznan.

Trutmann, HO. (1996). In M. K. Eberle (Ed.). Verteilt und doch zusammen - zum Steuersystem von Hybrid III. *Erweiterte Konferenz ueber hybride Automobilantriebe, IET ETH,* Zuerich.

Trutmann, HO. (1997). Flexible Small Scale Parallel System in the Development of a Hybrid Vehicle. *Parallel and Distributed Processing Techniques and Applications PDPTA'97.* Las Vegas.