The CIP Method: Component- and Model-Based Construction of Embedded Systems

Hugo Fierz

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

Abstract. CIP is a model-based software development method for embedded systems. The problem of constructing an embedded system is decomposed into a *functional* and a *connection* problem. The *functional* problem is solved by constructing a formal reactive behavioural model. A CIP model consists of concurrent clusters of synchronously cooperating extended state machines. The state machines of a cluster interact synchronously by multi-cast events. State machines of different clusters can communicate through asynchronous channels. The construction of CIP models is supported by the CIP Tool, a graphical modelling framework with code generators that transform CIP models into concurrently executable CIP components. The *connection* problem consists of connecting generated CIP components to the real environment. This problem is solved by means of techniques and tools adapted to the technology of the interface devices. Construction of a CIP model starts from the behaviour of the processes of the real environment, leading in constructive steps to an operational specification of the system behaviour. This approach allows stable interfaces of CIP components to be specified at an early stage, thus supporting concurrent development of their connection to the environment.

1. Introduction

The CIP method (Communicating Interacting Processes) presented in this paper is a formal software development method for embedded systems. By 'embedded system' we mean any computer system used to control a technical environment. Examples include highly automated devices, industrial robots and computer controlled production processes.

CIP specifications are constructed with the CIP Tool[1] [1], a powerful modelling framework with verification functions and code generators that transform CIP models

[1] CIP Tool® is a registered trademark. All graphic figures of model parts presented in this paper have been generated from models created with CIP Tool®. Associated textual descriptions such as condition definitions or condition allocations correspond to model reports produced automatically.

submitted to ESEC 99

into executable software components. The method and its tool has been used in many real projects. The benefit of a rigorous problem-oriented approach is an important improvement of software quality, reflected by understandable system models, robust and reliable software products and a considerable reduction of maintenance costs.

The starting point in the design of CIP was the JSD method (Jackson System Development) [2], adopting the real world oriented modelling paradigm of this approach. JSD treats dynamic information problems by means of concurrent sequential processes, simulating a part of the real world and producing requested information about it. CIP differs from JSD mainly in its modelling framework which is based on synchronously cooperating extended state machines rather than on concurrent processes described by structograms.

The CIP method is based on the following development concepts:

Problem Decomposition. The usual purpose of behavioural models in embedded system development is to specify the functional system behaviour in subject-matter terms. Such models, independent of technical interface concerns, are often called essential models [3]. The connection of an implemented essential model to the real environment represents a problem in its own right, demanding tools and techniques adapted to the technology of the interface devices.

Model-Based Operational Specification. The functional behaviour of a CIP system is specified by an operational model of cooperating extended state machines. 'Operational' means that the model is formally executable [4]. CIP combines synchronous and asynchronous cooperation of system parts within the same model. Synchronous cooperation, well known from real-time description techniques like Statecharts [5], ESTEREL [6] or Lustre [7], is needed to model synchronous propagation of internal interactions. Asynchronous cooperation on the other hand, supported by parallel modelling languages like SDL [8], JSD [2] or ROOM [9], is necessary to express concurrency.

Component-Based Construction by White Box Composition. CIP models are developed with the CIP Tool [1], a framework of graphic and text editors supporting full coherence among various architectural and behavioural views. Models are constructed by creating, composing and linking model components such as processes, channels, messages, states and operations. Modelling by compositional construction perfectly supports the problem-oriented construction process of the method, providing more flexibility and intuition than language-based specification techniques.

Component-Based Implementation by Black Box Composition. CIP models are transformed automatically into executable software components which are integrated on one hand with components connecting to the environment, on the other hand with system parts like technical data processing units or extensive algorithmic functions. The main goal of software component technology is usually to construct systems by means of reusable building blocks. Although reuse of embedded components is often not possible because of the specific behaviour of the particular environment, component composition is still of great value. Concurrent development and flexible system integration is easier when system parts are constructed as software components with stable interfaces.

Environment-Oriented Development. The development of a functional behavioural model must start with a rigorous definition of the model boundary. The widely used context schema of SDRS [3] for example models the boundary by means of event and data flows to and from the system. Although such boundary models allow the development of the behavioural model to be based on a well defined set of external interaction points, they fail to express any behavioural relationships with the environment. CIP starts the development by defining the set of valid interaction sequences by means of a behavioural context model. This approach supports the construction of robust and dependable systems because it incorporates a formal model of the environment behaviour.

The main part of the paper starts in section 2 by describing how CIP tackles the embedded system problem. Section 3 explains the architectural and behavioural constructs used to build CIP models. Section 4 presents the environment-oriented development process of the CIP method, illustrated by a simple but complete example of a CIP model construction. Section 5 finally describes how generated software components are connected to the environment.

2. CIP Application Area: Control Problems

An embedded system is a computer system which controls and monitors a number of external processes. The behaviour of the individual processes is partly autonomous and partly reactive. In the case of physical processes the behaviour can be deduced from physical properties. More complex processes are often already controlled by local microprocessors. An operator driven man-machine interface is another source of asynchronous influences on the system.

Two main problem are encountered when an embedded system is developed. The first problem concerns the functionality of the system: to bring about the required behaviour the embedded system to be constructed must react in a specific way to subject-matter phenomena of the environment. The second problem concerns the connection of environment and embedded system: the subject-matter phenomena related in the functional problem solution must be detected and produced by monitoring and controlling specific interface devices like sensors and actuators.

As a simple example we take an opening door where the door motor has to be turned off when the door is fully open. The function of the system is simply to produce the *MotorOff* action when the *Opened* event occurs. However, neither the event *Opened* nor the action *MotorOff* are directly shared with the embedded system. Instead the event *Opened* must be detected by means of a position sensor which is connected to the embedded system by a shared binary variable; and the action *MotorOff* must be produced by setting a binary variable of the motor actuator appropriately.

The CIP method is based on a complete separation of the functional and the connection problem. The functional problem is solved independently of the interface devices by constructing a rigorous behavioural model. The CIP model is constructed graphically with the CIP Tool and transformed automatically into concurrently executable CIP components.

The construction of a CIP model is based on a *virtual connection* between external processes and CIP components. The virtual interface of the environment consists of collections of events and actions designating instantaneous subject-matter phenomena occurring in the environment. *Events* are phenomena initiated by the environment. Process events, often called discrete events, are caused by the autonomous dynamics of external processes. Continuous behaviour of external processes is captured by periodic temporal events with associated state values (sampling). *Actions* are environment phenomena caused by reactions of the embedded system. The virtual connection transmits a corresponding message whenever an event has occurred or an action must be produced.

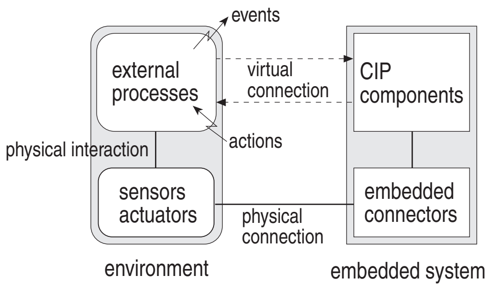


Fig. 1. Conceptual embedded system architecture

The working connection between external processes and CIP components consists of sensors and actuators connected to system modules called *embedded connectors*. Embedded connectors detect events by monitoring sensor phenomena, and transmit corresponding event messages to the CIP components. They receive action messages from CIP components, and initiate appropriate actuator phenomena to produce the corresponding actions in the environment.

In the door example, the reaction "*Opened causes MotorOff*" would be produced by a CIP component while the monitoring and setting of the interface variables is done by a separately constructed embedded connector.

The complete separation of the functional and the connection problem considerably benefits the development process since the two problems can be solved independently. This is not just a question of reducing a large problem to two smaller ones, but of disentangling two problem complexes belonging to different abstraction levels.

An important benefit of this problem separation appears for example when control functions have to be validated. Because the developed software can rarely be tested directly on the target system, it is necessary to use simulation models and specific test beds. The CIP approach allows a functional solution to be partitioned in various ways and corresponding software components to be generated that can be easily embedded within various test environments.

The proposed problem decomposition has been elaborated by means of the notion of problem frames recently introduced by Jackson [10]. The approach allowed us to understand more deeply the generic structure of the embedded system problem and its relation to the development process. Explicit discussion of the use of problem frames for embedded systems is beyond the scope of this paper and will be presented in another publication.

3. CIP Models

The CIP meta-model has been defined on the basis of a compositional mathematical formalism [11] developed for this purpose. The modelling tool has been designed as a component framework based on an object model implementing the CIP meta-model. The benefit is simple compositional models which can be transformed easily into executable code. Current research and development extends the modelling framework by tools allowing CIP models to be checked against independently defined behavioural properties.

CIP models are constructed graphically by means of architectural composition, well known as a basic paradigm of architecture description languages [12]: communication and interaction among state machines is specified by interconnecting these components by first-class connectors. A drawback of many synchronous real-time languages is that interconnections are described only implicitly, thus proliferating complex interaction dependencies. In addition to the architectural configuration, CIP requires the control flow among synchronous components to be explicitly modelled, thus preventing cyclic and conflicting chain reactions. Furthermore, behavioural structuring is supported by the novel model of *master-slave hierarchies* [13].

Clusters and Processes. A CIP model is composed of a set of asynchronously cooperating *clusters*, each consisting of a number of synchronously cooperating state machines termed *processes*. Formally a cluster represents a state machine with a multi-dimensional state space: the state of a cluster is defined by the tuple of its process states. Although clusters as well as processes represent parallel behavioural entities, their composition semantics is essentially different: clusters model concurrent blocks of a system, while processes represent orthogonal components of a cluster. Hierarchical composition structures based on a simple "part of" hierarchy relation can of course be introduced at both levels, resulting in structures of nested subsystems and nested subclusters.

Communication – Asynchronous Transmission of Messages. Processes of different clusters communicate asynchronously with each other and with the environment by means of *channels*. Communication is specified by a graphical net model (Fig. 2) in which channels are attached to process ports. Source and sink channels model the virtual connection to the environment while internal channels are part of the CIP model.

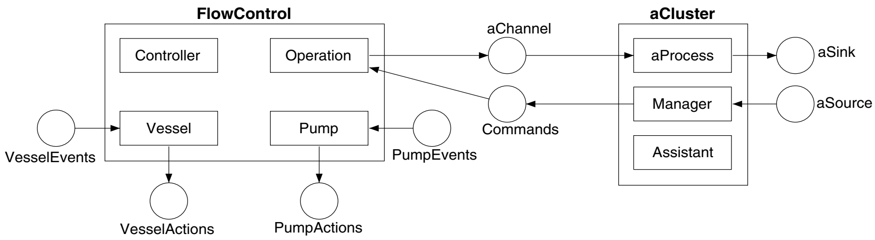


Fig. 2. Communication net of a CIP model

Channels model an active communication medium which retains the sequential order of transmitted messages. Asynchronous communication in a CIP model means that the write and the read action of a message transmission takes place in different cluster transitions. Processes represent receptive behavioural entities which must accept delivered messages at any time. The necessary buffer size of a channel is determined by the performance of the implemented system and is not specified within the CIP model.

Interaction – Synchronous Pulse Transmission. The processes of a cluster interact synchronously by means of multi-cast *pulses*. Pulses represent internally transmitted events. The directed connectors of the interaction net (Fig. 3) define the pulse flow structure of a cluster. Every connector has an associated partial function termed *pulse translation* which relates outpulses of the sender to inpulses of the receiver process.

A cluster is always activated by a channel message which leads to a state transition of the receiving process. By emitting a pulse, the receiving process can activate further processes of the cluster, which can in turn activate other processes by pulses. The chain reaction resulting from pulse transmission is not interruptible and defines a single state transition of the entire cluster. Activated processes can also write messages to their output channels.

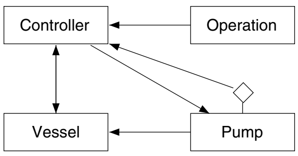


Fig. 3. Interaction net

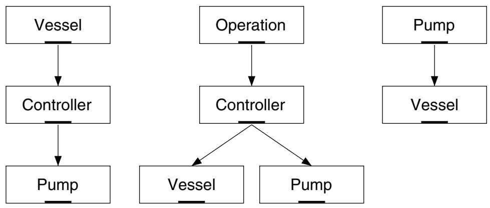


Fig. 4. Cascades of a cluster

The structure of the interaction net does not sufficiently restrict the potential pulse transmission chains, as conflicting process activations and cyclic transmission paths are in general possible. To ensure deterministic and bounded propagation of interaction, for each process with channel input the possible control flow of chain reactions is defined by means of a *cascade* (Fig. 4). A cascade is a sequential activation tree compatible with the interaction net structure of the cluster. The pulse interaction defined in the processes can be checked automatically against the specified cascades. A cascade is activated by a channel input for its topmost process. The execution order of a cascade is defined as tree traversal from left to right. To support control flow structuring cascade branches can be refined into alternative branches depending on the pulses transmitted.

State Inspection - Static Context Dependency within a Cluster. The conditions of a state transition structure of a process are allowed to depend on the states and variables of other processes of the same cluster. Read access to the data of a process is called *state inspection* and takes place as in object oriented models via access functions termed *inquiries*. By contrast to pulse transmission, where both transmitter and receiver are activated, in the case of state inspection, only the inspecting process is active. State inspection gives rise to additional dependencies between processes which are declared graphically as rhombic connectors in the interaction net (Fig. 3). The arrows denote the data flow direction.

Processes – Extended Finite State Machines. Processes are modelled as extended finite state machines. By means of state transition structures and operations executed within transitions, functionality can be specified on two different levels of abstraction.

The communication interface of a process is defined by one or more *inports* and *outports*. A port is specified by the set of *messages* to be received or sent. Each inport and outport is connected in the communication net to an incoming or outgoing channel respectively. Interaction inputs and outputs on the other hand are defined by two distinct sets of inpulses and outpulses.

PROCESS Door
INPORT DoorEvents
  MESSAGES Closed, Opened
OUTPORT DoorActions
  MESSAGES MotClose, MotOpen, MotOff
INPULSES close, open
OUTPULSES clsAck, opnAck

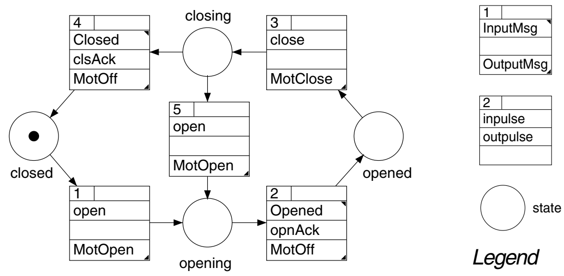
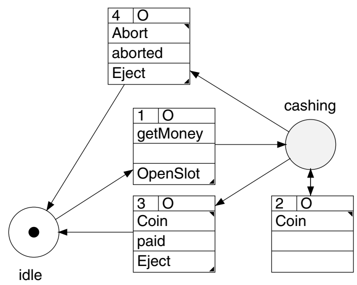


Fig. 5. Pure finite state machine

The transition structure depicted in figure 5 specifies the behaviour of the process *Door*. Process states are represented by circles, transitions by labelled transition boxes. The input messages *Opened* and *Closed* or the inpulses *open* and *close* of the process can activate a state transition. An input message for which the current state has no outgoing transition causes a context error. An inpulse for which there is no transition, on the other hand, is ignored. In each transition an outpulse and a message to each outport can be emitted.

To support data processing and algorithmic concerns CIP processes are specified as extended state machines. The extension consists of static process variables, data types for messages and pulses, operations and conditions.

PROCESS Cashier
VARIABLES amount: int, price: int
INPORT EventPort
  MESSAGES Coin; int, Abort
OUTPORT ActionPort
  MESSAGES Eject: int, OpenSlot
INPULSES getMoney: int
OUTPULSES aborted, paid: int
OPERATIONS
init `{SELF.amount = 0; SELF.price = IN;}`
incr `{SELF.amount = SELF.amount + IN;}`
...
CONDITIONS
`notEnough (SELF.amount < SELF.price)`



CONDITION ALLOCATION
2 notEnough, 3 ELSE_

OPERATION ALLOCATION
1 init, 2 incr, 3 change, report, 4 ...

Fig. 6. Extended finite state machine

The *Coin* message of the process *Cashier* (Fig. 6) for instance carries an integer value representing the value of the inserted coin. Operations allocated to transitions are used to update the inserted amount and to calculate the change. When the process is in the *cashing* state - shown in grey - there are two potential transitions for the input message *Coin*. Associated conditions render the process behaviour deterministic. Such conditions can depend on the input data and the values of the local process variables, but also, by state inspection, on the states and variables of other processes of the same cluster.

Variables, data types, operations and conditions are formulated in the programming language of the generated code. From the high level modelling point of view these constructs represent primitives which add computational power to the pure models. The specified code constructs are incorporated inline in the generated code. From a theoretical point of view it would be more elegant to use a functional specification language, but in practice the value of this pragmatic approach based on the implementation language has been clearly confirmed: it permits easy use of functions, data types and object classes from existing libraries.

The *Door* and *Cashier* processes presented react to event messages indicating discrete state changes of an external process. Control of continuous processes on the other hand is based on periodic sampling of the continuous process states. Figure 7 shows the transition structure of a process regulating the temperature of a liquid by means of a heater with continuously variable heating power. The process reacts to the periodically occurring *Sample* message giving the sampled liquid temperature. Control algorithms are allocated to transitions. Feedback control is performed by means of the data carried by the *Heat* output message.

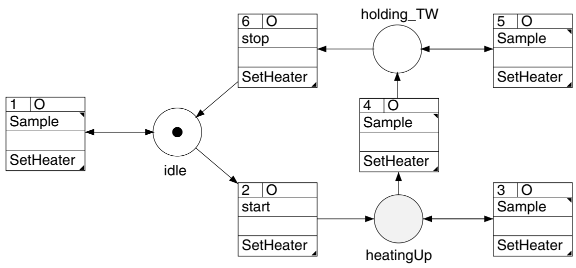


Fig. 7. Time driven state machine

Master-Slave Hierarchies – Behavioural Structuring. The state transition structure of a CIP process specifies how the process must react to inputs by state changes and generated outputs. Often such a behaviour becomes quite complex due to inputs which must influence the full future behaviour of the process. An alarm message, for instance, must lead to behaviour different from the normal case until the alarm is reset. The resulting transition structure will then represent a kind of superposition of the structures for the normal and the alarm case.

To disentangle such implicit superpositions the full behaviour of a process is modelled by a number of alternative *modes*. Each mode is specified graphically by a state transition diagram, based on the common sets of states, ports and pulses of the process.

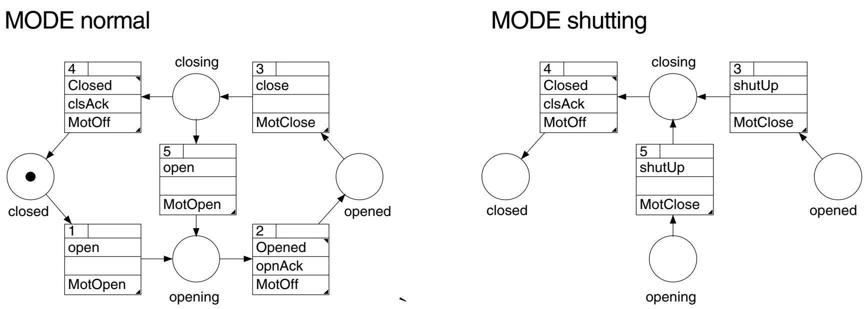


Fig. 8. Two modes of the Door process

Figure 8 shows an elaboration of the Door process (Fig 5) by an additional *shutting* mode. This mode describes an alternative behaviour of the process, usable when an alarm or error condition occurs.

The mode changes of a process can be induced by one or more processes designated as master. The behavioural semantics of a master-slave relationship is defined as follows:

The active mode of the slave is determined by the current states of the masters.

Thus a mode change of a slave can occur whenever one of its masters changes its state. Even when the Door process of figure 8 is in the closed state in the normal mode, a master can induce a switch to the shutting mode. The effect is simply that the door will remain closed when an open pulse is sent to the process. If for functional reasons a mode change should be prevented in certain states, it must be prevented by means of explicitly modelled interaction between slave and master.

A slave can itself initiate a mode change by sending a pulse to one of its masters. This pattern is used typically when an error is recognised by a slave and its master must then be triggered to induce a change to an error mode.

It is important to understand that a change of mode does not affect the current state of a slave, which can change only when a transition in the active mode is triggered by an input. The rule reflects the fact that a mode change does not alter the history of basic interactions. A striking example is the interface process Conveyor of the case study (section 4) whose states mirror the states of the real conveyor.

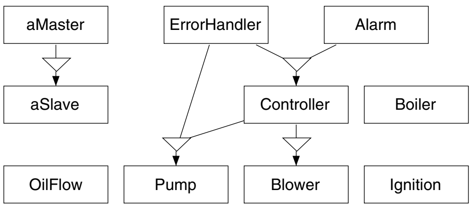


Fig. 9. Master-slave hierarchy graph

The master-slave hierarchy relation of a cluster is specified by a master-slave graph (Fig. 9). Master-slave connections are represented by triangles which are connected at the bottom angle to a slave and at the top side to one or more masters. The graph is restricted to be acyclic in order to define a hierarchical structure. The levels indicated in the graph of figure 9 have no formal meaning. Levelling is merely used informally to group processes interacting on a common level of abstraction. Usually not all processes of a cluster are involved in the hierarchical structure.

The associations of master states and slave modes are specified by corresponding *mode setting* tables. They define a total function from the Cartesian product of master states to the modes of the slave.

Process Arrays – Static Replication of Processes. Replicated processes are modelled as multidimensional process arrays. The multiplicities of the singular array dimensions are defined by abstract index types. Using common index types for different process arrays allows modelling of finite relations among process arrays, usually expressed by means of entity-relationship diagrams.

4. Domain-oriented Development

An operational behavioural model provides a well defined level of abstraction. In addition to these "guard rails", the CIP method adopts the concept of environment-oriented behavioural modelling from the JSD method. This concept bases the development of control systems on a realised model of the environment inside the system in order to capture the essential behaviour of the processes to be controlled. CIP models are therefore constructed in a sequence of three steps:

1. Specification of the virtual real world interface
2. Establishing a behavioural context model
3. Construction of control functions

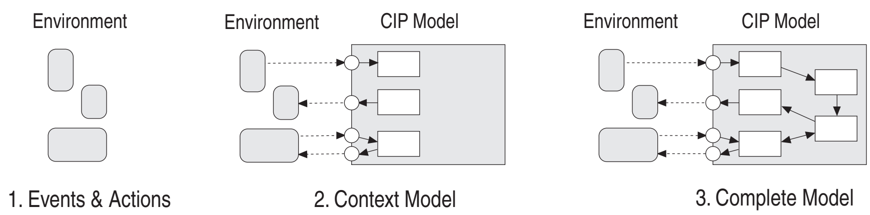


Fig. 10. Results of the three development steps

The virtual real word interface is specified by collections of events and actions used to bring about the required functional behaviour. The context model is the first part of the CIP model to be constructed. It consists of source and sink channels attached to CIP processes that consume event messages and produce action messages. The transition structures of these processes represent protocols of the virtual communication with the environment. In the third step the CIP model is completed by creating function processes which interact and communicate with the processes of the established context model.

The three developent steps are explained in this section by means of a simple case study. Following the case study, the essential step of defining events and actions is then treated in more detail at the end of the section.

4.1 Problem Statement of the TCS Case Study

The TCS (TransportControlSystem) example illustrates how a CIP model is developed and how a simple master-slave hierarchy works. The resulting cluster represents an executable solution of the stated problem. The algorithmic requirements are trivial, so the model consists of pure state machines only.

Plant description. The plant to be controlled comprises a conveyor moving objects in one direction, a scanner serving to identify loaded objects and a switch allowing the operator to enable and disable object scanning. A digital loading sensor at the front end of the conveyor detects loaded objects.

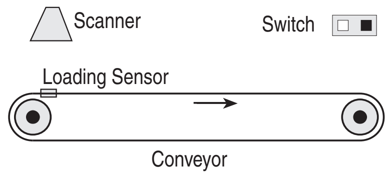


Fig. 11. TCS Plant

Requirements. The systems starts its function when the first object is loaded. When the conveyor is not loaded for 30 seconds, its motor is to be turned off. Two modes of operation are required. If the switch is set off, the conveyor moves continuously and the scanner is not activated. If the switch is set on, the conveyor must stop when an object is loaded, the scanner is activated, and the conveyor starts moving again when the scanner indicates that scanning is complete.

4.2 Virtual Real World Interface: Events and Actions

In the first step a virtual interface to the environment is specified by identifying events to be detected and actions to be produced by the embedded system. Events and actions designate instantaneously occurring subject-matter phenomena of the environment.

TCS – Virtual Real World Interface

Event List

On / Off	the switch is set on / off
Load / Free	an object is loaded / is moved away from the loading place
Scanned	scanning is completed

Action List

MotOn / MotOff	turning the conveyor motor on / off
Scan	activating the scanner

Fig. 12. Event and action lists

4.3 Behavioural Context Model: Channels and Interface Processes

The purpose of the context model is to establish the interface processes of the CIP model and to connect them virtually to the environment. The interface processes are deduced from descriptions and process models of the environment. They receive event messages and produce action messages through appropriately specified source and sink channels. The state transition structures of the interface processes describe the valid sequences of received event and produced action messages. Thus the context model formally describes the behaviour of the individual external processes, seen from the CIP model.

The channels of the context model represent a virtual connection to the environment. The event and action messages of these channels must correspond to the events and actions of the virtual real world interface. These channels are also used as the interface model for the CIP components to be generated later on as described in section 5.

TCS – Context Model

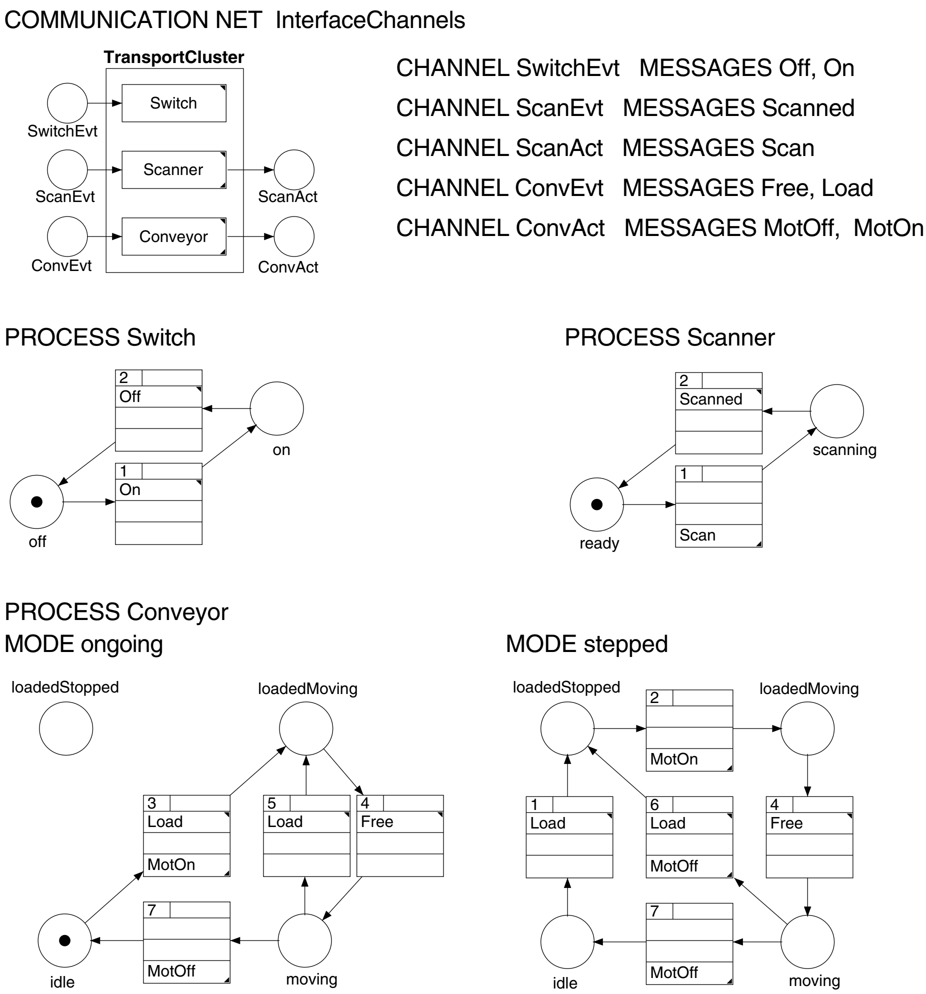


Fig. 13. Context model specification

The incomplete transition structures of the preliminary independent interface processes must be completed in the control function step. Modelling interface processes means understanding the behaviour of the external processes; but it also means anticipating the way they will be controlled when the system is completed. The interface behaviour of the Conveyor process, for example, has already been defined by the two modes *ongoing* and *stepped*, corresponding to the modes of operation of the required system function.

4.4 Construction of Control Functions

The interface processes are grouped into asynchronous clusters. The partition into clusters determines how the system can be implemented later on by concurrently running CIP components. A possible reason to refine the initial clustering is modularisation: because of their asynchronous behaviour, clusters represent very weakly coupled functional blocks, well suited to development and validation by different members of the development team.

To bring about the required behaviour of the environment, function processes are created and connected appropriately with the established interface processes. The primary functionality of the system is first developed, based on the normal behaviour defined by the model processes. To permit reaction to unexpected events, the interface processes concerned must usually be extended by error modes; also, additional supervisor and error handling processes must be introduced.

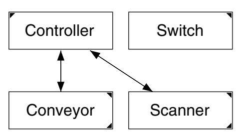
TCS - Complete Model

The CIP model of the simple case study consists of one cluster only. The function process Controller controls the cooperation of the Conveyor and the Scanner processes. The required modes of operation are modelled by corresponding modes of the Controller. The current state of the Switch master process determines the active mode of the Controller and the Conveyor process. The ongoing mode of the Conveyor interface process has been extended by transition 2; the transition is necessary because the switch can be set off even when the conveyor is stopped for scanning. The Controller process uses a timer supported by the CIP Tool; provision of such timers is a form of ‘modelling sugar’. Because the TIMEUP_ trigger represents an external cluster input, a Controller cascade is also needed.

Remark. The corner marks of a process box indicate external input or output respectively: top right: channel input, top left: timer input, bottom right: channel output.

CLUSTER TransportControl

INTERACTION NET



PULSE TRANSLATIONS

Controller.move	-> Conveyor.move
Controller.stop	-> Conveyor.stop
Controller.scan	-> Scanner.scan
Conveyor.loaded	-> Controller.loaded
Scanner.scanned	-> Controller.scanned

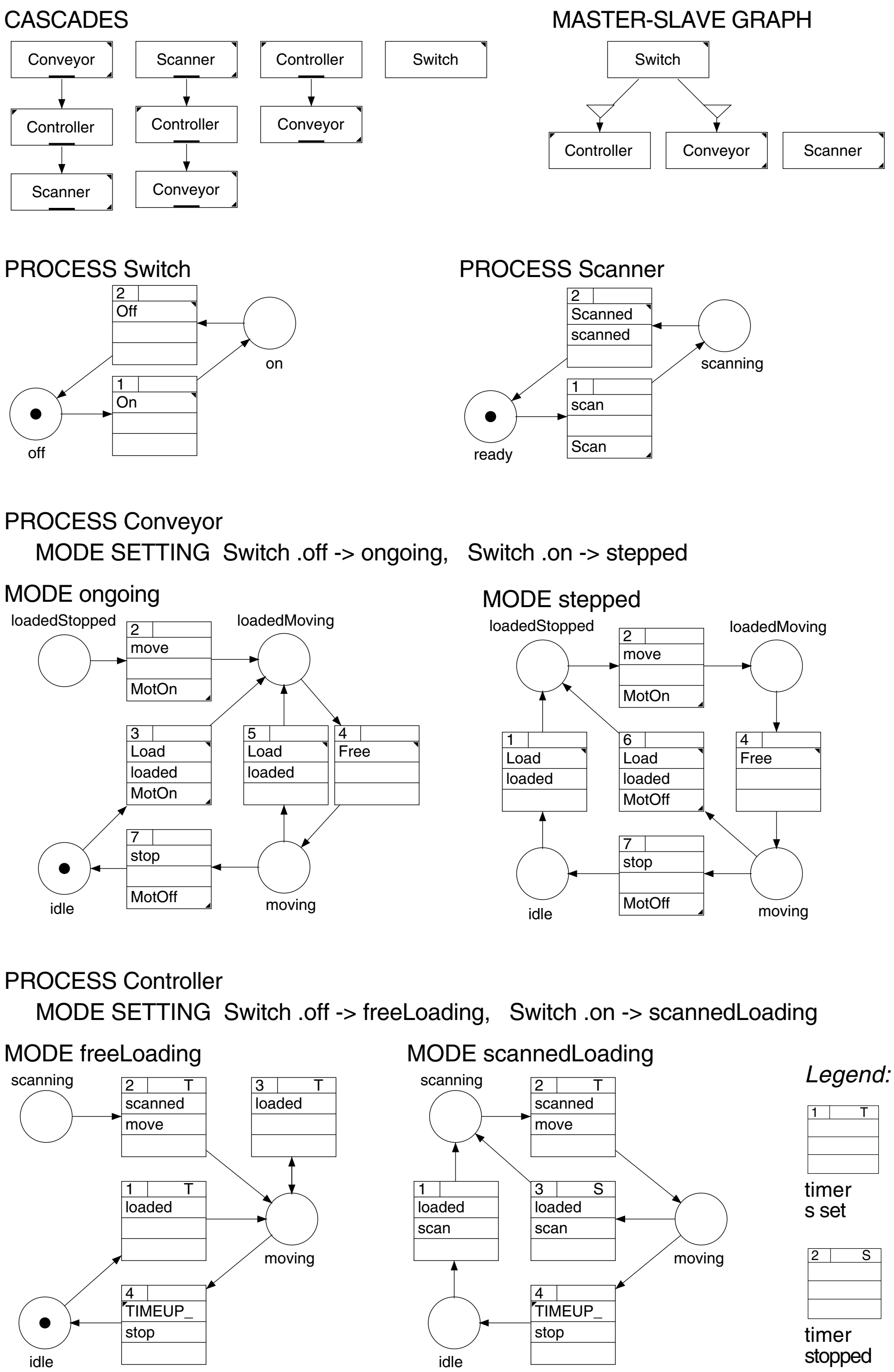


Fig. 14. Complete model specification

The model can be animated by "entering" event messages or the TIMEUP_ trigger. The following trace describes five cluster transitions of a particular animation:

PROCESS	MODE	PRESTATE	INPUT	POSTSTATE	OUTPUT
Conveyor	ongoing	idle	Load	loadedMoving	loaded, MotOn
Controller	freeLoading	idle	loaded	moving	T
Switch	-	off	On	on	
Conveyor	ongoing		POSTMODE stepped		
Controller	freeLoading		POSTMODE scannedLoading		
Conveyor	stepped	loadedMoving	Free	moving	
Conveyor	stepped	moving	Load	loadedStopped	loaded, MotOff
Controller	scannedLoading	moving	loaded	scanning	scan, S
Scanner	-	ready	scan	scanning	Scan
Scanner	-	scanning	Scanned	ready	scanned
Controller	scannedLoading	scanning	scanned	moving	move, T
Conveyor	stepped	loadedStopped	move	loadedMoving	MotOn

Fig. 15. Animation trace

5.5 More about Events and Actions

The elaboration of event and action collections represents a crucial development step because it determines the level of abstraction used to solve the functional problem. On the one hand, the collected events and actions must suffice to bring about the required behaviour of the environment. On the other hand the feasibility of the connection must be ensured by verifying that all events can be recognised and all actions can be produced by means of the available interface devices.

Events and actions can have attributes which are transmitted as data of the corresponding channel messages. An event attribute describes a circumstance of an occurring event: for example, the bar code read by a scanner or the parameters of an operator command. An action attribute describes a circumstance to be brought about when the action is performed, such as the required liquid flow of an opened valve.

Events are classified into process events and temporal events. Process events are caused by the autonomous dynamics of external processes, while temporal events occur at prescribed points in time. In general, a process event is related to a discrete change of the process. Discrete states often denote a whole range of external process states, thus representing abstractions essential for the functional behaviour required. Examples are the level ranges of a liquid in a vessel, or the various ready states of a complex device. Continuous states must be monitored by sampling events that capture the behaviour of continuous processes. Sampling events are periodically occurring temporal events whose attributes are the sampled process states. Similarly, the embedded system influences continuous processes by repeated production of attributed actions.

5. Component-Based Implementation of CIP Models

For implementation of a CIP model the set of clusters is partitioned into *units*. Each CIP unit can be transformed automatically into a CIP component, executable in a concurrent thread of the implemented system. The code generated from a CIP unit consists of a *CIP shell* and a *CIP machine*, and is produced in two individual gene–ration steps.

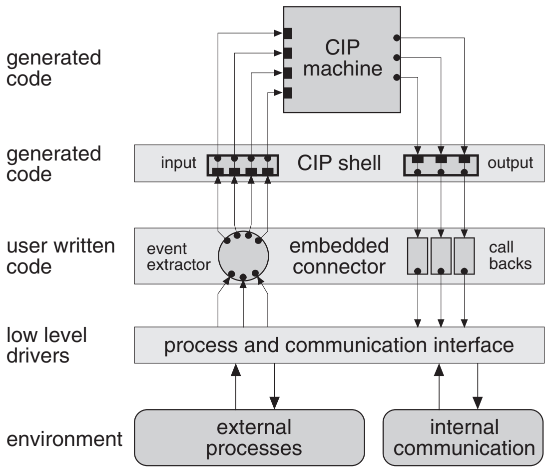


Fig. 16. Implementation of a CIP unit

The *CIP shell* represents the interface of the component: it consists of two linear structures of function pointers, one for the incoming and one for the outgoing channels. The CIP shell code is generated from the channel specifications only, and so is independent of the modelled clusters.

The *CIP machine* implements the reactive behaviour of the CIP unit; it is activated by channel function calls through the input shell. Every call triggers a cluster transition from which channel functions are called through the output shell.

Partitioning the model creates additional connection problems due to the channels interconnecting the CIP units. Thus the task of implementing a CIP unit is twofold. On one hand, a part of the controlled processes must be connected to the CIP machine: these connections correspond to the source and sink channels of the CIP unit. The implementation of this part of the connection demands tools and techniques suited to the technology of the interface devices. Communication between CIP units, on the other hand, can usually be implemented by means of standard transmission techniques based on field bus systems or serial connections.

Because of the cooperation of parallel entities modelled within CIP models, there is no need to implement conceptual parallelism by means of multi-tasking. Only one or very few CIP units are usually implemented on the same processor. Task scheduling and interrupt handling are thus reduced to hardware interface functions and background services [14].

The environment-oriented development process of the CIP method allows the various CIP shells to be modelled at an early stage. The tool supports locking of these interface specifications for extended periods. The generated shell code serves as semi-rigid joints among CIP machines and embedded connectors. Thus once a CIP shell is defined the associated CIP machine model and its embedded connector can be developed concurrently. The concept has been proven in a number of industrial and academic projects, where different partitions of the same CIP model had to be connected to simulation models, to test beds for specific system parts, and to the real environment of the final target system. In the development of a hybrid car, even the code for the connector has been generated from formal connector descriptions [15].

6. Summary

The CIP method is tailored to control problems typically encountered in the development of embedded systems. Identifying the characteristic difficulties of this problem class, the method offers suitable development concepts and modelling techniques to promote system development based on engineering activities.

A central difficulty in the development of embedded systems results from the need to monitor and control real world phenomena by means of specific interface devices. The general embedded system problem is therefore decomposed into a functional problem to be solved by means of formal behavioural models, and a connection problem demanding development techniques adapted to the technology of the interface devices. By stabilising dependencies at an early stage, the resulting development process allows the two problems to be solved independently of each other.

Most reactive behavioural models support either synchronous or asynchronous cooperation of behavioural entities. In order to support internal synchronous interaction propagation as well as flexible distributed implementation of system parts, the CIP method combines both cooperation paradigms within the same model. CIP models consist of asynchronous clusters of processes that are synchronously cooperating extended state machines.

In order to make interaction dependencies among processes explicit CIP models are constructed by means of architectural composition. However, the problem of conflicting and unbounded internal interaction can be solved only by restricting the set of possible interaction paths. CIP therefore requires the control flow of interaction to be specified by process cascades, resulting in deterministic behavioural models with bounded response times.

The difficult task of structuring behavioural models is supported by a novel hierarchical structure: the master-slave hierarchy. The hierarchical structure is based on master processes which induce high level behaviour changes in designated slave processes. The problem-oriented hierarchy relation is well suited to express powerful behavioural abstractions. This concept has proved much more flexible than rigid nesting of transition structures.

Control functions of embedded systems must maintain an ongoing behavioural relationship with the controlled external processes. The model construction process therefore starts by developing a behavioural context model that defines the legal histories of external interactions. The full functional solution is constructed in further development steps where function processes are added and connected to the context model.

The integration of the CIP code is based on component technology. Various configurations of generated software components can be connected to interface modules and components supporting other concerns such as validation and simulation. The resulting flexibility in building executable system parts becomes crucial when developed systems have to be validated in various test environments.

References

1. CIP Tool® - User Manual (1998). CIP System AG. Solothurn, Switzerland. Internet: http://www.ciptool.ch
2. Cameron J.R.(ed.): JSP and JSD: The Jackson Approach to Software Development. IEEE Computer Society Press (1989)
3. Ward. P. T., Mellor. J. M.: Structured Development for Real-Time Systems. Yourdon Press, Prentice-Hall, Englewood Cliffs, New Jersey (1985)
4. Zave P.: The Operational Approach versus the Conventional Approach to Software Development. Comm. ACM Vol. 27 No. 2, (1984) 104-118
5. Harel D.: Statecharts: A Visual Formalism for Complex Systems. Science of Computer Programming Vol 8, (1987) 231-274
6. Berry G, Gontier G.: The ESTEREL Synchronous Programming Language: Design,Semantics, Implementation. Science of Computer Programming, Vol. 19 (1992) 87-152
7. Caspi P, Pilaud D, Halbwachs N, Plaice J.A. LUSTRE: A declarative language for programming synchronous systems. In: Fourteenth Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, Munich (1987) 178-188
8. Faergemand O. (ed): SDL '93: Using Objects. Proceedings of the 6th SDL Forum. North-Holland (1993)
9. Selic B., Gullekson G., Ward P.T.: Real-Time Object-Oriented Modeling. John Wiley & Sons (1994)
10. Jackson M.A.: Problem Analysis Using Small Problem Frames. To appear in South African Computer Journal special issue for WOFACS'98. (1999)
11. Fierz H.: SCSM - Synchronous Composition of Sequential Machines. Internal Report No. 14. Computer Engineering and Networks Laboratory, ETH Zürich (1994)
12. Medvidovic N., Taylor R.N.: A Framework for Classifying and Comparing Architecture Description Languages. In: Proc. ESEC/FSE '97, Lecture Notes in Computer Science, Vol.1301, Springer-Verlag Berlin (1997) 60-76.
13. Fierz H., Müller. H.: Behaviour Control in Reactive Systems: Constructing Master-Slave Hierarchies in CIP Models. Submitted to 11th Euromicro Conference on Real-Time Systems. York, England (June 1999)
14. Trutman HO.: Well-Behaved Applications Allow for More Efficient Scheduling. Submitted to 24th IFAC/IFIP Workshop on Real-Time Programming and the Third Int. Workshop on Active and Real-Time Database Systems. Schloss Dagstuhl, Saarland Nay (1999)
15. Trutmann HO.: Generation of Embedded Control Systems. 23rd IFAC/IFIP Workshop on Real Time Programming, WRTP 98, Shantou, P.R. China (1998) 99-104