

# Specification and Validation of Information Processing Systems by Process Encapsulation and Symbolic Execution

Wolfgang Boßung, Thomas Geyer, Sorin Alexander Huss and Lars Wehmeyer  
Integrated Circuits and Systems Laboratory, Department of Computer Science  
Darmstadt University of Technology, Alexanderstr. 10, 64283 Darmstadt, Germany  
bossung|geyer|huss|wehmeyer@vlsi.informatik.tu-darmstadt.de

**Abstract** - A process notation based on a functional partitioning of a new system is proposed as a High-Level CoDesign Model for specification, evaluation, and implementation purposes. A well-defined computational model allows the synchronization and activation of concurrent processes. The design environment includes the refinement steps from scheduling with dynamic processes via simplified scheduling/execution methods to the complete implementation. The relationship between specified behavioral classes and implemented states in the final core programs is the key issue in this development flow. Time uncritical parts are implemented in a heterogeneous environment communicating via a standard communication protocol. On the other hand, time critical processes and process connections can be implemented in C++ using SystemC, thus forming core programs encapsulated in a shell to fit the introduced and implemented activation rules. An example from the world of digital image processing systems illustrates the approach.

## 1 Introduction

The question whether functional entities were designed according to the given specification is of great significance in the area of rapid prototyping, reuse, and encapsulation of intellectual property concerning hardware/software systems. The encapsulation of functional parts within entities bearing clearly defined Input/Output relations is a promising approach in order to solve details regarding this subject. An overall design model in form of concurrent processes as proposed in our approach has to describe the whole system in a consistent way (i.e., the I/O relation interface must be given for all process descriptions during every design phase). Time constraints further enforce requirements for communication methods. The methodology of activation and synchronisation of communicating functional parts thus becomes a key issue in HW/SW-Codesign. A new method for process activation forms the base for the CoDesignModel (CDM) proposed in [1]. This model permits capturing the timing behavior and early scheduling as well as the subsequent integration and execution of implemented system functionality. Every functional part is encapsulated in a CDM shell. This shell, designed in the first step of development, is used during the entire implementation. The complete encapsulation of functional parts with well-defined interfaces can thus be guaranteed in every design phase. Analysis of data- and control-flow by means of a conditional scheduling enables investigation of timing behavior in the beginning design phases as shown in [2]. The functional partitioning and the graphical representation in the form of CDM graphs make this model a means of communication among groups of specifiers belonging to different areas of expertise. The model is easily understood and can serve as the base for discussion. Usage of the model in practice is demonstrated by the example of an image processing system.

## 2 Related Work

Every model used for evaluating a specification should have a determined operational model. This fact is presently not adequately considered enough in specification issues for high-level modelling. However, an initiative in form of a denotational “Framework for Comparing Models of Computation” is presented in [12]. In addition to the commercialized CoWare tool set [17], some new approaches based on process notations with similarly well-defined computational models can be distinguished. In [21], a process net notation (SPI “System Property Intervals) is described for specification, which allows scheduling and allocation of systems with more than one execution model. An extension of the SPI system as described in [22] allows for the description of alternative kinds of process behavior and their coherence using so-called modes, which focus on the number of consumed I/O tokens by a process. FunState [16] enhances the SPI model by possible explicit modeling of state informations.

Dynamic process structures are investigated in [6], where conditions are assigned to processes. Depending on the conditions defined, different data dependencies are given. The underlying communication model is based on separate communication units for every process and leads to a process model that is always ready for communication. Multiple executions of a task lead to the notation of multiple instantiations of processes. One drawback is the high impact that a slight variation in the reactive and temporal behavior of a process can have on the respective timing analysis of the task graph. This issue, discussed in [1], is not addressed. The CASPER project [5] advocates acyclic task graphs as the basic graph structure for the specification task. For the first time it is thus possible for acyclic task sequences to be of a periodic and aperiodic nature. With respect to a system realization, the necessary time scheduling is often based on the consideration of acyclic task or process graphs as denoted in [20]. The specification methodology proposed in this paper is based on the communicating process structure of a CoDesign Model. This structure has the character of an iterative cyclic problem graph. As an overall model of concurrent processes it describes the whole system consistently during every design phase according to the interface of I/O relations for every process description.

## 3 The CoDesignModel

The vertices of a CDM graph represent the participating processes used to solve a complex task. The edges show the data dependencies and the communication interfaces among these processes. They serve as a means for data- as well as control-flow. Every process is attributed with a set of so-called *conditional input/output relations*, which further describes the process behavior. The I/O relations given in Figure 1 state that process *P1* may exhibit two different kinds of behavior upon receiving data from edge 1: either an

output is generated on edge 2 according to I/O relation  $\langle 1|2, S_0 \rangle$ , or there is no output, according to  $\langle 1|\{\}, S_1 \rangle$ .

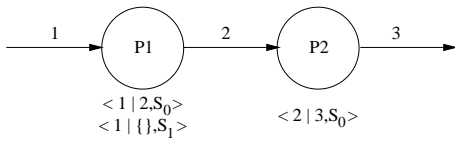


Figure 1: Example CDM graph

Thus,  $P1$  has two *behavioral classes* or *states* it can be in at a certain point in time: when in state  $S_0$ , an output is generated on edge 2 upon receiving data from edge 1, whereas for state  $S_1$ , no reaction is observable from the outside. Such behavior is common in practice (e.g., when only values within a certain range may pass a filter; i.e. a data-dependent change of state). On the other hand, a process may pass every other input to an output edge, regardless of its value. Note that *state* is used as a synonym for *behavioral class*. The definition of different behavioral classes for a process is necessary whenever a process may show different behavior for identical combinations of inputs. Every process has an actual status. Four possibilities are distinguished:

- *idle* (i.e., waiting for new data),
- *communicating* with one of its neighboring processes,
- *waiting* until its successor receives the newly computed data,
- *computing*.

Precise formal definitions regarding *communication time*, *computing time*, *process allocation*, etc. including a discussion of different activation rules are outlined in [2].

### 3.1 Target Scenarios

A CDM graph is generated by *functional partitioning* of a technical system. This kind of partitioning is well adapted to the developers' view, as engineers tend to think of a system in terms of functional units. This allows the CDM graph to act not only as a specification tool but also as an interface: it represents an efficient means of communication between the engineers delivering the technical specification of subcomponents on one side and the computer scientists on the other. The latter assemble the components to form the system as a whole and evaluate feasible implementation alternatives. Apart from that, a CDM graph model of a system is also executable so as to be able to determine if the components communicate and perform the way the designers intended them to.

If for example a new image processing system is to be implemented, the developer should have an idea of the functional partitioning of such a system. Early functional partitioning of the whole system allows distributed execution of implemented processes for evaluation purposes. A pure TCP/IP communication between processes is a useful entry point for rapid prototyping, as a design team generally needs support for many different operating systems and platforms for a thorough exploitation of design space. A heterogeneous target scenario for realization of the specified system may contain code for a specific DSP, VHDL descriptions as design entry for various special hardware, or code for software running on particular processors with various operating systems.

A possible scenario for the execution of a CDM graph is distributed execution on computers in a network cluster, whereas each

CDM process is mapped to a computer in the workstation cluster. If the processes are each mapped to a different computer, a genuine parallel execution within the cluster is realized. In case of a heterogeneous network employing different operating systems, execution is performed across the different architectures. In the final implementation, several processes may be executed in parallel. In the CDM, this behavior is modelled by distributing the processes of a CDM graph among the workstations within a cluster of hosts. A sequential execution may result as a consequence of mapping all processes in a CDM graph to only one processor during resource binding (i.e., only one single processor is used to execute the entire CDM graph).

Another scenario assumes some parts of the system have already been realized as hardware and their description is provided in the form of VHDL code. This code may be simulated using the CDM interface to the VHDL simulator. Alternatively, the code can be used as input for programming an adequate FPGA board. The remaining CDM processes in this scenario are executed on a single multiprocessor (MP) computer. Using this architecture, a parallel execution can be obtained on one host.

The last scenario deals with real time systems. In this approach, a fixed scheduling like the one determined in [11] is executed on a distributed multiprocessor system with TCP/IP-connections. The exact compliance with all requirements is crucial for the success of this method. Among these is the utilization of the "single user mode" on all incorporated computers as well as a guaranteed minimal bandwidth on communication channels combined with a specially designed resource management.

### 3.2 CDM shell

Processes communicate via links depicted as graph edges in Figure 1, exchanging control-information as well as data. The observable reactions to incoming data occur according to the specified I/O relations of each process. In order to implement the core program (which forms the actual algorithm), a uniform and precise interface must be defined, which allows the core program to be fitted into the CDM context. To keep this interface as universal and flexible as possible and at the same time reveal only the information needed to implement the core program, the interface is encapsulated within the CDM shell (*CDMsh*). The CDMsh thus forms a uniform and standardized interface that must be used for any kind of interprocess communication. Data arriving on the incoming links is passed on to the core program according to the internal interface. The CDMsh is also responsible for the compliance of process reactions with the given I/O relations. Results are passed to the outgoing links according to the current state (or behavioral class) of the process. The external interface adapts communication connections via TCP/IP as described in detail in [18].

Finally, the CDMsh implements the worst-case-execution-time-property by delaying outputs in such a way that reactions are only observable after the specified worst-case times. New data arriving on the incoming links is passed on to the core program once the first I/O relation becomes valid. The core performs its manipulation on the data and passes the results back to the CDMsh, which is in turn responsible for communication with the successor processes after the worst case execution time has elapsed. The CDMsh then waits for new data from the predecessor processes.

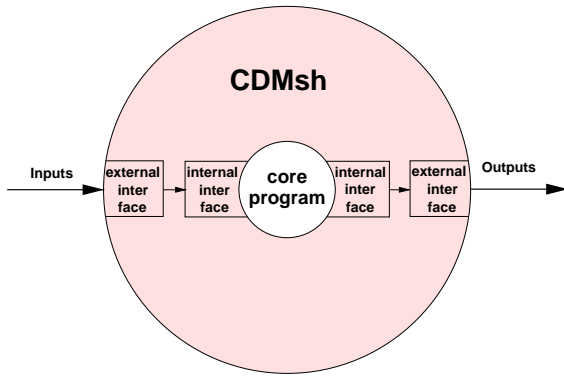


Figure 2: Generic CDMsh Interfaces

The CDMsh is thus active during the whole execution time of the CDM graph, whereas the core program is only triggered to perform the actual calculation.

### 3.3 CDM core

The core program is the actual working algorithm within each CDM process. It is only responsible for the necessary calculations - all other tasks like communication or observance of I/O relations are performed by the CDMsh. The required parameters are passed to the core program from the CDMsh using the internal (intra-process) interface, whose realization depends on the implementation of the core program. Uniform interfaces are used for several common implementation approaches for the core algorithm:

- *Tcl*: a scripting language [13] used as a basis for a first software implementation in the sense of rapid prototyping
- *C/C++*: high-level programming languages, resulting in an efficient software implementation that may satisfy the performance needs of a number of applications
- *SystemC*: a library of C++ classes, used to model hardware properties such as concurrency, timing and reactive behavior in C++
- *Other executables*: enables the CDMsh to execute arbitrary precompiled programs for the given platform with the goal of being able to reuse legacy code
- *Cantata workspace*: data and control flow descriptions programmed with the graphical user interface of the *khoroS* [10] library may be directly incorporated into the execution of a CDM graph
- *VHDL*: this common hardware description language is used to enable simulation of hardware specifications within the CDM graph
- *FPGA board*: a genuine hardware realization that is both efficient and flexible (*hardware in the loop*)

Using the interface paradigm in combination with a strong encapsulation of communication functions in the CDMsh, an independent implementation becomes possible: a developer may implement the core program according to his own conventions. The insertion takes place at one single spot in the CDMsh code, which is well commented. This allows an easy integration of any core program into the CDMsh. The remaining parts of the implementation of the

CDMsh generally remain invisible to the core developer, preventing the abuse of internal implementation details. In the case of VHDL and SystemC a more detailed description of useful adaptations will be given in the next sections.

### 3.4 VHDL

The real execution of the VHDL code is effected with the VHDL-C interface using CLI (C Language Interface). This is explained in detail in [14]. With the CLI-enhancements, VHDL architectures can be modelled using the C programming language as depicted in Figure 3. The CLI components are introduced to the VHDL simulator with special commands. This interface can also be used for the implementation of CDM processes in VHDL: the VHDL simulator is started and waits for input signals from a CLI component implemented in C. This input signal generator is initiated via the Tcl-C interface in such a way as to generate signal changes at the input of the VHDL component at those points in time when an I/O relation within the CDM process becomes valid, thereby activating the modelling VHDL process. The output of the results occurs analogously via an additional CLI component, which takes the data from the VHDL code and returns it to Tcl. The CDMsh can then convert and process these results.

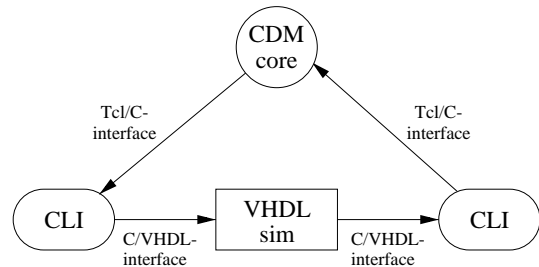


Figure 3: Encapsulation of VHDL Code

### 3.5 SystemC

In the case of SystemC, additional C++ libraries can be used both for core programming and as an overall target specification of the whole system or parts of it. The channel construct of SystemC [15] allows a new approach to evaluate an executable specification using a description of complete functionality combined with the integration of activation rules into main control programs of processes (c.f., *lps\_main.cc* in Figure 5). Refinements concerning communication details - like special blocking constructs between processes - can be incorporated at this channel specification level. It is thus possible to take a closer look at buffering and fifo behavior, which have until now been neglected in many system specifications. The detailed core program integration in form of a SystemC program part is depicted in Figure 5. For clarification we first take a look at a complete CDM specification of an implemented laser mouse tracker given in Figure 4.

The image of a computer screen can be projected onto a wall or any other planar area for presentation purposes. A second mouse system, controlled by a laser pointer in the hand of the speaker, forms a complex image processing system, which sends instructions for mouse interactions to the computer. This image processing system consisting of a camera and some computing resources has to detect the laser point in the captured image of the computer screen on the wall.

Con can decide whether an *Orientation Mapping* is to be performed or if the system should detect the laser pointer to control mouse movement. *Orientation Mapping* is necessary prior to laser point detection in order to compensate for projection distortions of both the projection surface and the camera optics.

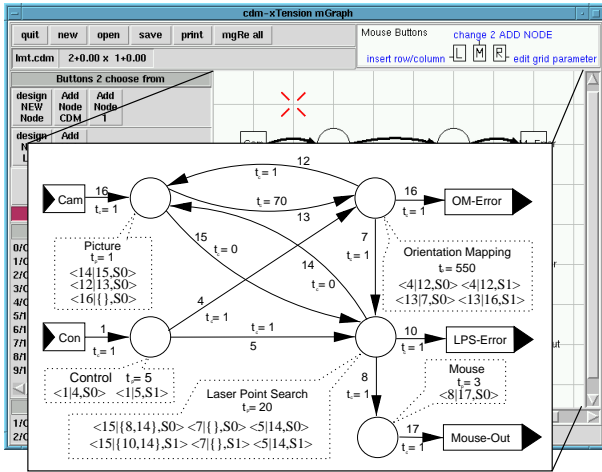


Figure 4: CDM of Laser Mouse Tracker

The camera delivers its pictures to either the *Orientation Mapping* or the *Laser Point Search* process, depending on which of the two requests an image. A picture is passed over to the communication edge only after a request, that is, after the previous image has been processed.

The *Orientation Mapping* either yields an error or passes its orientation parameters to the *Laser Point Search* process. The latter uses these parameters to determine the actual coordinates in the projected desktop image (as opposed to the pixels in the camera image). The *Laser Point Search* yields an error when no laser point is detectable; otherwise, it will pass the coordinates of the laser pointer to the *Mouse* process, which in turn updates the position of the mouse cursor on the screen.

The first task after this functional partitioning is the evaluation of functional blocks, which are given as CDM Processes. The following I/O relations belong to the laser point search process and are given in a more readable manner with successor and predecessor process names as synonyms for the edge IDs from Figure 4:

```
< picture | {mouse, picture}, S0 >
< orientation mapping | {}, {S0, S1} >
< picture | {lpserror, picture}, S1 >
< control | picture, {S0, S1} >
```

For coding the detection of a single point in a captured image an input file connection was first used for simplification. The realization of communication structures considering the callback structure of the video driver in the *Picture* process will now be discussed by notation of a SystemC core program and its interfaces. Figure 5 shows SystemC components of the *LPS* core program.

In `sc_main()` channels and processes are instantiated. There are two processes: `testbench` and *LaserPointSearch* (*LPS*) which are linked by a number of channels.

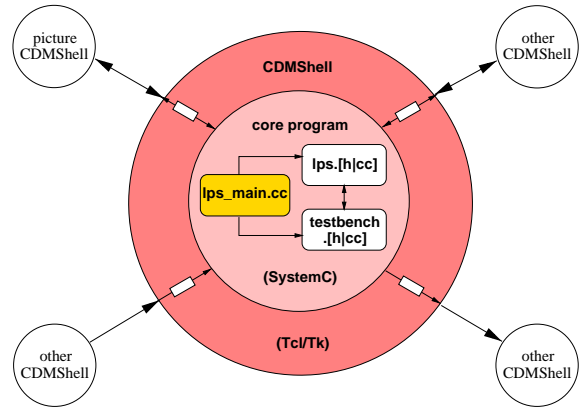


Figure 5: Encapsulation of SystemC Code

When simulation starts, the *LPS* process consumes the data sent via the channels from `testbench` to *LPS*, does some calculation and returns the result to `testbench`. The latter then returns the result to the CDMSh.

```
sc_main()
// do some command line processing here
(...)
//-----
// instantiation of channels
//-----
sc_channel_array <sc_array <double> >
  o_mapping(om_count, 1);
sc_channel <bool>
  o_mapping_helpflag(1);
// Activation of processes that are sensitive
// on channel arrays needs this workaround.
// Process is sensitive on helpflag too
// and activated by change of helpflag data.
sc_channel <bool>
  control(1);
(...)
//-----
// instantiation of processes
//-----
lps_laserpointsearch lp_search // sc_aproc
  ("laserpointsearch", // lps_laserpointsearch
  picture, picture_helpflag,
  o_mapping, o_mapping_helpflag,
  control, lps_error, pic_request,
  x_value, y_value, data_size);
lps_testbench t_bench // sc_sync
  ("testbench", main_clk.pos(), // lps_testbench
  lps_error, pic_request, x_value, y_value,
  picture, picture_helpflag,
  o_mapping, o_mapping_helpflag,
  control, pic_data, om_data, control_flag,
  command_line_indicator);
//-----
// start simulation
//-----
sc_start(2);
```

In `lps_testbench::entry()` (the behavior of the testbench process) writing and reading to/from the channels to *LPS* is done. Note that the writes must be non-blocking (`.nb_write()` instead of `.write()`) and the channels written to must have at least one buffer. Otherwise, the first write in a sequence of 'ordinary' writes would block execution. Before reading a result, a check for available data on the channel must be performed (`.data_available()`).

```
lps_testbench::entry()
while (true) {
  // check if parameter was supplied by CDMSh
  // and write values on the appropriate channel
  if (cmd_line_indicator[2] == 1) {
```

```

    control.nb_write(control_flag);
}
(...)
// wait for response from asynchronous process
wait();
if (pic_request.data_available()) {
    // print value of channel
    cout <<"PIC_REQUEST:"<<pic_request.read() <<"\n";
}
wait();
}

```

In `lps.laserpointsearch::entry()` (the behavior of the *LPS* process) each channel is checked for new data and read if this holds. Then the appropriate computation defined by the core function is done.

```

lps.laserpointsearch::entry()
while (true) {
    // [1] Mapping of the IO-Relations
    //-----
    // IO-Rel: <orientation | { }, state0>
    //         <orientation | { }, state1>
    //-----
    // Check if the input is available:
    if (ch_o_mapping.get_channel(0)->data_available()){
        // read available channel
        o_mapping_param = ch_o_mapping.read();
        // execute core program
        core(); // results written on channels in core
        (...)
        wait();
    }
}

```

The usage of channels for flexibility requires no specification of communication details. In the case of communication with the *Picture* process, the file-save connection used for the general test of the search function is not applicable because search results have to be determined at least 5 times a second. Consequently, the core programs of *Picture* and *LPS* have to be merged as shown in Figure 6 and the CDMsh has to be enhanced by I/O relations of the *Picture* process. I/O relations regarding communication between *Picture* and *LPS* are now handled internally.

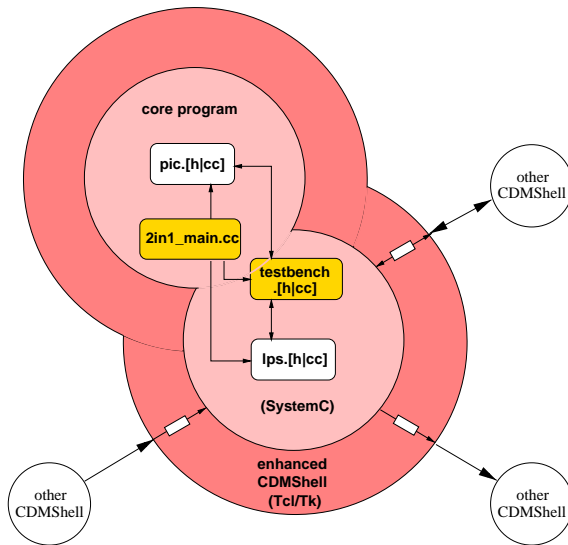


Figure 6: Two in One - SystemC Interfaces

Therefore, the process loop of every process has to be refined as shown in Figure 7. The INPUT LOOP body is required to react on every new input and current data set that matches an I/O relation. In

this case core function will be executed. The OUTPUT LOOP is used to wait for success of delivering data to all successor processes. That means that every consumption of data generally has to give feedback via separate channels the producer is sensitive to.

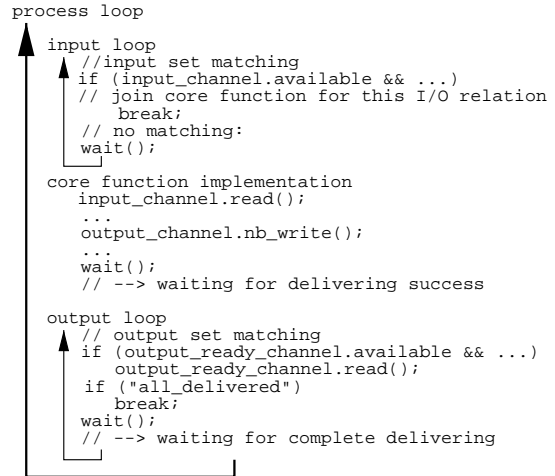


Figure 7: Two in One - SystemC Pseudo Notation

Note that the distinction of *Picture* and *LPS* process is required for a scheduling with an activation rule for CDM graphs as described in [2]. It must be possible to distinguish from the outside of an hierarchical process which internal process is currently active. A way to handle this problem is to allow the consumption and generation of data from and to the I/O edges within an hierarchical process during execution of internal processes. Solutions regarding this enhancement of activation conditions are currently under research. During communication refinement steps, SystemC channels may be changed into SystemC signals and/or enhanced by specialized buffering structures.

### 3.6 CDM Monitor

Since a CDMsh process has to be started for every process at the beginning of execution in a CDM graph, it makes sense to introduce an entity to perform and control this initial step. Alternatively, it would be possible to have every CDMsh carry responsibility for the successor-processes it spawns. This, however, leads to a strong duplication of source code and is not part of the tasks to be accomplished by the CDMsh. For these reasons, the so-called CDM monitor (*CDMon*) is introduced. It supervises the proper start of the processes and CDM shells as well as their behavior during execution. It is necessary to periodically check whether CDM shells that were started in the beginning are still running and ready to receive data.

## 4 Symbolic Execution

The actual execution of a CDM graph requires a complete implementation of all participating processes as well as I/O relations for their corresponding CDM shells. However, by using the conditional scheduling introduced in [11], it is possible to extract some properties of a specified system without first having to implement every process. Figure 8 illustrates the coherence between execution and scheduling. A conditional scheduling represents a symbolic execution of a CDM graph, during which certain assumptions have to be made. The I/O relations that are specified along with the processes

are needed as well as timing information for the processes. The latter values are obtained either by means of specified restrictions, by estimation, or by an already implemented version of the process. With this information alone it is possible to generate a conditional execution chart which represents the different possibilities of system behavior. All possible alternatives are listed in the execution chart. However, during a “real” execution, only one of these alternatives is chosen and pursued. If a process has two states due to different possible reactions on data on one incoming edge, the scheduling needs to be split into two parts in order to account for the consequences that arise from each of the two possibilities.

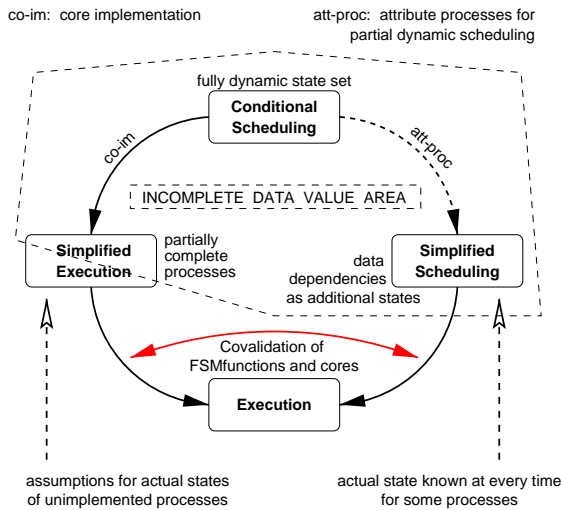


Figure 8: Validation during Refinement

In the case that some, but not all processes within a CDM graph have been implemented, a combined method is applicable: the processes with more than one behavioral class that have not been implemented cause the schedule to be split. The completely implemented processes have a known state at every point in time and are simply added to the chart without causing a split. This approach is called *simplified scheduling*, as the number of states that have to be traced in parallel is reduced compared to the full dynamic conditional scheduling approach. This is worthwhile to exploit, as the runtime complexity of scheduling depends exponentially on the number of states. Instead of simplified scheduling, it is also possible to perform a *simplified execution* of a CDM graph: Those processes that have already been implemented are treated as they would be during regular execution. If a process is to be executed, but no implemented core program is available yet, the CDMsh simply takes over the responsibilities for choosing a specific behavioral class and ensures a behavior according to the I/O specification. This means that it is no longer necessary to split paths.

Timing constraints have to be observed during execution of a CDM graph as well as during scheduling. Since scheduling is performed based on a “worst case” scenario, this behavior is also adapted for the actual execution, so that the results obtained during scheduling remain comparable to the observed execution times. This implies that each CDM process has to take all the time that was specified as being its worst case execution time before passing calculated data to the next process. If the core program finishes its computation before the specified time, the CDMsh will delay the propaga-

tion of outgoing values until the timing requirements are met. In this way, the CDMsh suppresses data-dependent differences in execution time.

## 5 Behavioral Classes

One of the characteristics of a CDM graph is the fact that it is not necessary to explicitly specify the state transitions of a process. During the conception phase the designer can, therefore, rely on the core programs: they will later take over this responsibility. The I/O relations of a process list the possible reactions on any combination of inputs. This specification is well suited for the engineers that specify the functional behavior of the system. Since a CDM graph is usually constructed by several people with different perspectives on the process, this common functional partitioning view is a central advantage of the CDM method. During functional partitioning, the reactive behavior of the system has to be specified. If there are two possible alternatives for a reaction to one specific input, then the system will automatically generate two different states or behavioral classes for this process.

### 5.1 Process State Refinement

It is, however, possible to denote known state transitions along with the processes. The advantage of supplying this information is a more precise model of system behavior. Especially the data-independent state transitions are fairly easy to capture, as they may be specified in the form of an FSM description within the CDM process. Data-independent in this case means that it is not necessary to know the actual data values. If the state transitions of a process can be specified using an FSM, it is easily possible to incorporate specific tools for use with FSMs (cf. [4, 7, 8, 9]). The data-dependent case is harder to express within the CDM design, as the state transition actually depends on the value of the incoming data, which is unknown during scheduling. Any additional knowledge about the characteristics of state transitions helps to restrict design space and therefore reduces the high effort required to perform conditional scheduling. For a complete specification, such additional information needs to be supplied for all participating processes. However, a formal definition of all state transitions will not generally be feasible, as this would correspond to a complete implementation. It will be possible to add attributes representing formal constraints to any process of a CDM graph in an envisaged stage of development. These constraints express conditions regarding fully dynamic, partially dynamic and fully static state transitions and dependencies.

### 5.2 Notation of Behavioral Classes

The processes within a CDM graph can have different internal structures, despite their uniform description. One process generally has several internal states that are a consequence of the different behavior patterns the process may show following a certain input token. The transitions between states do not have to be specified - they may be hidden within the core program of the process. However, by taking a closer look at these transitions, one notices that they can be distinguished by common qualities: A transition can be data-dependent or it may occur independent of the input data value. In the first case, the input data is consumed, its value analyzed, and some computation is performed on that value within the core program. Depending on the computed value, the next state of that process is determined. In the second case, the following state may be determined without even looking at the data value - the fact

that a token is present is sufficient for a specific transition to occur. Calculations on the token value may still be performed by the core program; however, they have no influence on the state transition. In this latter case, the transition function can be specified in the form of a Finite State Machine (cf.[19]). It is than possible to use specialized tools (e.g., *StateMate* [9] or *SGM* [8]) in order to model the behavior of the corresponding processes. Contrary to the CDM model, *StateMate* initially assumes that all transition functions are specified right from the start. A direction for future research work is to investigate a possible combination of these two paradigms and to study advantages and disadvantages one may have over the other, or even develop a system that combines the power of both in order to draw even more information out of the specification.

From now on we will concentrate on special cases of transition functions within a CDM process. If  $P_p \ni p = (In, Out, \{State\})$  (process  $p$  only has one input and one output), there are two possibilities: Either the state of the process is **data-dependent** (depends on the value of input token  $In$ ) or the state of the process is **not data-dependent**. In the first case the transition function is generally contained in the core program. It analyzes incoming data, performs calculations on the data and determines the new process state from the result. In the latter case the transition function may be given as a Finite State Machine  $FSM = (I, S, O, in, \delta, \lambda)$ . The following equations are valid for this FSM within the CDM process:

- $I \equiv \bullet p$  Input of the process
- $S \equiv$  internal states of the FSM  
(generally  $\neq States(p)$ !)
- $O \equiv States(p)$

Considering processes with more than one input, the following forms are possible: for a process with two inputs and one output  $P_p \ni p = (\{In_1, In_2\}, Out, \{State\})$ , where  $In_1$  and  $In_2$  trigger a data-independent and a data-dependent transition, respectively, it is true that

- if there is an I/O relation with  $(I_1, Out, S_x)$ , then it is possible to express this part of the transition function as an FSM.
- if there is an I/O relation with  $(\{I_1, I_2\}, Out, S_x)$ , then one may be able to restrict the set of possible following states depending on  $I_1$ .

These revelations lead to a substantial reduction in the number of considered states during scheduling of a CDM graph. Thus, an improvement of runtime behavior can be expected at the cost of a higher description complexity.

### 5.3 CDM State Transitions

For the notation of additional state transition information every I/O relation can be enhanced by a transition function TF. So, an I/O relation becomes a tuple

$$\langle I|O, [TF_{pre} |] State [ |TF_{post}] \rangle,$$

where  $TF_{pre}$  and  $TF_{post}$  are optional transition functions, which have to be evaluated before and after activation of the core program in a CDM State (i.e., a behavioral class). These functions can restrict the possible successor state set without activation of the core program in case of  $TF_{pre}$  (checking for values of input data) and after activation of the core program. If every  $TF_{post}$ , for example, predicates a single successor state and no  $TF_{pre}$  function is given for any state of the process, the case of a fully static process behavior will be specified. Note that if no values on input data is given

(one or more only partially implemented processes) an evaluation of  $TF_{pre}$  functions makes no sense. If such a function is still specified, the scheduling has to handle further activations as conditional events as pictured in Figure 9 of the next chapter. The placement of  $TF_{post}$  also allows introduction of fault checks and restarts of core programs with changed/adapted internal parameters in the sense of fault tolerant processes. For example, this takes place for the *laser point search* in a picture with a priori unknown contrast and brightness parameters.

## 6 Application Example

Figure 9 shows the results of a conditional scheduling without specification of any state transitions of processes regarding the CDM depicted in Figure 4. Communication and pre-estimated execution times are distinguishable.

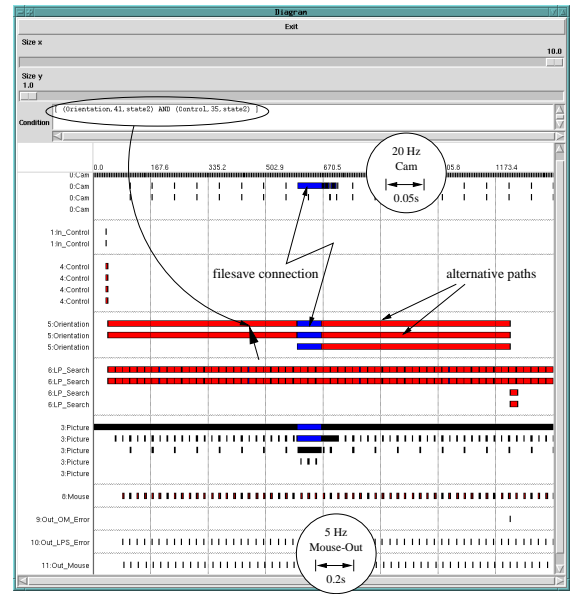


Figure 9: Symbolic Execution Trace

This simulation of token flow provides a schedule that enables investigations about the timing analysis of the proposed CoDesign Model and leads to the detection of time-critical processes and communication overhead. Predictions about the delay between any two processes of the system are also possible, as well as the speed of processing external inputs and outputs, iteration times of determined periods, and hence, all derivable time criteria. The parallel execution of the same process indicates different condition lists. For example, the condition lists for the first two parallel activations of the *Orientation Mapping* process are:

$$C_1 = \{((Control, 35, S_0)(Orientation Mapping, 41, S_1))\}$$

$$C_2 = \{((Control, 35, S_0)(Orientation Mapping, 41, S_2))\}.$$

The process *Control* must be in the state  $S_0$  at point in time 35 (i.e., it starts the *Orientation Mapping*). The *Orientation Mapping* has two different states, either success of the orientation mapping or failure. These actions can be planned in parallel, since only one case can occur during execution.

The camera picture at the *Cam* input port is supplied at a rate of 20 Hz. Upon a request from another process, the picture is relayed accordingly by the picture process. The *Con* input can only have

one signal applied to it at a time, which means that either an "orientation mapping" or an "LP-Search" takes place. This can be seen in Figure 9. It also shows that the system creates new mouse coordinates with a rate of 5 Hz (1 time unit = 0.01s). Considering the condition list for the generated output data, one can ascertain that the process "LP-Search" must have been activated and completed successfully. For more details regarding extractable timings like time delays because of a "file save to disk" connection between processes refer to [3].

The computation time for this example is twenty seconds with an Intel Pentium II 233 MHz and a Tcl/Tk implementation of the conditional scheduling. A first software-oriented laser mouse tracker realization also runs on a Pentium II 233 MHz. Except for *Orientation Mapping* and *Control*, which are implemented in Tcl, all processes are either coded in C or in SystemC. As obvious from Figure 9, the detection of 5 laser points per second can currently be guaranteed.

The conditionally noted activities permit the masking of exactly those paths that have contributed neither to the generation of output data nor the activation of a process. One can interactively exclude conditionally handled output data or process activations if the semantic interpretation of condition list shows an irregular or unintentional generation path. This implicates a restriction of the state transition for single processes. The refinement of single processes is performed by taking the knowledge of the overall system functionality into account. For instance, output data for *Mouse-Out* does not make sense without prior *Orientation Mapping* activity, which in return restricts the initial state of the *LPS* process, thus leading to "faster" conditional scheduling for these "boundary conditions". Waiting times for processes are selectively analyzable with regard to specific output data. In addition to detailed I/O rates, conditional time delays between arbitrary processes can be evaluated according to the conditional activity of the afore-mentioned processes. This is particularly mandatory for a HW/SW-Codesign in the initial state of development. The "minimal waiting time" for a specific process, considered over a longer period of simulation time, can be used as the "minimal overall waiting-time" for a prolonged execution time on a slower processor or for the minimization of resources (e.g. the number of processors). Parallel implementations of processes can be initiated on the same processor if the combination of the corresponding conditions lists shows the possibility of alternative execution (XOR execution). Interactive tools for automatic Boolean combination of condition lists are under investigation as well as tools for notation of partial FSM-informations of processes and their scheduling. The latter will close the gap between scheduling with fully dynamic processes and the simplified scheduling as introduced in Chapter 4.

## 7 Conclusions

Because of their generality, asynchronous objects in form of CDM processes can be used in a wide range of image processing systems and, furthermore, in other domains too. Starting with a CDM specification and a standard communication model for the communication between functional parts, a design team can use many different environments for exploration of the design space. Further refinement steps consider harder communication requirements between assigned processes. By using SystemC program cores as described, the critical communication connections can be specified in C++. The combination of SystemC core programs of different CDM

shells into one shell encapsulates this process construct as a standard CoDesign description and makes this part of the system useful for synthesis steps supported by future design automation tools. The consideration of VHDL core programs as "hardware in the loop solutions" offers great perspective. The overall design model of concurrent processes describes the whole system in a consistent way with respect to the interface of I/O relations for every process description during every design phase.

## References

- [1] W. Boßung and S.A. Huss. Cyclic process nets as a high-level behavioral specification model for embedded systems synthesis. IEEE CS Workshop on VLSI, Orlando, pp. 116-121, 1998.
- [2] W. Boßung, S.A. Huss, and S. Klaus. High-Level Embedded System Specifications Based on Process Activation Conditions. J. of VLSI Signal Processing, Special Issue on System Design, Kluwer Academic Publishers, vol. 21, no. 3, pp. 277-291, 1999.
- [3] W. Boßung, Sorin Alexander Huss, Stephan Klaus, and Lars Wehmeyer. Functional Specification of Distributed Digital Image Processing Systems by Process Interface Descriptions. Proc. of Int. Conf. on Parallel and Distributed Processing Techniques and Applications, Las Vegas, Nevada, 1999.
- [4] E.G. Cochlovius. *Spezifikation, Analyse und Simulation großer VLSI-Entwürfe mit Statecharts und Activitycharts*. PhD thesis, Naturwissenschaftliche Fakultät der Technischen Universität Carolo-Wilhelmina zu Braunschweig, 1994.
- [5] B.D. Dave and N.K. Jha. CASPER: Concurrent Hardware-Software Co-Synthesis of Hard Real-Time Aperiodic and Periodic Specifications of Embedded System Architectures. IEEE/ACM Proc. Design Automation and Test in Europe Conf., Paris, 1998.
- [6] P. Eles, K. Kuchcinski, Z. Peng, A. Doboli, and P. Pop. Scheduling of Conditional Process Graphs for the Synthesis of Embedded Systems. IEEE/ACM Proc. Design, Automation and Test in Europe, pp. 132-138, 1998.
- [7] D. Harel. Statecharts - A Visual Formalism for Complex Systems. Sci. of Comp. Progr., Aug. 1987.
- [8] P. Hsiung. Sgm: A real-time verification tool. <http://www.iis.sinica.edu.tw/~eric/sgm>.
- [9] I-Logix. Statemate MAGNUM. <http://www.ilogix.com>, 1999.
- [10] Khorol Research Inc., Khoros Core Page. <http://www.khorol.com>, 1999.
- [11] S. Klaus. I/O-Relationen in Prozeß-Spezifikationen für das HW/SW-Codesign. Project alfa core, available from <http://www.vlsi.informatik.tu-darmstadt.de/staff/bossung/publications.html>, 1998.
- [12] E.A. Lee and A. Sangiovanni-Vincentelli. A Framework for Comparing Models of Computation. IEEE Transactions on CAD, 1998.
- [13] J. K. Ousterhout. *Tcl und Tk*. Addison-Wesley, 1995.
- [14] Synopsys Inc.: SOLD - Synopsys Online Documentation. Help System for Synopsys-Tools, 1998.
- [15] SystemC. Reference Manual, Rel. 0.9. Synopsys, Inc., 1999.
- [16] L. Thiele, K. Strehl, D. Ziegenbein, R. Ernst, and J. Teich. FunState-An Internal design Representation for Codesign. Proc. International Conference on Computer-Aided Design (ICCAD '99), San Jose, November 1999.
- [17] D. Verkest, K. van Rompaey, I. Bolsens, and H. de Man. CoWare- A Design Environment for Heterogeneous Hardware/Software Systems. Design Automation for Embedded Systems, Vol. 1, No. 4, pp. 357-386, 1996.
- [18] L. Wehmeyer. Spezifikation und Analyse eines Bildverarbeitungssystems. Master thesis, available from <http://www.vlsi.informatik.tu-darmstadt.de/staff/bossung/publications.html>, 1999.
- [19] S. Wendt. *Nichtphysikalische Grundlagen der Informationstechnik, Interpretierte Formalismen*. Springer Verlag, 1991.
- [20] W.H. Wolf. An Architectural Co-Synthesis Algorithm for Distributed, Embedded Computing Systems. IEEE Trans. on VLSI Systems, vol. 5, no. 2, pp. 218-229, 1997.
- [21] D. Ziegenbein, R. Ernst, K. Richter, J. Teich, and L. Thiele. Combining Multiple Models of Computation for Scheduling and Allocation. IEEE Proc. Codes/CASHE'98, pp. 9-13, Seattle, March 1998.
- [22] D. Ziegenbein, K. Richter, R. Ernst, J. Teich, and L. Thiele. Representation of process mode correlation for scheduling. Proc. International Conference on Computer-Aided Design (ICCAD '98), San Jose, November 1998.