

On Code Generation for Rapid Prototyping Using CDIF

A. Burst, B. Spitzer, M. Wolff, K. D. Müller-Glaser

Institute for Information Processing Technology, ITIV

University of Karlsruhe

76128 Karlsruhe, GERMANY

<http://www-itiv.etec.uni-karlsruhe.de>

email: {burst,spitzer,wolff,kmg}@itiv.etec.uni-karlsruhe.de

ABSTRACT

In this paper the code generation for an open environment supporting heterogenous system design is discussed. The approach uses the CASE data interchange format CDIF as a separation layer between modeling tools and backend tools like analysis, simulation and code generation. In detail the structure of a software prototype for rapid prototyping purpose is presented including the dynamic generation of the modeling code for discrete and time-continuous domains as well as static components for the model execution. Furtheron, we present the results of several examples to get an impression of the execution time and the code size.

Keywords

Rapid prototyping, CDIF, code generation, real-time code

INTRODUCTION

The improvements in the area of microelectronics lead to shorter design cycles and rapidly rising complexity of electronic designs. Therefore, developers are forced to refine the system development process to avoid a productivity gap. Rapid prototyping is a technique to support the system development. The term rapid prototyping is defined as a type of prototyping in which emphasis is placed on developing prototypes early in the development process to permit early feedback and analysis [10].

The traditional development process can be represented with the V-model. The V-model enhanced by different levels of rapid prototyping is called VP-model (Fig. 1). The idea of rapid prototyping is to skip time intensive steps of the development process and reduce the time to get a first functional prototype of the entire system or a part of the system [3]. This prototype allows the validation of functionality and early performance tests. The comparison of different design alternatives can be simplified and therefore the conceptual validation rises. Compared to system simulation, rapid prototyping offers a test in the real environment. Hence, real-time behaviour, interferences and interfaces of the real environment must also be taken into consideration.

Three different levels can be distinguished:

OOPSLA 1998, Vancouver, Canada

Workshop #25:

*Model Engineering, Methods
and Tools Integration with CDIF*

Copyright by ACM / SIGPLAN

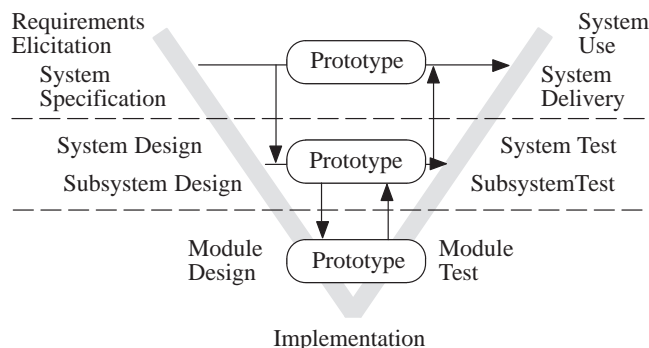


Figure 1: VP-model

- *concept-oriented rapid prototyping* represents the fast conversion of an executable specification into a functional prototype. This prototype serves mainly for clarifying system goals and must support a widespread range of hardware interfaces. It uses automatic code generation based on CASE tools like MATRIXx¹ or STATEMATE² and powerful, general purpose, extensible hardware. The cost of the rapid prototyping hardware is not critical, because the hardware is not specific and can be reused for different prototypes.
- using *architecture-oriented rapid prototyping*, the final architecture of the system is fixed. Some components of the entire system are already developed or standard components are used, while other parts yet are under development.
- *implementation-oriented rapid prototyping* is highly specialized, and the prototypes used do not permit a widespread applicability. They usually consist of a far developed prototype close to the final release or of components of already existing systems, which are supplemented by new components. At this level, an accurate performance analysis is often possible.

This paper mainly focuses on concept-oriented rapid prototyping. To design heterogenous electronic systems often CASE-tools of different design domains are needed. Hence, one CASE-tool can only be a part of the entire design process and a tight integration of different CASE-tools is required. Especially in the area of rapid prototyping different components must be connected: modeling tools, code generation, input/output configuration and general purpose input/output hardware. Because rapid prototyping is used to test a lot of differ-

¹ MATRIXx is a registered trademark of Integrated Systems Inc.

² STATEMATE is a registered trademark of i-Logix, Inc..

ent prototypes, a project management is needed to keep track of different configurations. So, we can note the following requirements for a rapid prototyping system:

- system design of different domains like state/event domain, time-continuous domain and object-oriented domain
- fine granular project management for entire system analysis, simulation and code generation
- common platform for entire system analysis, simulation [6] and code generation
- model data representation in a unique manner
- efficient code generation
- generated code should offer feedback for the system development
- easy connection of model variables with physical signals of the environment
- powerful, general purpose, extensible input/output hardware
- process visualization

In the following, we will summarize the related work in the area of rapid prototyping based on CASE-tools (section 'related work'), before we present the concept of a rapid prototyping system that can meet all requirements (section 'concept'). Afterwards, code generation based on the CASE data interchange format CDIF is discussed in particular (section 'code generation') and the structure of the code components is presented (section 'code structure'). The next section presents our results. Finally, the last section offers a conclusion on the topic.

RELATED WORK

Commercial solutions like STATEMATE [8] or MATRIXx can generate C code from a system design, modeled with statecharts or control blocks which can be used for rapid prototyping purpose [9]. The main intention of this generated code, however, is the acceleration of the system simulation. Therefore the code is not optimized for speed or size. Additionally, the code is domain specific (for example STATEMATE generates only code for a statechart representation) and only executable on specialized hardware platforms (like the AC-100 series for use with MATRIXx).

The Advanced Simulation and Control Engineering Tool (ASCET) from ETAS [5] mainly supports the time-continuous domain and only has a basic support of the state/event domain. ASCET targets transputer technology and generates target specific code. The input/output hardware of ASCET is tailored for automotive use.

In research, a table base driven approach for discrete systems is tested [12]. Here, statecharts are transformed to state transition tables, which will be sequentially executed by an interpreter. This approach uses very few memory, but the code is very slow and has strict modeling limitations. Hence this is a special solution for microcontrollers. For a rapid prototyping system which has its emphasis on the fast generation of an executable specification, this approach is not suited.

PROCORS [11], another code generation module for STATEMATE, produces optimized code which is better suited for real-time systems than the original STATEMATE code generation module.

All these approaches, however, use specific tools, data formats and hardware. Hence, all solutions do not meet the requirements. The rapid prototyping system in this paper is based on a standardized abstraction layer (CDIF), not limited to domain specific code generation and can generate efficient real-time code.

CONCEPT

Our rapid prototyping environment supports common existing CASE-tools like STATEMATE or MATRIXx. To build the structure of the entire system, an additional self-written CASE-tool (Fig. 2) is used to model the system structure with function blocks [1]. A function block can include further function blocks to refine the system structure, or a behavioral description which is represented by encapsulated CASE-tools. To couple several function blocks, the interfaces must be connected. This is done by the definition of data flows. The data of the entire model is represented with the CASE Data Interchange Format CDIF [4].

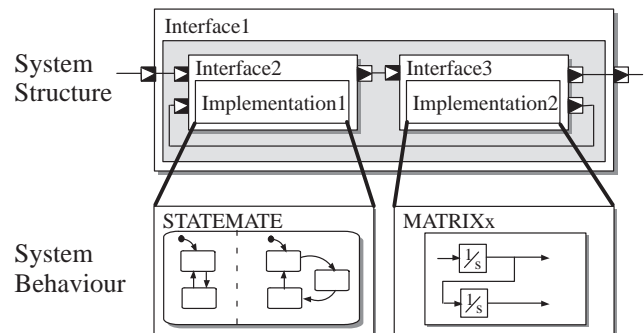


Figure 2: Modeling the system structure

The collected data of the entire model can be used for back-end tools like analysis, simulation or code generation. Hence, CDIF is used as an abstraction layer between modeling tools (front-end tools) and back-end tools (Fig. 3) [2].

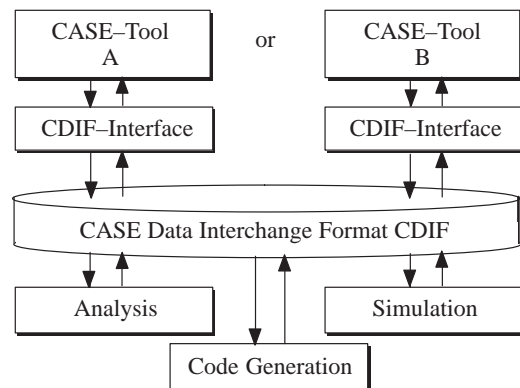


Figure 3: Data Abstraction with CDIF

For rapid prototyping purpose a code generation module translates the abstract system representation of CDIF into an executable software prototype. This prototype is executed on processor boards of backplane based systems like VMEbus

systems supplemented by powerful, general purpose, extensible hardware.

The software prototype consists of three code components:

- The *model code* is built by generating a tree representation of the model from CDIF which can be easily translated in elements and structures of a programming language e.g. C using reference code fragments. This component is responsible for the logical behaviour of the model and target independent.
- The *executer code* is a static component. It controls the timing of the software prototype and the task synchronisation.
- The *input/output code* controls the configuration of hardware components and connects model variables with physical signals of the environment.

CODE GENERATION

Starting point for the code generation is a system representation in CDIF. This representation contains all data which is necessary to describe the structure and behaviour of the system. Behavioural descriptions can be modeled with CASE-tools of different design domains and connected with data flows. However, nothing is known about the system implementation. Usually, to generate a software prototype the target platform must be known and the most fitting execution algorithms must be chosen. By separating the prototype in three components, the model code is target independent and can be generated only with knowledge of the CDIF data.

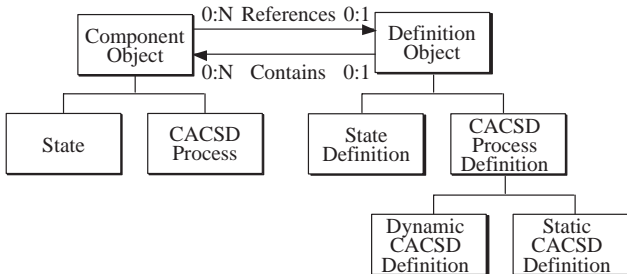


Figure 4: Part of the CDIF meta-model

As the first step in code generation, the domain of the system must be examined. An analysis of the system structure is realized by an analysis of definitions in the CDIF data (Fig. 4). Based on the meta-entity *DefinitionObject*, a discrete system is characterized by a *StateDefinition* whereas a time-continuous system is characterized by a *StaticCACSD³ProcessDefinition* or a *DynamicCACSDProcessDefinition*. If system components are coupled, a *CACSDProcessDefinition* is produced which includes all system components. In this way even hierarchical structures can be recognized and resolved. Every system component is encapsulated by a *CACSDProcess* which includes the definition of the system component. If the system component has a discrete behaviour, its attribute *IsControl* is set *TRUE*. It is possible to determine which system components are required and therefore only those code fragments are produced which are really necessary.

To determine the input/output signals, the formal ports of the definition of the *CACSDProcessDefinition* which includes all

system components are analyzed. Hence, name, type and unit of the signals can be discovered. The coupling of signals between system components is realized similar. If name, type and unit of all system components are discovered, a *Flow* and its corresponding *FlowDefinition* characterize the connection between the system components.

Generating Discrete Systems

Beginning with the highest *StateDefinition* the structure of all states is analyzed. All elements like states, transitions, actions, events or conditions which are contained in the highest *StateDefinition* will be determined. If the element is a state, its own *StateDefinition* will be determined and the analysis will continue recursively. Every state will be put with a special sorting algorithm (see section 'code structure') in a state array.

Then, for every state in the state array, a function must be generated which describes all possible transitions. The targets of all possible transitions must be examined to generate a priority list. For example, a transition can lead to a state within an orthogonal state and states in the other parts of the orthogonal state can be affected (see Fig. 11 for an example). To execute an action, every *ActionDefinition* is examined on structure and content to enable the generation of a corresponding instruction.

Generating Time-Continuous Systems

In the CDIF modeling data, *DynamicCACSDProcesses* are only represented as a state-space representation. This representation is sufficient to describe the behaviour of function blocks like numerator-denominator blocks or integrator blocks. A time-continuous system, however, can consist of several *DynamicCACSDProcesses* coupled with each other. Therefore, a state space representation of the entire system has to be build. To obtain this, a recursive algorithm examines the structure of the *DynamicCACSDProcesses*. If a process with a state space representation is arrived, the coefficients of this local state space are transformed to the global state space.

Contrary to *DynamicCACSDProcesses*, *StaticCACSDProcesses* often have trivial equivalents like the summation operator for a *SummationExpressionDefinition*.

CODE STRUCTURE

The code consists of an infinite loop executing a model step each iteration cycle. At first, the input data is updated and the coupling of model tasks is done. Next, the model tasks are executed and the output data is written (Fig. 5). These steps will be repeated periodically.

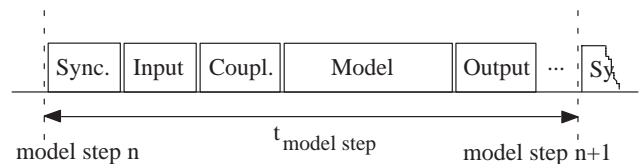


Figure 5: Structure of a model step

The synchronisation is responsible for the execution of the prototype. A prototype can be executed as fast as possible, rate monotonic or with adaptive intervals.

In the following subsections we will present the model code structure for discrete and time-continuous systems, then the

³ Computer-Aided Control System Design

executer code for the prototype synchronisation and finally the input/output code.

Model Code

Discrete Code

In every orthogonal branch of a discrete system only one state can be active within one model step. With these active states, all possible transitions to be executed are known. In our system, every state is represented by a function. These functions contain the transitions of the represented state with if-then-else constructs. Most important for the fast execution is the sequence of these constructs. The most probable conditions must be executed at first to avoid the time consuming execution of conditions.

For execution purpose, a state array is built in which states are arranged to a tree. This tree is an abstract representation of the CDIF representation. All states on one level have the same hierarchical depth. Orthogonalities are represented in different branches of the tree. Then, a one-dimensional state array is built. The first state is the top-level state which has no further states above (Fig. 6, state S0). If the state following this top-level state is a basic state (state S1) it is written in the state array. Because a basic state cannot contain additional states the examination of this branch ends and the next state which was not processed before is examined. If a hierarchical state is approached (state S2), the state will be written to the state array. After this, all successors of this state are examined in the same way (state S21 and S22). In this way the entire tree is processed. The resulting state array is presented on the right side of Fig. 7.

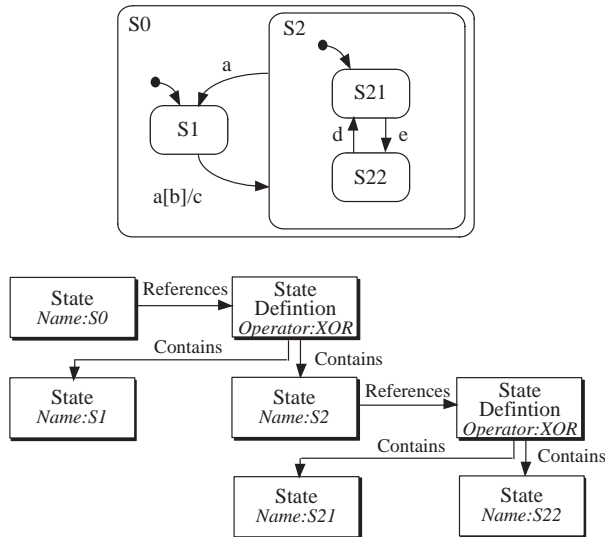


Figure 6: Hierarchical statechart example

At the bottom of Fig. 6 the CDIF representation of the statechart is shown (states only). Fig. 7 presents on the left hand the generated tree. The similarity between the CDIF tree and the generated tree are obvious. The state array stores for every state a pointer to its corresponding function and three bits (a part of the integer value StateArray[stateNo].flags) which indicates if the state was, is or will be active. If a basic state will be set active, all hierarchical states above this state in its

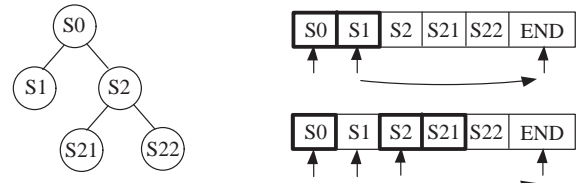


Figure 7: Generation of the state array

branch will be set active. The top-level state is active by default.

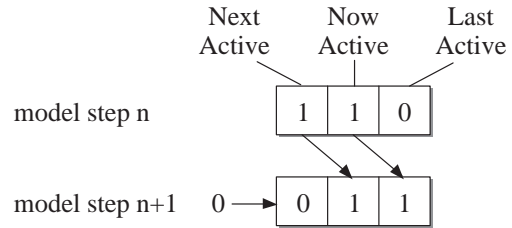


Figure 8: Shift operation of StateArray[stateNo].flags

Before a model step is executed, three bits of every state of the state array are shifted. A state which was set to *NextActive* in the last model step is set *NowActive*. After this operation, all *NextActive* flags will have the value '0'.

The main execution loop of the model code consists of a for loop which examine the flag of every state whether *NowActive* is set. If *NowActive* is set, the function of the state is executed.

1. for (loop=1; loop<MaxNoOfStates; loop++) {
2. if (StateArray[loop].flags & NowActive)
3. (*StateArray[loop].state) ();
4. }
5. StepCount++;

Figure 9: Main execution loop

If a transition is fired during the execution of a state function (the state function is invoked in Fig. 9, line 3), the corresponding actions are executed and the next active states are determined by setting the *NextActive* bit. The C code fragment for state1 has the following structure:

1. void S1 (void) {
2. if ((a == StepCount) && (b == TRUE)) {
3. StateArray[S21].flags |= NextActive;
4. StateArray[S2].flags |= NextActive;
5. c = StepCount + 1;
6. loop += 3;
7. }
8. }

Figure 10: C code fragment for state S1

The function of state S1 starts with the examination whether the event *a* occurs and the condition *b* is true (Fig. 10, line 2). The value of the current model step is stored in the variable StepCount. An event occurs if the variable *a* was set in the model steps before to the value of the step the event shall occur. In our example the variable *a* must have been set to the current value StepCount. A condition is represented with boolean values. If both of the triggers are true, the next active basic state will be state S21 (line 3). Additionally to state S21

the hierarchical state above, state S2, is also set to active in the next model step (line 4). The variable *c* is set to 'StepCount + 1' (line5). Because the value of the variable StepCount is increased next model step (Fig. 9, line 5), the event *c* will be detected in the next model step.

The execution of the model step is shown in Fig. 7 on the upper right hand. State S1 is highlighted because it is the only active state (*NowActive* is set). During the execution process, all states are examined whether their *NowActive* flag is set. State S1 is detected as active and executed. All other states of this hierarchical level and below (except for orthogonal branches of the same hierarchical level) can be skipped because only one state can be *NowActive* in state S0 (Fig. 10, line 6). Hence, the execution process finishes at END. After the model step (lower right hand) the states S2 and S21 are highlighted.

The execution process can also be enhanced if hierarchical states are left. If the execution of a transition to a higher level is fired, the transitions of all states below must not be examined. Because of that, the loop pointer for the execution of the state array is increased for the number of all states below. By doing that the current branch will not be examined any longer and next orthogonal branch is entered. As a result a faster execution is obtained. To show this in Fig. 6 we have to fire an event *a* (assumed we are still in state S21). The loop pointer will at first execute state S0 without any transitions. State S1 will not be executed because its flag is not set to *NowActive* (Fig. 7 lower right hand). State S2 is the first state whose corresponding function will be executed. Because the event *a* was fired the transition to state S1 will be entered. Then, the loop variable is increased by two because of the number of states inside state S2. Because every model step the variable loop is increased by one (Fig. 9, line 1), the loop pointer is increased by three. Therefore the states S21 and S22 are not executed and the loop ends.

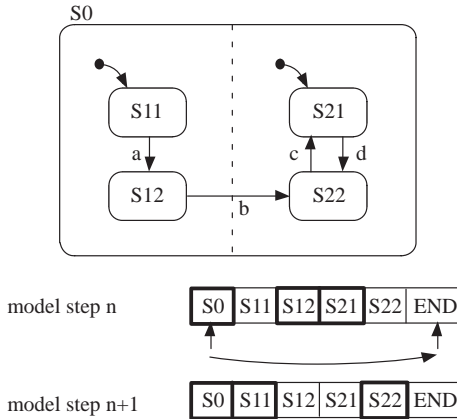


Figure 11: Orthogonal Statechart example

Orthogonal states can also be executed efficiently. Fig. 11 shows a statechart which consists of two parts executed together. Assumed states S12 and S21 are active and the event *b* was fired in the last model step, the transition from state S12 to state S22 has to be performed. The result will be an active state S22 from the fired transition and an active state S11 because the control flow of the left side was generated again with the

default transition. Our code generation module will not execute the transition from state S12 to state S22 when executing the function of state S12 but when executing the function of state S0, because the transition has a priority of a transition leaving state S0. The C code fragment for state S0 has the following structure:

```

1. void state0 (void) {
2.     if (b == StepCount) {
3.         StateArray[state22].flags |= NextActive;
4.         StateArray[state11].flags |= NextActive;
5.         loop += 4;
6.     }
7. }

```

Figure 12: C code fragment for state0

All states inside state S0 are skipped (Fig. 11, lower hand) and the execution ends.

Time-Continuous Code

For the execution of the dynamic time-continuous domain, all numerator-denominator or integrator components are transformed to a common state space representation. If a rational transfer element [7] is given

$$Y(s) = G(s) U(s) \quad \text{with}$$

$$G(s) = \frac{Z(s)}{N(s)} = \frac{b_0 + b_1s + \dots + b_ns^n}{a_0 + a_1s + \dots + a_ns^n}$$

the corresponding derivation function is:

$$a_n y^{(n)} + \dots a_1 \dot{y} + a_0 y = b_0 u + b_1 \dot{u} + \dots + b_n u^{(n)}$$

To prepare the transformation the equation is transformed to:

$$Y(s) = b_0 \frac{1}{N(s)} U(s) + b_1 \frac{s}{N(s)} U(s) + \dots + b_n \frac{s^n}{N(s)} U(s)$$

State variables X_v are defined to:

$$X_1 = \frac{1}{N(s)} U(s), X_2 = \frac{s}{N(s)} U(s), \dots, X_n = \frac{s^{n-1}}{N(s)} U(s)$$

They have the following easy relations in frequency domain:

$$sX_1 = X_2, sX_2 = X_3, \dots, sX_{n-1} = X_n$$

If we assume that the starting values are zero, we get the following representations in time domain:

$$\dot{x}_1 = x_2, \dot{x}_2 = x_3, \dots, \dot{x}_{n-1} = x_n \quad \text{and therefore}$$

$$\dot{x}_1 = x_2, \ddot{x}_1 = x_3, \dots, x_1^{(n-1)} = x_n$$

Together with the equation of X_1 we get:

$$\dot{x}_n = -\frac{a_0}{a_n} x_1 - \frac{a_1}{a_n} x_2 - \dots - \frac{a_{n-1}}{a_n} x_n + \frac{1}{a_n} u$$

This representation can be transformed into a matrix representation which is called *state equation*.

$$\dot{\underline{x}} = \begin{bmatrix} 0 & 1 & 0 & \vdots & 0 \\ 0 & 0 & 1 & \vdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \vdots & 1 \\ -\frac{a_0}{a_n} & -\frac{a_1}{a_n} & -\frac{a_2}{a_n} & \vdots & -\frac{a_{n-1}}{a_n} \end{bmatrix} \underline{x} + \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ \frac{1}{a_n} \end{bmatrix} u$$

The *output equation* is received by transforming the equation of $Y(s)$ in time domain and introducing state variables:

$$y = b_0x_1 + b_1x_2 + \dots + b_{n-1}x_n + b_n\dot{x}_n$$

$$y = \left[b_0 - a_0 \frac{b_n}{a_n}, \dots, b_{n-1} - a_{n-1} \frac{b_n}{a_n} \right] \underline{x} + \frac{b_n}{a_n} u$$

The short notation of the state and output equations looks like:

$$\begin{aligned} \dot{\underline{x}} &= \underline{A} \underline{x} + \underline{B} u \\ y &= \underline{C} \underline{x} + \underline{D} u \end{aligned}$$

This state space representation has the same behaviour as the first equation and has to be solved. Several algorithms to solve this state space equation exist and the user has to choose an appropriate integration algorithm. If a high precision is wanted, the speed is low and vice versa. For real-time execution, only lock-step based approaches like Euler or Runge-Kutta are suited. Every model step mainly consists of one or more integration steps and the update of internal and external variables. The model code of different models differs only in values for state and output equations and in the state space dimension. The state space dimension has the most important influence on the duration of the execution.

The Runge-Kutta algorithm which is best suited for a fast and high precision execution is presented in Fig. 13.

1. CalculateDerivationFunctionsOfX()
2. for each dimension do
3. **let** SavedIntegrationValue = IntegrationValue
4. **let** k1 = StateEquation
5. **let** IntegrationValue = SavedIntegrationValue + k1 * ModelStep
6. **end for**
7. CalculateStateAndOutputEquations()
8. CalculateDerivationFunctionsOfX()
9. for each dimension do
10. **let** k2 = IntegrationValue
11. **let** IntegrationValue = SavedIntegrationValue + 0,5 * (k1 + k2) * ModelStep
12. **end for**
13. CalculateStateAndOutputEquations()

Figure 13: Runge-Kutta algorithm

The algorithm starts with the calculation of the derivation function in t . *SavedIntegrationValue* represents the value of the last integration step. The variables k_1 (line 4) and k_2 (line 9) serves as a temporary storage of the derivation function of x to the times t and $t + \text{ModelStep}$. The for loop (lines 2 - 6) represents a first approximation. With this approximation the state and output equations are computed approximately and the step is calculated once again to obtain a better solution (lines 9 - 12).

Executer

When the software prototype is executed, timing issues are of great importance. There are several possibilities to execute a software prototype. The prototype can work as fast as possible and executes the step after the other without delay. With this, however, it is impossible to guarantee an execution time. Additionally, an absolute timing is needed to execute time-continuous components like a pulse wave block. Only event based systems can be executed this way. Another possibility is

a rate monotonic execution with an auxiliary timer. The execution then is processed during equal time intervals. Without additional effort an absolute timing can be obtained and code of all domains can be processed. The time intervals must be determined before the execution starts. If the execution of a prototype needs less time than the chosen interval, the synchronization block of Fig. 5 waits until the interval finishes. Then, the next model step is executed. The timing resolution is the limiting fact for the execution of the entire prototype. Therefore, it is most important to optimize this prototype component.

The executer as a static component is responsible for the timing of the execution. If we execute as fast as possible, the executer is very small and mainly consists of an infinite while loop to determine the absolute time. If the execution is done rate monotonic, we obtain two code components. The first component (Fig. 14) serves for the execution of the model step and model data updates. At first, the auxiliary timer is set to the model step time (line 4). The auxiliary timer as an interrupt service routine (ISR) periodically calls the function AuxTimer (line 5) which is represented in Fig. 15. When the auxiliary timer is started (line 6), the while loop does nothing because the conditions of both if statements are not true (line 8, line 16). When the function AuxTimer is started, CheckSemaphore is set to '0' (Fig. 15, line 9) and the execution of the first if statement begins. CheckSemaphore and ModelSemaphore are set to '1' to indicate, that the execution of the model has started. The input and coupling data is updated (line 11) and the model step begins (line 12). After the execution of the model code, the output data is updated (line 13). If the execution is interrupted from the ISR while executing the model, the ModelSemaphore is still set to '1'. If AuxTimer is executed, the auxiliary timer is disabled and the variable ExecutionError is set to '1' (Fig. 15, line 5).

1. **let** CheckSemaphore = 1
2. **let** ModelSemaphore = 0
3. **let** ExecutionError = 0
4. sysAuxCikRateSet(ModelStepTime)
5. sysAuxCikConnect(AuxTimer)
6. sysAuxCikEnable()
7. while true do
8. **if** CheckSemaphore == 0 **then**
9. **let** CheckSemaphore = 1
10. **let** ModelSemaphore = 1
11. UpdateInputsCoupling()
12. ModelStep()
13. UpdateOutputs()
14. **let** ModelSemaphore = 0
15. **else**
16. **if** ExecutionError == 1 **then**
17. break
18. **end if**
19. **end if**
20. **end while**

Figure 14: Executer algorithm

If CheckSemaphore is set to '0' when entering the AuxTimer function, the assignment within the while loop was not exe-

cutted and an execution error is detected. Then, the executer is also finished (Fig. 14, line 16).

```

1. if CheckSemaphore == 0 then
2.   sysAuxClkDisable()
3.   let ExecutionError = 1
4. else
5.   if ModelSemaphore == 1 then
6.     sysAuxClkDisable()
7.     let ExecutionError = 1
8.   else
9.     let CheckSemaphore = 0
10.  end if
11. end if

```

Figure 15: Auxiliary Timer algorithm

With these algorithms, a proper timing is obtained. Instead of finishing the model execution in case of on execution error an adaption of the chosen equal time intervals can be performed.

Input/Output Code

A graphical user interface was developed to perform the connection of model variables with physical signals of the environment. The code is mainly static for every hardware board and only the current variables are inserted. The code includes the basic initialization routines of the hardware boards and updates the input/output data within every model step.

RESULTS

Our CDIF based rapid prototyping system has been used for several projects and has proven as a well suited basis for developing electronic systems. Important for a rapid prototyping system is the fast execution of the prototype. So we tested the performance of our code generation with a FIR (finite impulse response) filter up to 50th dimension and with an array of sequential arbiters both with the execution mode 'as fast as possible' (AFAP) and rate monotonic with an auxiliary timer (RM). The FIR filter consists of one summation, one multiplication and one state space. The sequential arbiter consists of 5 states and 7 transitions per cell. The cells are orthogonally executed. For our tests we used a Motorola VMEbus based system with a 200 MHz PowerProcessor PPC 604, the real-time operating system VxWorks and self-developed general purpose, extensible and programmable hardware modules. Fig. 16 shows the results of our tests. Even for these huge models, the execution times are below 100 μ s. The overhead of the rate monotonic execution with an auxiliary timer is small. To get a determined real-time behaviour the rate monotonic execution should be used.

Fig. 17 shows the code size of the examples (complete executables). The FIR code size mainly remains the same whereas the arbiter code size is rising with the number of the sequential arbiter array. This is obvious if we take a look at the code. The FIR code only gets greater state space dimensions while the arbiter code is enlarged by more state functions with additional if-then-else constructs.

With execution times in μ s area, the rapid prototyping system can execute for example automotive applications. We tested this with a power window regulator. The code, which includes pinch protection, is executed in 90 μ s and has a size of 45 kByte. Please note, that this code is not suited for mass pro-

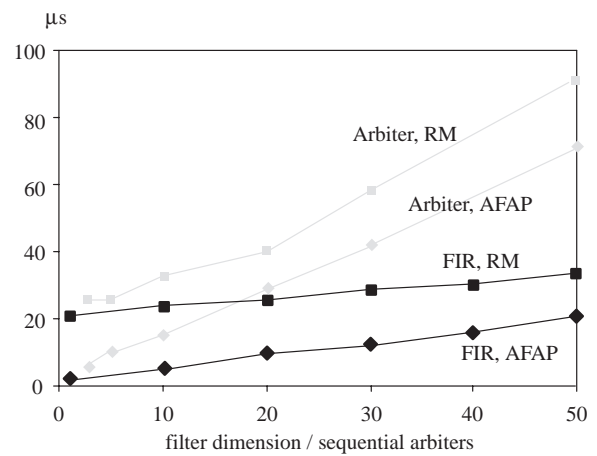


Figure 16: Execution times of FIR filter and sequential arbiter code

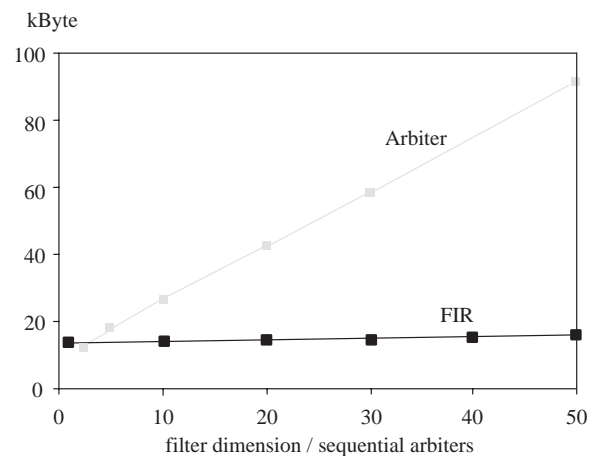


Figure 17: Size of FIR filter and sequential arbiter code

duction. As shown before, rapid prototyping supports the conceptual validation in early design cycles. However, the faster the prototype can be executed, the more complex applications can be supported. Even applications in the high speed area of engine control can be supported in future.

CONCLUSIONS

The presented rapid prototyping system is the first system of its kind supporting the CASE Data Interchange Format CDIF. Especially for heterogenous system design the advantages of using such a separation layer become obvious. The user can choose the best suited tools for his design, because of the tool independence of CDIF. It has been shown that the code generation from CDIF is able to support a wide area of applications.

Our rapid prototyping system consists of three code components: model code, executer code and input/output code. Because of the modular structure single components can be modified easily. Additionally, an efficient code generation is offered. The most important component for fast execution times is the executer code. We presented an optimized structure of the executer. The basic structures of the model code for discrete and time-continuous domains are given.

Future work will examine further code optimization. Additionally, tests in time critical applications like engine manage-

ment are necessary to discover the limits of a rapid prototyping effort.

ACKNOWLEDGMENTS

We thank WindRiver Systems, Inc. and the Daimler-Benz AG for their generous support of our work.

REFERENCES

1. Burst, A.; Wolff, M.; Kühl, M.; Müller-Glaser, K.D.: *A Rapid Prototyping Environment for the Concurrent Development of Mechatronic Systems*. European Concurrent Engineering Conference, Erlangen, Germany, 1998.
2. Burst, A.; Wolff, M.; Kühl, M.; Müller-Glaser, K.D.: *Using CDIF for Concept-Oriented Rapid Prototyping of Electronic Systems*. Rapid System Prototyping, Leuven, Belgium, 1998.
3. Dollas, A.; Kanopoulos, N.: *Reducing the Time to Market Through Rapid Prototyping*. IEEE Computer, February, 1995.
4. EIA/CDIF Technical Committee: *CDIF / CASE Data Interchange Format*. EIA Interim Std. EIA/IS- 106 -112, 1994.
5. Eppinger, A.: *Computer-Integrated System Technology with ASCET*. Ph.D. thesis, University of Paderborn, 1993.
6. Ernst, J.: *An Open Simulation Architecture for the Development of Complex Embedded Systems Using Distributed Objects*. Proc. SAE, Detroit, 1997.
7. Föllinger, O.: *Control System Design* (in german). Hüthig, 1985.
8. Harel, D.: *Statecharts: A Visual Formalism for Complex Systems*. Science of Computer Programming, 1987.
9. Harel, D.; Gery, E.: *Executable Object Modeling with Statecharts*. IEEE Computer, July, 1997.
10. Kurpis, G.; Booth, C.: *The New IEEE Standard Dictionary of Electrical and Electronics Terms*. New York, 1993.
11. Spreng, M.: *Rapid Prototyping for Automotive System Development*. Proc. SAE, Detroit, 1995.
12. Wietzke, J; Cochlovius, E.: *Generating Small Controller Code from Statecharts* (in german). Elektronik, 20,1996.