FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Model-Based Development of Distributed Embedded Systems by the Example of the Scicos/SynDEx Framework

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

BERNHARD FISCHER
bernhard.fischer@gmx.at
Matrikelnummer 0126057

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuer:
PRIVATDOZ. DI DR. WILFRIED ELMENREICH

Wien, 12.09.2010

(Unterschrift Bernhard Fischer) (Unterschrift Betreuer)

Kurzfassung

Die Erstellung eingebetteter Systeme wird mit erhöhten funktionellen Anforderungen, der raschen Weiterentwicklung von Komponenten, und verkürzten Lieferzeiten konfrontiert. Entwicklungsmethoden, die sowohl eine schnelle Anpassung an Veränderungen und fehlerfreies Design, als auch die Minimierung von Kosten und die Möglichkeit einer zeitgerechten Produktauslieferung bieten, werden benötigt. Eine Antwort auf erhöhte Anforderungen und Komplexität sind Entwicklungsmethoden basierend auf Modellen. Die modellbasierte Entwicklung trennt fachliche und technische Belange auf verschiedenen Ebenen, und unterstützt dabei die konzeptuelle Erfassung von Systemen, automatisierte Codegenerierung, formale Verifikation, und Zertifizierung.

Diese Arbeit demonstriert den modellbasierten Entwurf von eingebetteten Systemen mit dem Scicos/SynDEx Framework. Scicos ist ein Modellierungs- und Simulationswerkzeug für hybride Systeme. SynDEx ist eine integrierte Entwicklungsumgebung für die Prototypenerstellung verteilter Systeme. Die Umsetzung des modellbasierten Ansatzes wird mit zwei konkreten Beispielen aus dem Bereich der Regelungstechnik auf einer Entwicklungsplattform bestehend aus vernetzten Mikrokontrollern, Sensoren, und Aktuatoren durchgeführt. Resultierende Ergebnisse helfen für mittelgroße eingebettete Systeme Vor- und Nachteile des modellbasierten Ansatzes gegenüber handgeschriebenen Ansätzen abzuwiegen.

Einprozessoranwendungen zeigten einen erhöhten dynamischen und statischen Speicherverbrauch im Vergleich zu handgeschriebenen Ansätzen – dagegen steht der von Werkzeugen gebotene Komfort durch Simulation, Verifikation und automatischer Codegenerierung. Der Einsatz des Frameworks ist für einzelne Applikationen unökonomisch, dies wurde durch den nötigen Aufwand für die Erstellung eines nahtlosen Modellierungs-Frameworks und den erhöhten Entwicklungsaufwand bezeugt. Ein weiterer Nachteil bestand darin, dass der mittels SynDEx generierte Code nicht an Mikrokontroller angepasst werden konnte ohne das Zeitverhalten der Applikation signifikant zu verändern. Das Scicos/SynDEx Framework bietet vielversprechende Ansätze, da jedoch noch viele Verbesserungen ausstehen, wird es derzeit nur für experimentelle Zwecke empfohlen.

Abstract

The embedded systems engineering industry faces increasing demands for more functionality, rapidly evolving components, and shrinking schedules. Abilities to quickly adapt to changes, develop products with safe design, minimize project costs, and deliver timely are needed. A response to the broader range of requirements and the problems brought along with system complexity are development methods based on models. Model-based development (MBD) follows a separation of concerns by abstracting systems with an appropriate intensity, such as the separation of functional requirements and specification from implementation details. MBD promises higher comprehension by modeling on several abstraction-levels, formal verification, automated code generation, and certification.

This thesis demonstrates MBD with the Scicos/SynDEx framework on a distributed embedded system. Scicos is a modeling and simulation environment for hybrid systems. SynDEx is a rapid prototyping integrated development environment for distributed systems. Performed examples implement well-known control algorithms on a target system containing several networked microcontrollers, sensors, and actuators. Results of these demonstrations support the decision-making process of either preferring MBD or classical, hand-written approaches. The addressed research question tackles the feasibility of MBD for medium-sized embedded systems.

In the case of single-processor applications experiments show that the comforts of tool-provided simulation, verification, and code-generation have to be weighed against an additional memory consumption in dynamic and static memory compared to a hand-written approach. Expenses for establishing a near-seamless modeling-framework with Scicos/SynDEx and an increased development effort indicate a high price for developing single applications, but might pay off for product families. A further drawback was that the distributed code generated with SynDEx could not be adapted to microcontrollers without a significant alteration of the scheduling tables. The Scicos/SynDEx framework forms a valuable tool set that, however, still needs many improvements. Therefore, its usage is only recommended for experimental purposes.

Contents

List of Figures

List of Tables

1 Introduction

Embedded systems engineering combines the fields of software, hardware and control engineering. The main characteristic which separates embedded development from plain software development is its intrinsic link to the physical world, often in a safety-critical environment. Embedded systems usually realize a set of requirements for precise control of electromechanical devices (e.g. MEMs[1]).

Advancements in microfabrication lead to cheaper and more powerful embedded components which come along with higher customer demands and tighter schedules. The increasing number of system functionalities implies higher complexity and diminished perceptibility especially when it comes to simultaneity: Systems that might appear as being simple at a first glance are in fact hard combinatorial problems, the limits of the conceptual landscape in the human mind are easily reached: the number of states in an automaton explodes with every additional degree of freedom.

Current development methods for embedded systems incorporate plan-driven and agile characteristics to a certain degree. Document-centric plan-driven methods provide less flexibility to requirement and specification changes than agile methods which focus on interaction with customers and adaptability. Development methods are responsible for the successes of IT-projects of which about 20% failed according to studies between 2002 and 2006 [EK08]. Main reasons include over-budget, too many scope changes and the management being not sufficiently involved. These reasons are based on two common denominators, namely uncertainty and risk. At project start the possible solution space is blurred, estimations about costs and project time are vague.

A response to all these issues are model-based development (MBD) paradigms. In MBD concerns are separated by their importance for the current activity in the development process, such as the separation of implementation details from requirements specifications. Centering the development process around models allows a higher flexibility to change requests by automation, reduces risks by enabling simulation and verification in early project phases, and supports the management in planning project costs and time. Low costs, timely delivery and safe design are the driving forces of systems engineering.

[1] Micro-Electro-Mechanical Systems

Introducing models has a price and many aspects have to be considered when shifting to a model-based design culture [SPF07]. Educating stakeholders with new modeling techniques and tools might be expensive, requires time, and adopting already established processes or legacy systems is costly. These investments increase the development effort in the scope of short-term planning. Embedded systems are often restricted by computational power, memory and disk space. Are model-based solutions efficient? This thesis prepares information that helps to weigh costs against gain in the context of MBD.

Models act as a base for consistent, automatic artifact generation. Implementation (code generation) and verification can be carried out with trusted, standardized tools which are usually based on synchronous languages and are current research topics in computer science. Mathworks' MATLAB and Esterel Technologies' SCADE are well-established, commercial tool-sets on the market and specialized inside the domain of safety-critical embedded systems. An overall, generally applicable modeling theory does not exist, but modeling techniques and scientific tools provide research opportunities, such as the non-commercial Scicos/SynDEx framework. This framework combines hybrid systems modeling with Scicos and discrete temporal, distributed modeling, optimization and code-generation with SynDEx.

This thesis evaluated the benefits and costs of a MBD-based development process with the Scicos/SynDEx framework on a distributed target-platform. Capabilities of the framework were explored and MBD compared to classic, hand-written development. Development effort, executable code size, code structure (perceptibility), and memory consumption were compared. In literature there are no such examples of using Scicos/SynDEx on an distributed architecture with microcontrollers.

Objectives of this work are to exercise and analyze the outcomes of well-known control and data observation algorithms following a model-based development process. Examples are PID control algorithms, and a data observation application carried out on a scientific target-platform consisting of several microcontrollers and peripherals including sensors and actuators.

Algorithms in the discrete domain are modeled together with continuous environment peripherals. The demonstrations include a single- and a multi-processor example. In the single-processor application a PID algorithm controlling a cooling fan is realized. In the multi-processor example, temperature data is observed at one node, displayed on an LCD on another, and forwarded to the development workstation via an embedded gateway node.

Implemented applications are hybrid systems which are designed, simulated and verified with Scicos. Temporal design, scheduling, optimization, distribution and code-generation of transformed Scicos models are performed with the help of SynDEx. A seamless modeling environment is approached by adapting the Scicos/SynDEx tool-chain.

The research in this thesis is limited to a multi-processor target-platform and the Scicos/SynDEx framework. Several other tools require research, but the results in this thesis can be used to compare this framework's performance to tool-sets incorporating other modeling techniques. The research shows strengths and weaknesses of the framework with respects to scheduling, modeled versus real behavior, code metrics and resource allocation. Estimations for the development effort of MBD and classic development are not quantitative, they are only based on a few examples.

Results of this thesis demonstrate how control algorithms can be modeled with this particular model-based development toolset. Problems and pressing issues were identified and contribute information for enhancing the Scicos/SynDEx framework. The data gained by comparing classical development to MBD eases the decision of migrating model-based paradigms into the development process. With model-based development the possible solution space is narrowed down in early stages by rapid prototyping and design faults are detected easier. However, all this comes with the price of extra development effort, program size and memory consumption.

1.1 Outline

The thesis is divided into **five chapters**: ***(1)*** *Introduction*, ***(2)*** *Concepts and Related Work*, ***(3)*** *Modeling Tools*, ***(4)*** *Demonstrations: MBD with Scicos/SynDEx*, and ***(5)*** *Conclusion*.

Chapter 1 introduces the background, scope of research, applied methods, the objectives and relevance of this thesis.

Chapter 2 encapsulates important terms and definitions. Control theory relevant for the examples is introduced briefly. Basic concepts in modeling systems are presented, including model-based and model-driven development and approaches.

Chapter 3 explains the modeling tools Scicos and SynDEx in a detail necessary for understanding the examples. Syntax, semantics, underlying modeling techniques, and modeling with the used tools are surveyed. Strengths and weaknesses of temporal design with SynDEx is given special consideration to.

Chapter 4 shows the methods and realized experimental designs. The design of PID algorithms with Scicos and modeling hardware, communication, and software architectures with SynDEx is presented and discussed. Finally, this chapter summarizes the examples' results.

Chapter 5 reviews the results from the previous chapter. Advantages and drawbacks of systems design by modeling are discussed based on development effort and designs. Improvements are proposed for the SynDEx tool, as well as concepts for establishing a working modeling tool-chain.

2 Concepts and Related Work

Model-based and model-driven development have been emerging terms in embedded systems development. Demonstrations in this thesis require the definition of concepts in control theory, modeling theory, and development methodologies. The following definitions resulted from a literature study to build a conceptual base for this thesis. Since the topic of this thesis derives from different disciplines such as computer science, control engineering, and embedded systems, the existing nomenclatures are often not clearly defined in literature, have ambiguous meaning, are used freely, and often depend strongly on their context.

2.1 Basic Concepts in Systems Engineering

This section defines general methods and concepts needed in the following chapters of this thesis. The model-based development demonstrations require definitions in the fields of software, systems, models and control theory.

Software Engineering

First defined in a NATO[1] conference in 1968 [NR68], *Software Engineering* has been a term not easy to describe in a way that satisfies the whole IT - community. Even though there is no complete consensus, the IEEE[2] offers a standardized definition:

> "(1) The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software. (2) The study of approaches as in (1)." [IEE90]

System

> "A collection of components organized to accomplish a specific function or set of functions." [IEE00b]

[1] North Atlantic Treaty Organization
[2] Institute of Electrical and Electronics Engineers, Inc.

Distributed System

In a *distributed system* the components are distributed in space and are contributing functionality to the whole system. A similar definition by IEEE:

> "A computer system in which several inter-connected computers share the computing tasks assigned to the system." [IEE00a]

Reactive System

A *reactive system* is a system which reacts to stimuli, outputs are set dependent on the input and the system function. Such a system is in a continuous interaction with its environment.

Hybrid System

The combination of a system operating at discrete time instants and a corresponding environment located in a continuous time domain, can be called a *hybrid system*.

Systems Engineering

> "Systems engineering is an interdisciplinary engineering management process that evolves and verifies an integrated, life-cycle balanced set of system solutions that satisfy customer needs." [Col01]

Architecture

> "The fundamental organization of a system embodied in its components, their relationships to each other, and to the environment, and the principles guiding its design and evolution." [IEE00b]

Life Cycle Model

> "A framework containing the processes, activities, and tasks involved in the development, operation, and maintenance of a software product, which spans the life of the system from the definition of its requirements to the termination of its use." [IEE00b]

Software Process

A set of activities and its following results building a software product is called a *Software Process* (sometimes also called a Software Life Cycle or Software Development Process) which is an instance of a Software Development Methodology. Today, almost every software process contains following four basic process activities [Som04]:

- **Software Specification** - Customer and developer define how the software is developed within specified constraints.
- **Software Development** - Design and development of the software.
- **Software Validation** - Software test and check against the requirements/specification.
- **Software Evolution** - Adaption of the software to fulfill new customer requirements.

Development Process

This term is often set in relation to a software process, but a brief stand-alone definition looks as follows. A *Development Process* consists of ...

Phases and corresponding activities.

Activities carried out by developers.

Artifacts as results of activities.

Software Process Models

A *Software Process Model* presents a simplified, abstracted view of a Software Process. Those models can contain activities as part of the software process, roles of persons, products and scheduling. Most of them build on following universal paradigms [Som04]:

- **Waterfall Model.** In this paradigm each software process activity is seen as its own development phase with an strictly defined start and end of the phase. The software process proceeds with a close of the preceding phase to the next phase. The concept of the Waterfall Model was initially created by Winston W. Royce in 1970. [Roy70]. This paradigm should be used if the requirements are fixed and no major changes are foreseen until the end of the project, otherwise, due to the inflexible partition of the phases, the development effort will rise tremendously.
- **Evolutionary Development.** Specification, Development and Validation are proceeding parallel and with close cooperation with the customers. Information gained in each activity is shared to each other. The practice of building prototypes can help to get more knowledge about the customers' requirements.
  This paradigm is good for a stepwise refinement of the specification but in the view of the management it is hard to measure the advancement of the project since it is not cost-effective to document each intermediate product. Another possible drawback is a bad structure of the system

architecture due to the ongoing refining of the specification, which makes this paradigm not well-suited for very large projects ($> 500\,000$ LOC[3]).

- **Component Based Software Engineering.** This approach is focused on reusing and adapting software components.

Software Life Cycle

> "The period of time that begins when a software product is conceived and ends when the software is no longer available for use. The life cycle typically includes a concept phase, requirements phase, design phase, implementation phase, test phase, installation and checkout phase, operation and maintenance phase, and sometimes, retirement phase. These phases may overlap or be performed iteratively, depending on the software development approach used." [IEE90]

Process Iterations

A way to handle requests for changes in the software product is to repeat the software process completely or partially. A *Process Iteration* is either a repetition of the whole software process or single/several process activities. If an activity is repeated, all successors have also to be handled to avoid inconsistencies between specification, design, code and documentation.

- **Incremental Development**
  Incremental Development is a combination of the Waterfall Model and Evolutionary Development: The specification and the design of the system and subsystems are defined once followed by the subsequent development, integration and validation of subsystems. Subsystems are delivered stepwise to the customers, starting with the most important functionalities.
- **Spiral Model**
  Process activities are visualized using a spiral. Each loop of the spiral describes a process phase. Starting in the inner phase with a feasibility study, ensued by the definition of the specification, design, development etc. One main feature of this paradigm is a risk analysis embedded in each loop providing valuable information for the project management. [Boe88]

Rational Unified Process (RUP)

The *Rational Unified Process* is a phase-oriented model spawned of the Unified

[3] Lines of Code

Software Development Process . RUP applies UML[4] notations and combines parts of several software process models. In contrast to the other mentioned software processes, RUP supports three different views to the Software Process:

- a *dynamic perspective* showing all model phases timely ordered.
- a *static perspective* visualizing process activities.
- a perspective proposing further procedures for the process.

Software Development Methodology
In literature the term *Software Development Methodology* has ambiguous meanings and strongly depends on the context it is used in (e.g. some authors identify a methodology with a process). A software development methodology is a framework for solving a technical challenge. In a general point of view a software development methodology is an orchestration of:

- Software process (software process model) which provides fragmentation and schedule of the software development in several phases.
- Notations and techniques (e.g. object-oriented, prototyping) which are established for the documentation of products and intermediate products.
- Management methods for the analysis and transformation of documents.

2.1.1 A Brief Glimpse on Control Theory

Every technical process intriniscally comprises reaction delays and disturbances. Imagine a fan-cooling system of a computer-chip has to provide the correct amount of air-flow to ensure a certain operation-temperature that does not damage the chip. Just setting a reference speed will usually not result in the fan moving as desired due to disturbances. Reasons for this are characteristics of physical processes involved such as friction, acceleration time, inertia and many more. Not all fans will behave the same way because of production tolerances, and in the case of digital systems the references provided to the fan are of a discrete nature which is per se a reason for diverging behavior. Such dynamical systems require an automated control component that keeps it in a desired state such as following a pre-defined function. *Control Theory* deals with mathematical approaches for the control of a system.

A basic control structure is set up by a *controller* P providing *inputs* $u(t)$ to the *system under control (process)* P that is interfered with by $d(t)$. The

[4]Unified Modeling Language

output $y(t)$ of the system is fed back and the *error* $e(t)$ between *reference value* $r(t)$ and $y(t)$ calculated (figure 2.1.1).

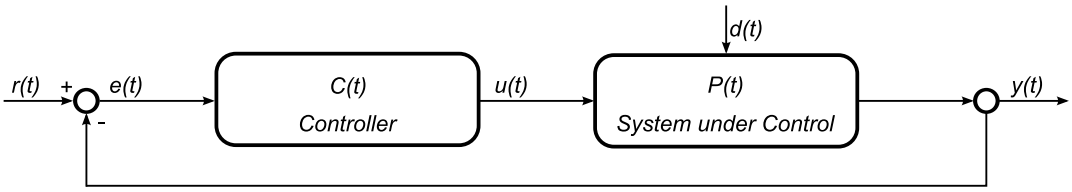


Figure 2.1: Single, closed-loop control structure.

The behavior of the process $P(t)$ is determined with standardized test-functions and characterized by the resulting *step responses*. This technique is often used to gain an approximate description of the process instead a complete mathematical model. A widely used control algorithm $C(t)$ is the PID (Proportional Integral Derivative) controller. Controllers are set-up and tuned with the goals of appropriate rise time, minimal overshoot and no steady-state errors. PID algorithms are tuned with several approaches such as the Ziegler-Nichols, Cohen-Coon, or the Chien-Hrones-Reswick formula. Fine-tuning of the algorithm might be required.

PID Algorithms

The output of a PID algorithm is composed by a proportional K_P, integral K_I and derivative K_D part, with the integral time T_n and the derivative time T_v in the *standard form* :

$$u(t) = K_P * \left(e(t) + \frac{1}{T_n} \int_0^t e(\tau) d\tau + T_v \frac{\mathrm{d}e(t)}{\mathrm{d}t} \right)$$

. . . which is equivalent to the *ideal parallel form*:

$$u(t) = K_P\, e(t) + K_I \int_0^t e(\tau) \mathrm{d}\tau + K_D \frac{\mathrm{d}e(t)}{\mathrm{d}t}$$

An implementation on a discrete controller requires a differential representation with sample period T_S:

$$u(i) = K_P \left(e_i + \frac{1}{T_n} \sum_{v=0}^{i} e_v\, T_S + T_v \frac{e_i - e_{i-1}}{T_S} \right)$$

Systems Under Control

Dynamic system responses are mainly classified into P (proportional), I (integrational), first-order $PT1$ and second-order lag elements $PT2$. $PT1$ elements approximate direct-current motor behavior (figure 2.1.1):

$$T_0\ y'(t) + y(t) = K_P\ u(t)$$

... and the Laplace-transformed form:

$$F_S(s) = \frac{Y(s)}{U(s)} = \frac{K_P}{1 + s\ T_0}$$

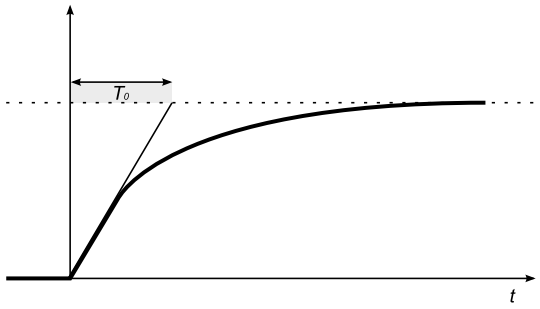


Figure 2.2: PT1: first-order delay element.

Going into more detail would exceed the limits of this thesis – further information is found in literature [SK98, Elm09].

2.2 System Development Using Models

Classical system development methodologies are mostly plan-driven where the results of each development phase are prerequisites for others. Changes in one phase affect the immediatly surrounding phases directly, they receive new development artifacts and pass them on to their subsequent following phase. In this way, deviations from the original design are handed on while every phase represents a potential source for additional errors in the solution.
The idea of introducing models into the development process counteracts error-chains by centering the process around a model. A model acts as a reference that is consistent to all phases in the cycle. The term *model* was manifoldly defined in literature and in the following two definitions are presented:

> "**Models** provide **abstractions** of a physical system that allow engineers to reason about that system by ignoring extraneous details while focusing on the relevant ones." [Bro04]

> "Engineering **models** aim to reduce risk by helping us better understand both a **complex problem** and its **potential solutions** before undertaking the expense and effort of a full implementation." [Sel03]

These definitions emphasize:

- Abstraction that enhances understanding.
- Representation of the problem domain on an appropriate level of abstraction that hides distracting details.
- Reduction of risks by acquiring solution information early.
- Minimize costs.

Introducing models in the development process requires extra effort spent on establishing a working infrastructure. Several tools already exist which provide a tool-chain reaching from requirements specification to implementation or at least assist in this process. Such tools are mostly proprietary and the question if the expenses are worth it is reasoned.

IT projects are very sensitive due to their per se complex nature, which makes it hard to imagine a possible solution for a given problem. Why IT projects fail has been studied for decades, for example in the large-field studies/roadmaps of the Standish-Group[5]. Emam and Koru [EK08] identify the reasons of IT project failures (table 2.1). The four largest entries have one common characteristic: They strongly depend on flexibility and system representation. Senior management is not sufficiently involved because projects could not be presented in an appropriate abstraction level, scope/requirement changes impact on the whole inflexible chain of development phases resulting in over budget.

All the mentioned reasons for IT project failures are related to the combination of risk and uncertainty - the less information and the less clear the envisioned concept of the product is, the higher are the costs for necessary changes in later development stages. The concepts of uncertainty and risk in software engineering can have the following reasons:

- Customers may not exactly know what they want.
- Requirements can be ambiguous and interpreted diversely by customer and contractor.
- Technical and management risks.
- Development efforts are over- or underestimated.

[5] http://www.standishgroup.com

Reason for cancellation	Percentage of respondents
Senior management not sufficiently involved	33
Too many requirements and scope changes	33
Lack of necessary management skills	28
Over budget	28
Lack of necessary technical skills	22
No more need for the system to be developed	22
Over schedule	17
Technology too new; did not work as expected	17
Insufficient staff	11
Critical quality problems with software	11
End users not sufficiently involved	6

Table 2.1: Reasons for project cancellations 2007, source: [EK08]

- Changes of methodologies and priorities during the project.

The relation between risk and uncertainty during an iterative software project are depicted in figure 2.3. The project starts with the definition of a the product accompanied by a high uncertainty about the real solution. The earlier the project time, the more unclear is the deviation from the defined end-product. Of course, requirements and specifications were defined under collaboration with the customer, however, there are always unknown characteristics (for example if a new technology could not perform as well as expected). These unknowns make a qualified prediction about the future outcomes hardly possible. With elapsing project time more information about development process and product is gathered - resulting in a reduction of uncertainty. According to the figure, an increased adaptability and flexibility to direction changes during the development is desirable. In this way the uncertainty is effectively narrowed and the development costs significantly decreased.

Efforts to master the system engineering task and the related pressing issues were usually pointing to the increase of manpower - which did not lead to satisfying results. Since then, a period of creating formalized system development methodologies was started: System development using models. Depending on the special use of models in the development cycle, the terms *model-based development*, with a model as pure reference, and *model-driven development*, with the focus on automatically artifact generation, are defined.

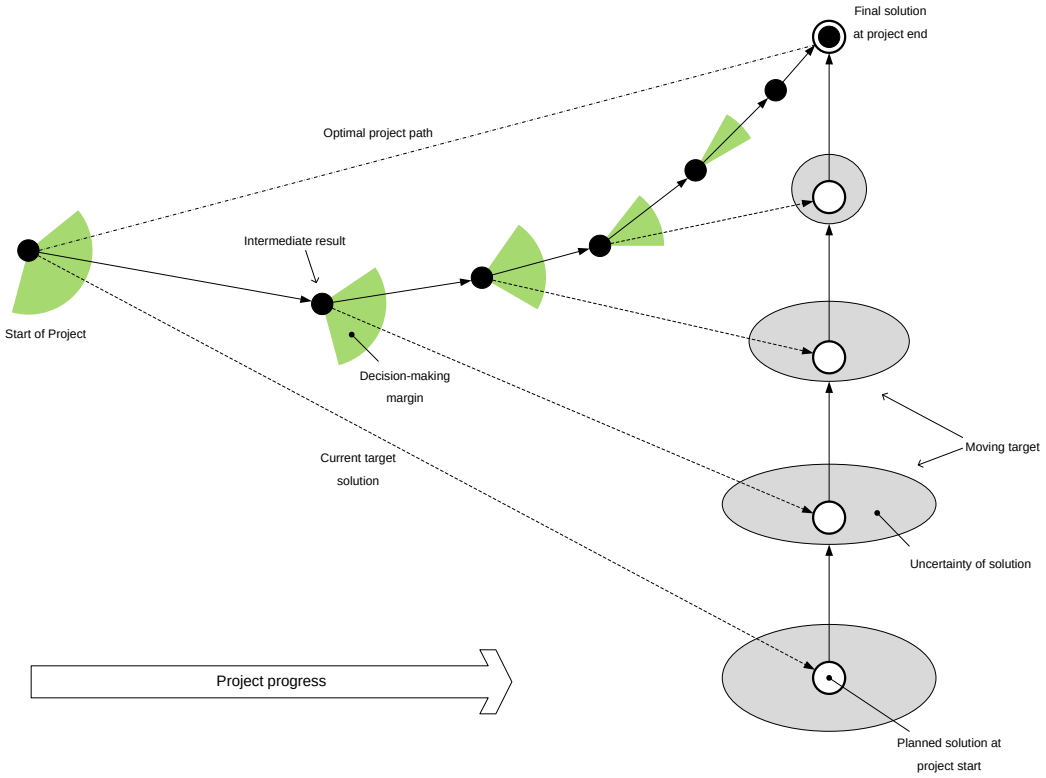


Figure 2.3: Uncertainty and Risk, source: [iGfiSm03], modified illustration

Models and Systems

Generally, it is desirable to describe systems with models. Integrating models can establish several advantages leading from improving the software development life cycle to estimations of project costs and formal verification of the system design.
A major concern in the development of embedded systems is to design systems and subsystems represented by different mathematical approaches: Controlled objects residing in a data and time domain with continuous nature have to be interfaced with controlling, discrete information systems. For example, a continuous systems might be described by an ordinary or partial differential equation, while discrete systems, such as a controller, could be described via a discrete event system. Shorty said, the composition of continuous and discrete dynamics is referred to as a *hybrid system*. A formal definition of a *hybrid system* follows.

Hybrid System

A *hybrid system* has a continuously evolving nature including casual jumps. These jumps can either be caused by the continuously evolving system or a change of states in a controlling system. An example of a hybrid system would be a plant controlled by a discrete controller system interfaced by an interface-model (see figure 2.4). Since hybrid systems are in the focus of this work, a formal definition of hybrid systems is being described.

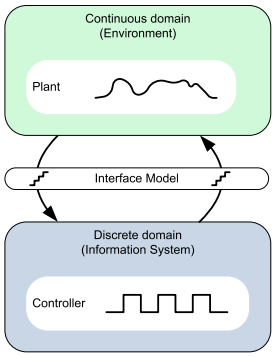


Figure 2.4: Scheme of a hybrid system.

Hybrid System - Formal Definition

A literature research about the definition of a hybrid system ended with the work of Carloni, Passerone, Pinto and Sangiovanni-Vincentelli who gave a formal definition of a hybrid system [CPPSV06]. The following definition

is essentially their work - recapitulations were made where it was possible without losing essential information.
Formally defining a hybrid system, requires some notations to be introduced beforehand. X, U, V shall be vector fields over subclasses of continuous dynamical systems:

X... Continuous state.
U... Input.
V... Disturbance.
$\mathcal{U}_\mathcal{C}$... Class of measurable input functions $u : \mathbb{R} \to U$.
$\mathcal{U}_d$... Class of measurable disturbance functions $\delta : \mathbb{R} \to V$.
$\mathbf{S}_C(X, U, V)$... is the class of continuous time dynamical systems which is defined by:

$$\dot{x}(t) = f(x(t), u(t), \delta(t))t \in \mathbb{R}x(t) \in X$$

f is a function such that for all $u \in \mathcal{U}_\mathcal{C}$ and for all $\delta \in \mathcal{U}_\lceil$, the solution $x(t)$ exists and is unique for a given initial condition.

Definition 1 *Hybrid System. A continuous time hybrid system is a tuple* $\mathcal{H} = (\mathbf{Q}, \mathbf{U}_D, E, X, U, V, S, Inv, R, G)$ *where:*

- $\mathbf{Q}$ *... set of states.*
- $\mathbf{U}_D$ *... set of discrete inputs.*
- $E \subset \mathbf{Q} \times \mathbf{U}_D \times \mathbf{Q}$ *... set of discrete transitions.*
- X, U, V *... continuous state, input, disturbance.*
- $S : \mathbf{Q} \to \mathbf{S}_C(X, U, V)$ *... mapping associating to each discrete state a continuous time dynamical system (in terms of differential equations).*
- $Inv : \mathbf{Q} \to 2^{X \times \mathbf{U}_D \times U \times V}$ *... mapping called invariant.*
- $R : E \times X \times U \times V \to 2^X$ *... reset mapping (the initial conditions upon entering a state).*
- $G : E \to 2^{X \times U \times V}$ *... guard mapping.*

A *discrete time* hybrid system could similarly be defined by replacing $\mathbb{R}$ with $\mathbb{Z}$ for the independent variable, and by considering classes of discrete dynamical systems underlying each state.

The tuple $(\mathbf{Q}, \mathbf{U}_D, E)$ might be seen as the definition of an automaton characterizing a discrete transition structure:

$\mathbf{Q}$ ... state set.
$\mathbf{U}_D$ ... inputs.
E ... transitions.

This automaton can change the states if either a discrete input event occurs or the invariant in Inv is not satisfied.

The dynamical behavior of a hybrid system is described by *executions*. An execution is a set of functions over time for the evolution of the continuous state as the system transitions through its discrete structure. The notation of a *Hybrid Time Basis* is needed beforehand the definition of the a hybrid systems execution:

Definition 2 *A Hybrid Time Basis τ is a finite or an infinite sequence of intervals:*

$$I_j = \{t \in \mathbb{R} : t_j \leq t \leq \acute{t_j}\},\ j \geq 0 \textit{ where } t_j \leq \acute{t_j} \textit{ and } \acute{t_j} = t_{j+1}.$$

Definition 3 *Hybrid System Execution. T is considered to be the set of all hybrid time bases.*

An execution $\mathcal{X}$ of a hybrid system $\mathcal{H}$, with initial state $\hat{q} \in \mathbf{Q}$ and initial condition $x_0 \in \mathbf{X}$, is a collection $\mathcal{X} = (\hat{q}, x_0, \tau, \sigma, q, u, \delta, \xi)$ where $\tau \in \mathcal{T}, \sigma : \tau \to \mathbf{U}_D, q : \tau \to \mathbf{Q}, u \in \mathcal{U}_C, \delta \in \mathcal{U}_d$ and $\xi : \mathbb{R} \times \mathbb{N} \to X$ satisfying:

1. *Discrete evolution:*
   - $q(I_0) = \hat{q}$;
   - *for all* $j, e_j = (q(I_j), \sigma(I_{j+1}, q(I_{j+1})) \in E$;
2. *Continuous evolution: the function ξ satisfies the conditions*
   - $\xi(t_o, 0) = x_0$;.
   - *for all j and for all $t \in I_j$,*

     $$\xi(t, j) = x(t)$$

     where $x(t)$ is the solution at time t of the dynamical system $\mathbb{S}(q(I_j))$, with initial condition $x(t_j) = \xi(t_j, j)$, given the input function $u \in \mathcal{U}_C$ and disturbance function $\delta \in \mathcal{U}_d$.
   - *for all* $j, \xi(t_{j+1}, j+1) \in R(e_j, \xi(\acute{t_j}, j), u(\acute{t_j}), v(\acute{t_j}))$.
   - *for all j and for all $t \in [t_j, \acute{t_j}]$,*

$$(\xi(t,j), \sigma(I_j), u(t), v(t)) \in Inv(q(I_j)).$$

- *if* τ *is a finite sequence of length* $L+1$*, and* $\acute{t}_j \neq \acute{t}_L$*, then*

$$(\xi(\acute{t}_j, j), u(\acute{t}_j), v(\acute{t}_j)) \in G(e_j).$$

In their work they state, that the behavior of a hybrid system consists of all the executions that satisfy the Hybrid System Execution definition:

- *Discrete Evolution Constraint.* The transitions (according to the transition relation E) of the system throughout its discrete states is constrained.
- *Continuous Evolution Constraint.* The execution must satisfy the system for each state and the invariant condition. If a invariant condition is violated, the system takes a transition to another state where the condition is satisfied. This constraint means, that a matching discrete input has to be supplied to the system.

An Hybrid System Execution is classified by its Hybrid Time Basis (definition 4).

Definition 4 *"A hybrid system execution is said to be (i) trivial if* $\tau = \{I_o\}$ *and* $t_0 = \acute{t}_0$*; (ii) finite if* τ *is a finite sequence; (iii) infinite if* τ *is an infinite sequence and* $\sum_{j=0}^{\infty} \acute{t}_j - t_j = \infty$*; (iv) Zeno, if* τ *is infinite but* $\sum_{j=0}^{\infty} \acute{t}_j - t_j < \infty$*."*

In this model of a hybrid system, an single input can lead to several valid executions - in that case, the system is non-deterministic (e.g. for incomplete systems or choice models). Defining priorities among the transitions can create a deterministic hybrid system.

2.2.1 Model-Based Development (MBD)

In model-based development the development process is centered around a model. This model has the purpose of pure documentation – there is only a mental connection between model and system implementation. This model should in the best case be consistent in any state of the development process, but this might be an impossible task since every process phase may adapt the model without a process-wide modification doctrine. Advantages to the classical plan-driven methods without models exist, however errros may still be introduced if the models are not evolved by well-defined guidelines.

The term model-driven development (MDD) is incorporated in the concept of model-based development. In this paradigm the model is not just pure documentation, it acts as specification and implemenation at the same time. The focus lies on the automatically generation of artifacts ensuring a system-wide model consistency: the generated code is always up-to-date. Several definitions of model-driven development can be found in literature, a pertinent definition for this thesis is given by Stahl:

> "Model-Driven Software Development is a generic name for techniques which automatically generate runnable software based on formal models." [SVEH07]

A sub-variant of model-driven development is *Architecture-Centric model-driven development* : Models are only used to generate the basic system infrastructure (e.g. skelleton components) which is enhanced manually by hand.

In this thesis the term model-based development is set equal to, or incorporates model-driven development depending on the context the term is used in. Note that in literature, abbreviations relating to model-based development and model-driven development exist, their meaning changes with the context. *MBSD* can stand for model-based system development or model-based software development - same case for *MDSD*. The terms *MBE (model-based engineering)* and *MDE (model-driven engineering)* are used when models meet system engineering issues (e.g. costs, risks, realization time). They relate to an approach of abstracting systems with the means of models and transform these systematically, with increasing concretization, until the level of executable models is reached.

Models in MDD

In the context of MDD models describe contents of a domain. This description has to be done formalized, otherwise a partial automation of the software process is hardly possible. It is not enough to go straight forward into the design of the model, first of all, the structure of the domain has to be understood and on base of that an abstract *meta-model* developed. Once a *domain specific language* (DSL), a synonym for meta-model, is created, it is possible to build a formal model within all the characteristics given by the meta-model. Figure 2.5 depicts a graphical overview of the relationship between formal models and meta-models.

A meta-model describes how a model (or a modeling language) can be structured and defines constraints, validity and modeling rules. As it is the case in the MOF[6] 2.0, instances are described by a model, which itself is

[6]Meta Object Facility

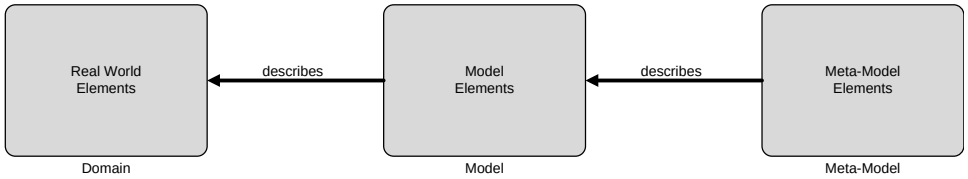


Figure 2.5: Model and meta-model relationships, source: [SVEH07], modified illustration.

described by an UML 2.0 model, which is in turn described by the MOF 2.0 (this is a meta-meta-model for an instance). Automated actions, like generations of code or model-to-model transformations are based on the meta-model holding the abstract syntax and static semantics of a modeling language (see figure 2.6). Formal models do not have to be graphical, a textual representation is also possible.

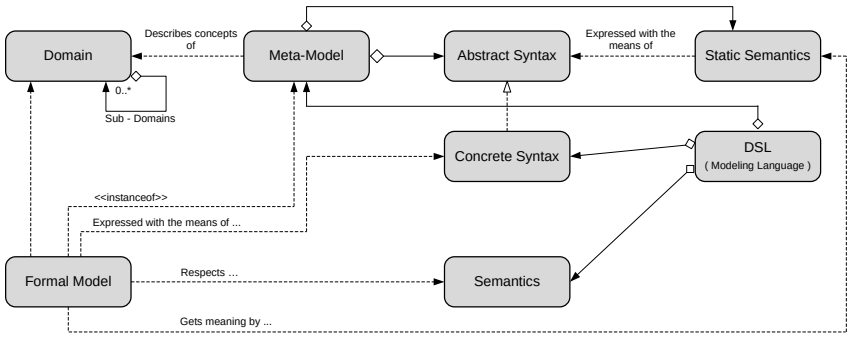


Figure 2.6: Models and DSLs, source: [SVEH07], modified illustration.

MDD and Approaches to MDD

This section gives an overview of some selected, already existing approaches to the model-driven development paradigms. The concepts of the popular MDA[7] approach are focused.

Model Driven Architecture (MDA)

MDA is the specialized approach to model-driven development by the OMG[8] which aims for a high portability (platform independence) and interop-

[7]Model Driven Architecture
[8]Object Management Group

erability (neutral to manufacturers) of software systems. In contrary to GSD, MDA is not explicitly focused on system-families.
MDA is an approach to link various technologies together by using formal models (MOF, UML, etc.) describing their relations and configurations. Based on these formal descriptions models can be technology and platform independently used, allowing the building of functionalities and services by model design.
The three primary goals of MDA are portability, interoperability and reusability through the architectural separation of concerns [OMG03].

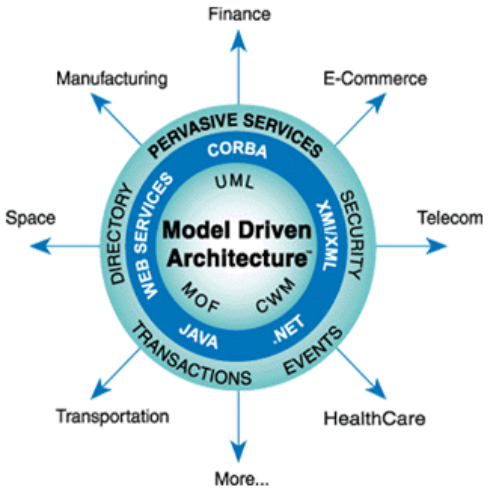


Figure 2.7: MDA - Overview, source: [Gro01]

In MDA, three different default models emerged from three different corresponding viewpoints.
The first model is the *Computation Independent Model* (CIM). The CIM describes the system in a less detailed manner, omitting details about internal processes, behavior and dependencies within the system. The *Platform Independent Model* (PIM) specifies the architecture of the underlying system without implementation (platform specific) details. Hence, the model should be suitable for different platforms that are described by a *Platform Specific Model* (PSM). PSMs are platform dependent views that use the platform specifications described by the PIM. Lastly, working code (or at least pseudo-code) will be generated from the PSM.
A *Platform Model* provides a set of technical concepts, representing the different kinds of parts that make up a platform and the services provided by that platform. It also provides concepts representing the different kinds of elements in specifying the use of the platform by an application. [OMG03]

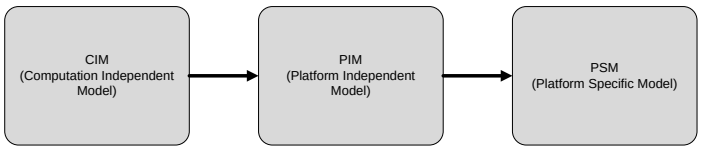


Figure 2.8: Dependencies in the Model Driven Architecture.

The key concept of MDA is the *Model Transformation*. In the Model Driven Architecture, additional information is added to a given PIM resulting in a PSM. These transformations are held as generic as possible. Any model can be transformed into another one in combination with a proper *transformation specification*. This transformation is also called the *MDA Pattern* (see 2.9). The types of transformations can be done manually, by using Profiles (UML), by using patterns or markings, or automatically. In the latter case, the design of PSMs becomes obsolete - all information for PIM to PSM transformations are already included in the PIM. The MDA pattern is not a one step process - it can be used in an incremental manner, meaning that a PIM can be transformed into various different PIMs before the generation of the PSM occurs.

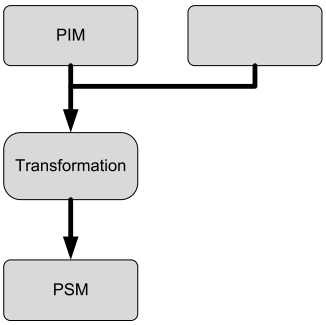


Figure 2.9: The MDA Pattern, source: [OMG03], modified illustration

Other Approaches

MDA might be the first thought when it comes to model-driven development, but other approaches exist which are not of lesser significance.

Software Factories

A *software factory* pattern faciliates the creation of individual software by assembling components. Software factories are primarily used for interoperability while focusing more on productivity in contrary to MDA. [GS03]

Generative Programming

Generative Programming (GP) gained popularity with the release of the book "Generative Programming" [CE00] which defines GP as follows:

> "Generative Programming is a software engineering paradigm based on modeling software system families such that, given a particular requirements specification, a highly customized and optimized intermediate or end-product can be automatically manufactured on demand from elementary, reusable implementation components by means of configuration knowledge."

Model-integrated Computing

In embedded systems, it is crucial to incorporate methods for a fast, secure and reliable real-time environment. *Model-integrated Computing* (MIC) sets its task to support development in this area, by providing an architecture for model-driven design - comparable to MDA. MIC is described on the ISIS page:

> "MIC focuses on the formal representation, composition, analysis, and manipulation of models during the design process. It places models in the center of the entire life-cycle of systems, including specification, design, development, verification, integration, and maintenance."[9]

[9] http://www.isis.vanderbilt.edu/research/MIC

3 Modeling Tools

Product requirements have been raising in parallel with cheaper embedded hardware. The complexity introduced by more demands and tight schedules require special techniques and tools to meet systems engineering requirements:

- **Minimized Costs**. The development effort should be kept as low as possible.
- **Reduced Risks**. System complexity should be understood as early as possible, changes in late project phases and resulting costs avoided.
- **Timely Delivery**. Time-to-market should be minimized and long overdues of deadlines prohibited.
- **Safe Design**. Safety and reliability standards are of importance in safety-critical embedded systems.

Modeling Techniques, including domain-architectures, domain-specific languages and best practices, provide ways for designing embedded systems with models. These techniques are utilized by modeling tools such as the commercial tool-set SCADE and Mathworks MATLAB, or the scientific prototyping software SynDEx and the hybrid system simulation environment Scicos.

Formal techniques abstract the characteristics of embedded systems by models. Such models can be viewed in different ways to support the understanding of the system. Formal models have the advantage of being verifiable with formal verification techniques (theorem provers, model checkers, tests) and the possiblity of automatic code generation. In some cases it is possible to obtain qualification/certification for code-generators (for example the KCG code generator in the SCADE suite).

This thesis investigates the Scics/SynDEx framework capabilities for model-based development. Scicos is based on a Signal-like formalism. Signal is a synchronous language. SynDEx is a tool for the distribution and temporal design of distributed systems and implements the AAA-Methodology that optimizes/distributes a given algorithm onto a target-architecture.

3.1 Scicos

Scicos (Scilab Connected Object Simulator) is a simulator toolbox and graphical system modeler and focuses on the design and simulation of hybrid systems (section 2.2) and a variety of DAE hybrid systems. Furthermore, a C code generator is included. Scicos finds its purpose in several physical and biological domains like signal processing and systems control. The cornerstone for designing continuous dynamics is a Simulink-like language extending a synchronous language. The main features of Scicos (Version 4.2) are[1]:

- Graphical models, simulation and compilation of dynamical systems.
- Combination of continuous and discrete-time behaviors in the same model.
- Selection of model elements from palettes of standard blocks.
- Programming of new blocks in C, Fortran, or Scilab language.
- Run simulations in batch mode from the Scilab environment.
- Generate C code from Scicos model using a code generator.
- Run simulations in real-time with and real devices using Scicos-HIL.
- Generate hard real-time control executables with Scicos-RTAI and Scicos-FLEX.
- Simulate digital communications systems with Scicos-ModNum.
- Use implicit blocks developed in the Modelica language.
- Discover new Scicos capabilities using additional toolboxes.

Scicos is a toolbox of the scientific laboratory software *Scilab*, a scientific open source software package for numerical computation which has been distributed freely since 1994. INRIA[2] and ENPC[3] have been developing Scilab with commitments of the Scilab Consortium, which was established in 2003 (16.03.2003) and is holding 25 members[4]. Signs for the popularity of Scilab are some notable companies and organizations like Axs Ingénierie, Cril Technology, CEA, CNES, Dassault Aviation, EDF, ENPC, Esterel Technologies, INRIA, PSA Peugeot Citroën, Renault and Thales being members of the consortium as well as about 10.000 downloads of the Scilab package per month [fRiCSC03].

[1] Features are listed on http://www.scicos.org
[2] Institut National De Recherche En Informatique Et En Automatique
[3] ENPC - École nationale des ponts et chaussées, Engineering Institute University Paris-Est.
[4] June, 2007. See http://www.scilab.org.

Since this software is used for some research in following sections of this thesis, more pages were spent to introduce this software in a little more detail. In literature, the book "Modeling and Simulation in Scilab/Scicos" [CNC06] is devoted to Scilab and Scicos - this section's work is leaned strongly towards it, equations and examples are taken from the book.

Scilab/Scicos Models

Scicos provides several ways of building models with different mathematical approaches, such as ordinary differential equations, boundary value problems, difference equations, and differential algebraic equations - they are described briefly in the following.

Ordinary Differential Equations. Scilab always assumes that an *ordinary differential equation (ODE)* is written in first-order (3.1). An ODE is is a differential equation with one independent variable (usually the time). ODEs of higher orders can be rewritten into ODEs of first-order by introducing new variables (see equation 3.2).

$$y = f(t, y) \tag{3.1}$$

$$\begin{aligned} &\text{The second-order ODE:} \\ \ddot{y}_1 &= \dot{y}_1 - y_2 + \sin(t), \\ \ddot{y}_2 &= 3\dot{y}_1 + y_1 - 4y_2 \\ &\text{can be written as:} \\ \dot{y}_1 &= y_3, \\ \dot{y}_2 &= y_4, \\ \dot{y}_3 &= y_3 - y_2 + \sin(t), \\ \dot{y}_4 &= 3y_3 + y_1 - 4y_2. \end{aligned} \tag{3.2}$$

Boundary Value Problems. A *boundary value problem (BVP)* is a differential equation with boundary conditions. A two-point boundary value problem has the general form of equation 3.3.

$$\dot{y} = f(t, y), \quad t_0 \leq t \leq t_f, 0 = B(y(t_0), y(t_f)). \tag{3.3}$$

Difference Equations. Difference equations find their utilization in problems with discrete data - data values change only at discrete points in time. Difference equations can often be approximated by numerical computations of differential equations. A solution solving a difference equation is a sequence $y(k)$ (equation 3.4).

$$y(k+1) = f(k, y(k)), \quad y(k_0) = y_0. \tag{3.4}$$

Differential Algebraic Equations. *Differential Algebraic Equations (DAEs)* are a general form of differential equations for vector-valued functions composed of differential and algebraic equations (equation 3.5). A characteristic of DAEs is that they might only be solvable for certain initial conditions (so called consistent initial conditions). Scicos is able to solve index-one DAE systems. If a DAE model is given there are two ways: Rewrite the DAE model into a simpler DAE or an ODE.

$$F(t, y, \dot{y}) = 0 \tag{3.5}$$

Scicos Model Abstraction and Simulation

Subsystems are modeled for the control algorithms residing in the discrete domain and the environment located in the continuous domain. These two models are interfaced to each other, date and time discretized which enables the simulation of the complete system (see figure 3.1).

Scicos Formalism

The formalisms used in Scicos are based on Signal and its extension to continuous-time systems [NAN03]. The simulator of Scicos uses two standard ODE/DAE numerical solvers which handle the continuous and discrete parts of the diagrams. A Scicos diagram is made by combining blocks connected via signals while single blocks offer a good opportunity to distribute the development to several teams.

Scicos Syntax

Blocks are connected via *regular signals* and *activation signals*. In that way a data precedence (regular signal) between computational functions (blocks) is achieved and the activation time of blocks (activation signal) is modeled. Figure 3.2 depicts a simple Scicos diagram: An event generation block (clock

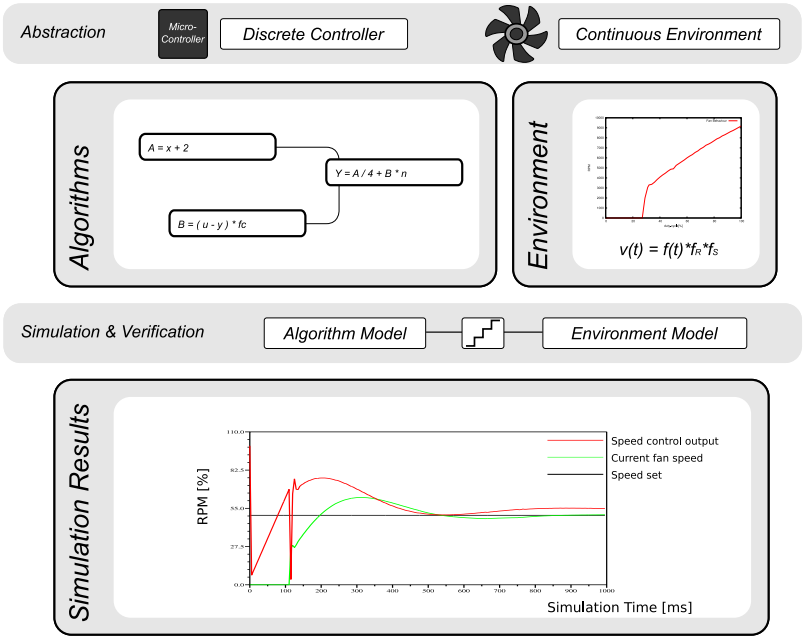


Figure 3.1: Scicos - Model abstraction and simulation.

block “Event at time t”) evolves the activation signal (signal between clock and ”Source”). The activation signal triggers the functional block “Source”. The two regular blocks “Source” and “Func. 1” have a data signal (black) connecting them. The data signal models the passing of data between blocks and shows the “Source” block preceding the “Func. 1” block.

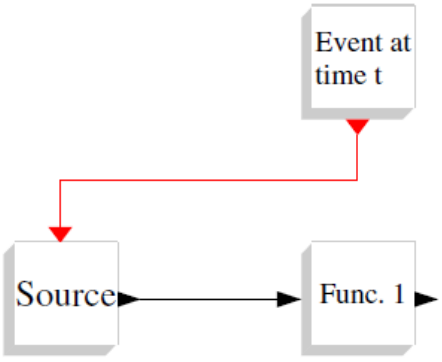


Figure 3.2: A simple Scicos diagram, source: [CNC06]

Scicos Blocks

A *Scicos Block* is a graphical representation of a simulation function. In general, there are two types of representing blocks: *basic blocks* and *super blocks*. The purpose of *super blocks* is to contain Scicos sub-diagrams and *super blocks* in a single block. Thus, a design-hierarchy is created which supports the readability of low-level diagrams. Scicos is delivered with a library of several standard blocks for building algorithms.

If customized blocks are needed, the *Scifunc* block can be used to define an algorithm by Scilab expressions, or new *basic blocks* can be constructed by implementing an *interfacing function* and a *computational function*. The interfacing function handles the graphical behavior in Scicos, input/output port definitions and the initialization states of the block, while the computational function specifies the behavior of the block during the simulation. How an example *basic block* can be constructed is briefly described in the appendix (A.2). Blocks may take parameters for defining initial and behavioral values. Parameters can be immediate values or symbolic parameters defined in the Scicos diagram context.

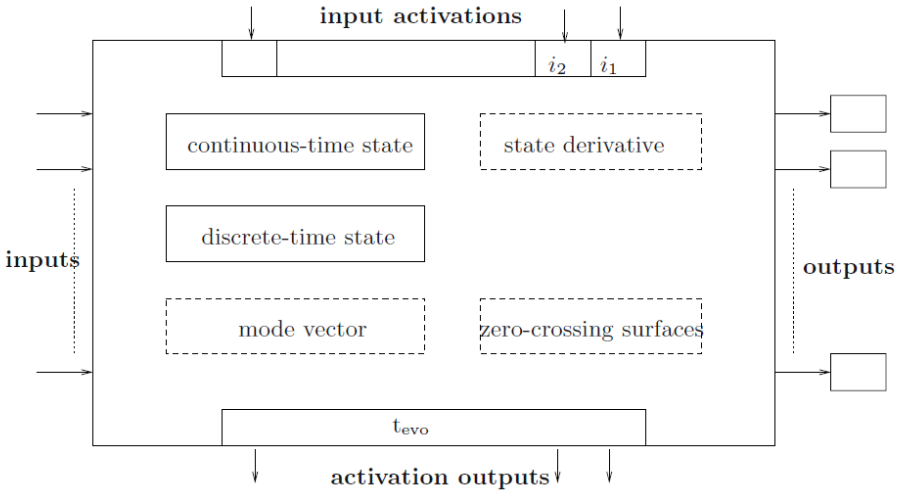


Figure 3.3: A Scicos Block. [CNC06]

A Scicos block comprises several components (figure 3.3):

- **Regular inputs and outputs.** Vector input are passed through regular paths to the vector outputs.
- **Input and output activations.** Activation paths link input event signals to output activation signals.

	Regular Input	Regular Output	Discrete State	Continuous State	Activation Output	Activation Input
CBB	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$
DBB	$\surd$	$\surd$	$\surd$	$\emptyset$	$\surd$	≥ 1
ZCBB	$\surd$	$\emptyset$	$\emptyset$	$\emptyset$	$\surd$	$\emptyset$
SBB	1	$\emptyset$	$\emptyset$	$\emptyset$	≥ 2	1

Table 3.1: Scicos - Basic Block Interfaces.

- **Continuous-time vector state.** The continuous time state of the block.
- **Discrete-time vector state.** The discrete time state of the block.
- **Zero crossing surfaces and Mode vector.** Inside Scicos the numerical integrator has problems with functions being not continuously differentiable. A mode can be defined to make sure that the numerical integrator never reaches such points of discontinuity inside an integration interval.
- **State derivative.**

Depending on the combination of the simultaneously included features and how a block is activated, Scicos blocks can be classified into four different groups: *Continuous Blocks*, *Discrete Blocks*, *Zero-Crossing Blocks* and *Synchro-Blocks* [CPPSV06].

A *Continuous Basic Block (CBB)* continuously monitors its input ports and updates its output ports and states. A *Discrete Basic Block (DBB)* is activated only if it receives an event on its activation input. A *Zero-Crossing Basic Block (ZCBB)* is activated only if one of its regular inputs crosses a not differentiable point (e.g. crossing zero in an absolute value function). *Synchro Basic Blocks (SBB)* can generate output activation signals that are synchronized with their input events - these blocks are the "event select block" and the "if-then-else block" represented in C code as "switch" and "if then else" respectively. These blocks can for example be used for a frequency division of an activation signal. A summary of the block types with their corresponding features is listed in table 3.1.

Scicos Semantics

In the following a summarized view about the behavior of blocks and signals in Scicos is presented.

Scicos Signals

Scicos activates blocks by activation signals. Such activations can make the block updating its respective in-/outputs, updating its states, and computing its state derivative. When such a block is activated depends on the activation type which can either be continuous or discrete. Continuous activation allows a signal to evolve permanently, while an *event* triggers a block to update itself. Inbetween two events the regular signal remains unchanged (see figure 3.4). These concepts provide a certain amount of time-control for the design of hybrid systems.

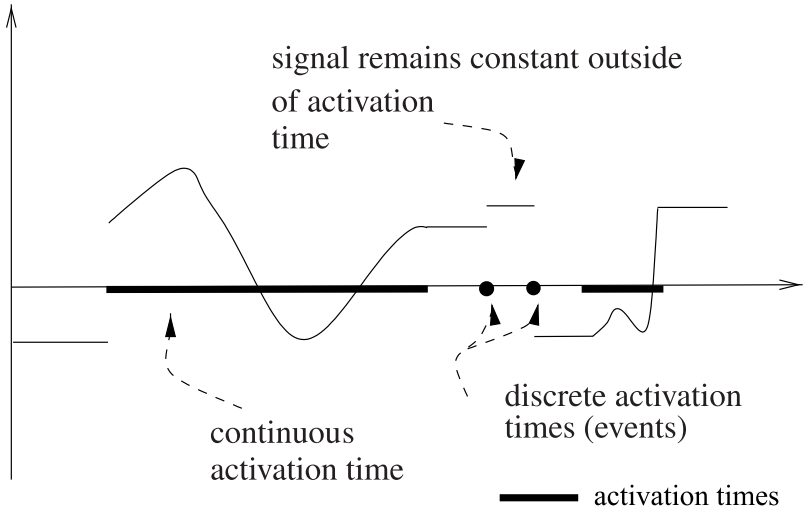


Figure 3.4: Scicos - Activation time dynamics, source: [CNC06].

Scicos blocks are activated in three different ways - *External Activation*, *Always Activation* and *Internal Zero-Crossing* (figure 3.5).

- **External Activation**. A Block is activated when it either receives an activation signal on its activation input port, or inherits activation (3.1).
  - *Event Activation*. An event triggers the update of the block's out-

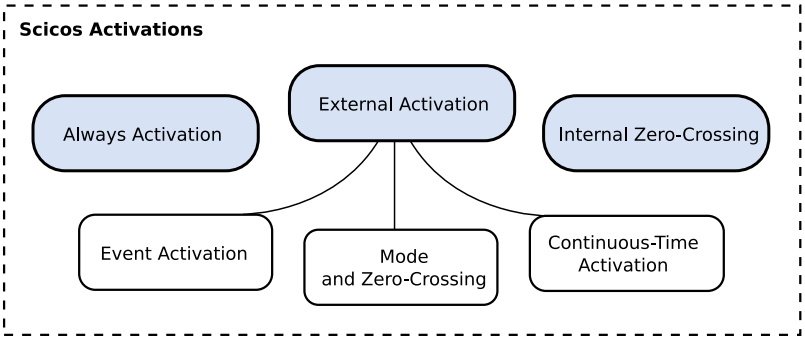


Figure 3.5: Scicos - Activations - Hierarchy.

put.

$$y(t_e) = f_1(t_e, x(t_{\bar{e}}), z(t_{\bar{e}}), u(t_e), \mu(t_e)).$$
$$y(t_e) \ldots \text{Vector-outputs.}$$
$$u(t_e) \ldots \text{Vector-inputs.}$$
$$t_e \ldots \text{Event time.}$$
$$x(t_{\bar{e}}) \ldots \text{Continuous-time state.}$$
$$z(t_{\bar{e}}) \ldots \text{Discrete-time state.}$$

A blocks can also implement its own activation outputs and provide thus a delay function:

$$t_{evo} = f_3(t_e, x(t_{\bar{e}}), z(t_{\bar{e}}), u(t_e), \mu(t_e)).$$

Updates of internal states are associated to following update-function:

$$[z(t_e), x(t_e)] = f_2(t_e, x(t_{\bar{e}}), z(t_{\bar{e}}), u(t_e), \mu(t_e)).$$

– *Continuous-Time Activation.* If a block is defined as *always active* then it is activated at specified time intervals instead of events. The output depends on a continuous time activation period:

$$y(t) = f_1(t, x(t), z(t), u(t), \mu(t)).$$

– *Mode and Zero-Crossing.* If a function is not continuously differentiable (smooth) then the numerical integrator cannot post a solution. The *mode* is used to avoid the integrator reaching such points inside an integration interval (the interval depends on the step size of

the solver) by defining an integration start period. A *Zero-Crossing Surface* is introduced to make sure the integration stops at points of discontinuity inside an integration interval.

- **Always Activation.** The block is defined to be always active (e.g. the Scicos sine-generator block). This is a special case of a continuous-time activation with a fictitious activation in put port.
- **Internal Zero-Crossing.** The internal state of the block is updated if a zero-crossing occurs inside the block. Internal zero-crossing events are not predictable.

Scicos Activation Inheritance

Not all the Scicos blocks may have activation input ports. If a block does not, the activation is inherited through its regular input port. In figure 3.6 a simple diagram is shown where block "Func. 1" inherits the activation from the block "Source" after the Scicos pre-compilation phase.

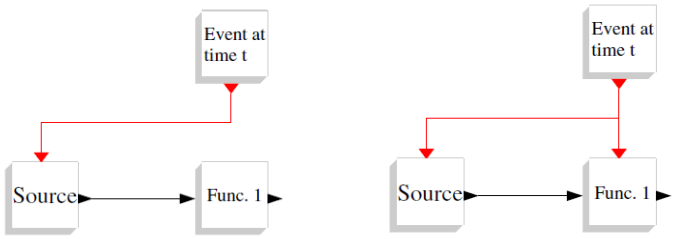


Figure 3.6: Scicos activation inheritance, before and after pre-compilation phase [CNC06].

Scicos Synchronism

Scicos blocks are synchronized if they are activated by the same activation source (e.g. the *Event Clock* block). If two blocks are connected together, Scicos executes them in the correct order. A design pitfall might be choosing two *Event Clocks* with the same activation period to induce synchronism: This will not execute blocks synchronized - they might be activated by Scicos in any arbitrary order (see figure 3.7).

Memory Blocks Modeling a discrete delay in Scicos can be done with the *Register* block: When it is activated it copies its internal state on the output

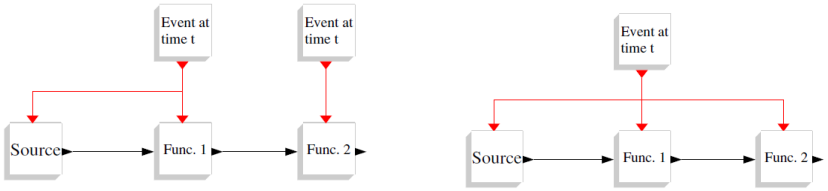


Figure 3.7: Scicos synchronism concepts. Asynchronous and synchronous diagrams, source: [CNC06].

and the input is copied into the internal state (see figure 3.8 for a simple Scicos diagram with a discrete delay).

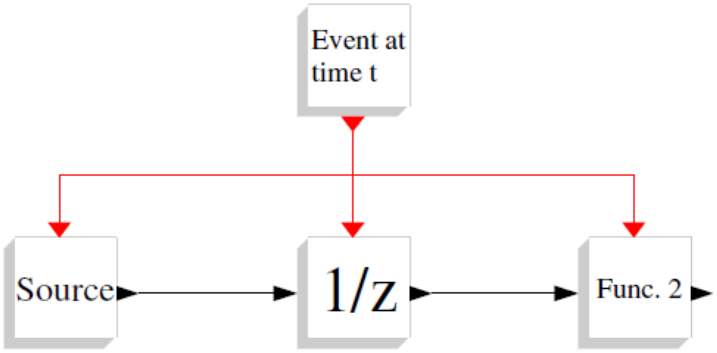


Figure 3.8: Scicos - Register block, source: [CNC06].

3.2 SynDEx

SynDEx (Synchronized Distributed Executive) is a system level CAD software based on the *Algorithm Architecture Adequation (AAA)* [Sor94] and has been designed at INRIA in the Rocquencourt Research Unit France, by the AOSTE team. Its goals are to provide a tool for rapid prototyping and the optimization of distributed real-time embedded applications on multi-component architectures.

The key features of SynDEx are[5]:

- **Rapid prototyping** of complex distributed real-time embedded applications including automatic code generation.
- **Automatic generation of safe and optimized distributed real-time code**. Formal verification of possible implementations can be done manually or automatically using the optimization heuristics based on multi-periodic distributed real-time scheduling analysis.
- **Hardware/Software co-design through multi-component architecture** if some parts of the application must be implemented by software and run on processors, while others must be implemented by hardware and run on specific integrated circuits.
- **Interface with domain specific languages** such as synchronous languages (*Esterel*, *SyncCharts*, *Signal*) providing formal verifications, AIL [PCM01] (a language for automobile), *Scicos* a Simulink-like language, *CamlFlow*[6] a functional data-flow language, *UML2.0* with the *MARTE* profile, etc.
- **System level CAD tool**. SynDEx offers a software environment reaching from the specification level (functional specifications, distributed hardware specifications, real-time and embedding constraints specifications) to the distributed real-time embedded code level, through (multiple) workstation functional and timing simulations.
- **Interface with the integrated circuit level CAD tool** *SynDEx-IC* which allows the implementation of a function (operation) onto a specific integrated circuit that can be used as a non-programmable component in the multi-component architecture.
- **SynDEx is freeware** - free of charge for non-commercial applications.

[5] as listed on http://www.syndex.org/scicosSyndexGateway/index.htm

[6] CamlFlow is a Caml to data-flow graph translator. Caml is a general-purpose programming language, designed with program safety and reliability in mind.

SynDEx Models

SynDEx leads the two fields of software and hardware development closer together by combining software algorithms (algorithm models) and highly abstracted target hardware (architecture model). The main focus of SynDEx lies in optimizing, scheduling and distributing algorithms under constraints (e.g. given by the capabilities of the target hardware). Thus a hardware architecture model in SynDEx is a high level abstraction of the execution environment and is merely more than a pure description, while the algorithm models are abstracted at a level of the developer's desire, matched to constraints, computed and optimized.

In SynDEx models, information is transferred between the targets by a communication medium. A communication medium definition (e.g. a bus) contributes twofold: to the architecture model by defining the connections between the target processors, and the algorithm model, through the definition of durations for information transfers.

Summarizing, SynDEx combines three models covering several aspects in system development: An *algorithm model* for the representation of computational functions, a *medium model* defining the communication medium between the execution environments and an *architecture model* describing the hardware structure and capabilities. With these models available, SynDEx performs an optimization of the computational function by algorithm adequation (*adequation* is the seeking for an optimized implementation of an algorithm onto a distributed architecture by the execution of heuristics). The algorithm model is thus partitioned into several computational units, and these units are distributed onto the execution targets. Under consideration of hardware constraints (algorithm parts restraint to an execution environment) and timing properties of algorithm execution and communication media (durations), the optimization is performed by a greedy algorithm. This algorithm takes advantage of potential parallel structures, thus optimizes the execution time of the final executive. The result of the algorithm adequation is an algorithm distributed in portions over the whole hardware architecture, that has a total order for communication and algorithm actions and is guaranteed to be deadlock free.
This generated, distributed algorithm model can then be transformed into a *macro code model (MCM)* based on a M4 implementation of the UNIX macro processor. The target-independent MCM can then be translated into a target-dependent model (such as a programming language like C) with the use of customized M4 definitions. The resulting target-dependent model can then be translated into an executable model (e.g. by using a C compiler).

SynDEx Model Abstractions

All SynDEx models reside in the discrete time and data domain. They are thought to be used for modeling discrete computation algorithms executed on a discrete hardware (e.g. microprocessors). Computational functions are interfaced to each other by: defined discrete data types, execution periods and durations. Algorithm portions, residing on different operators, communicate via the *communication medium* which is an abstract representation of a synchronized communication channel.

An algorithm model is conceptually a *platform independent model (PIM)*, while the *platform description model (PDM)* contains the hardware architecture information. Applying the algorithm adequation on these two models results in a partitioned algorithm that is dependent on the hardware in respect of timings and target placements. At this stage the algorithm represents a *platform specific model (PSM)* - more precisely, a *target-language independent PSM (TLI-PSM)*. The development-chain finishes with the transformation into a MCM, its expansion to a *target-language dependent PSM (TLD-PSM)*, followed by a transformation into the *executable (EM)* - see figure 3.9.

SynDEx Formalism

An *algorithm model* is a (preferably) graphically specified directed, acyclic graph (DAG). The DAG consists of *blocks* representing a sequence of operations, and *signals* representing dependencies between operations.

An *architecture model* describes the heterogeneous multiprocessor execution environment by a non-oriented hypergraph of operators. Operators in the architecture model are connected by a *communication model*, also called a *communication medium* (see figure 3.10).

SynDEx, as seen as a DSL, is basically defined by a graphical syntax with semantics and corresponding meta-models defined by the AAA-Methodology. SynDEx's formal models are meant to describe a subset of the real-time embedded distributed systems domain.

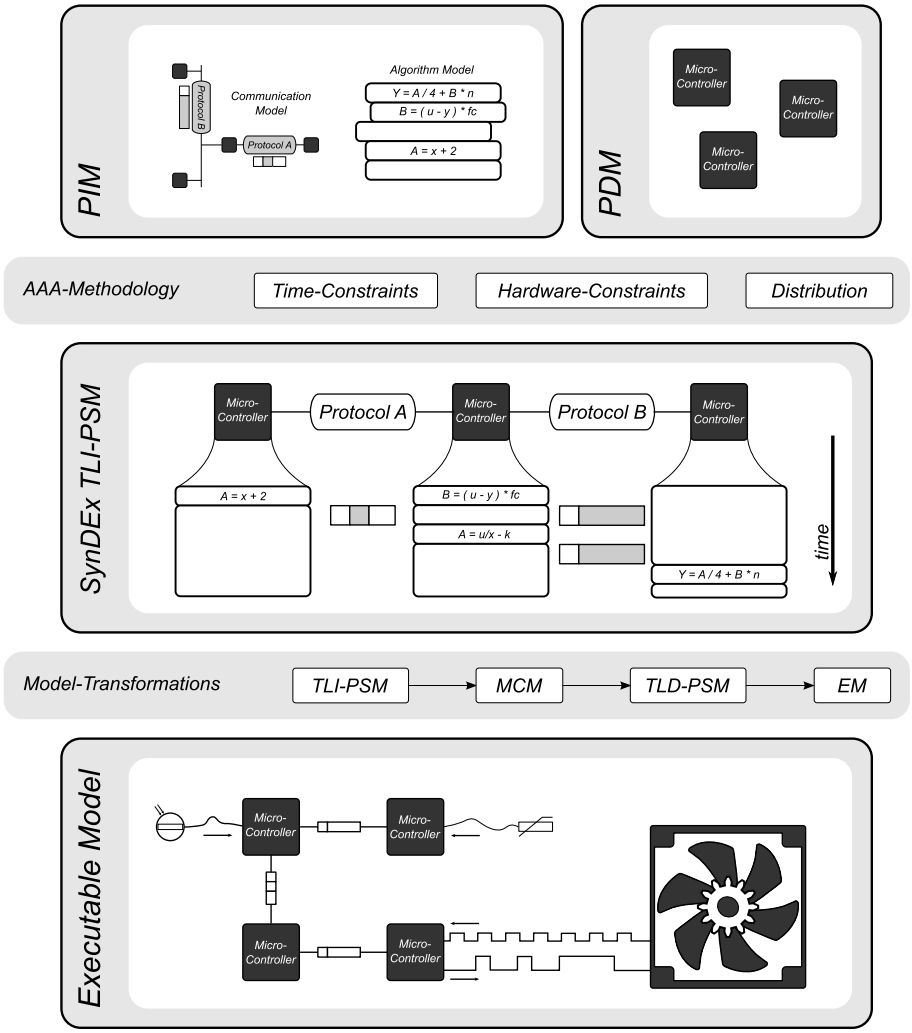


Figure 3.9: SynDEx: models and abstraction.

SynDEx Syntax

In the DSL of SynDEx, graphically represented models (*concrete syntax*) are instances of the AAA-Methodology's DAGs (*abstract syntax*). Model corresponding semantics are defined by the AAA-Methodology. SynDEx-models can also be designed in a textual way, of course with the drawback of reduced cognitivity.

A simple, graphical overview of an algorithm model is depicted in figure 3.11. A constant and a sensor block (e.g. temperature sensor) are providing data to the following functions *function1* and *function2* with the data types *uint16*

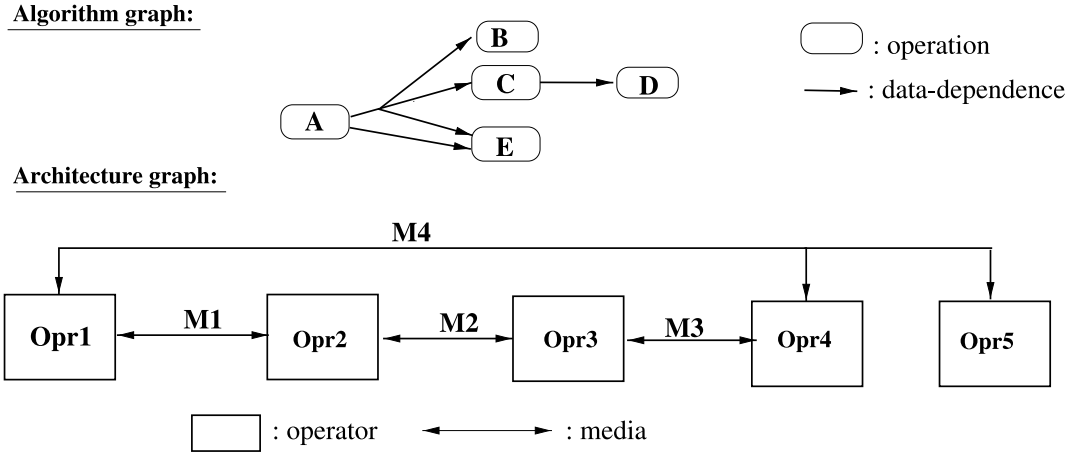


Figure 3.10: SynDEx formalism - Algorithm, architecture and communication models, source: [GLS99].

(16-bit unsigned integer) and *uint8* respectively. After termination of function1 and function2 the data is forwarded to *superblock1* (a block containing a sub-model). The results of this block are delayed by *delay1*, computed in *function3* (which takes a double data type as input, and provides an uint8 output) and finally passed to *actuator1* (which could be e.g. an electric motor).

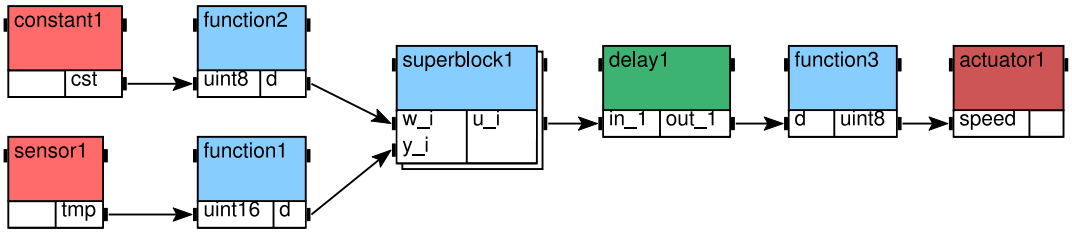


Figure 3.11: SynDEx syntax - graphically represented algorithm model containing all types of blocks.

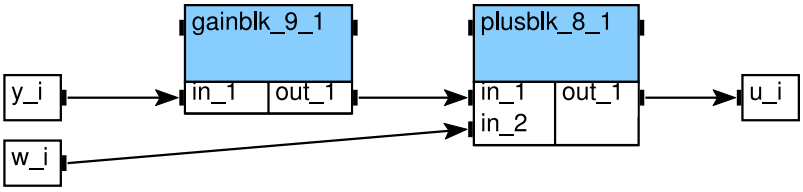


Figure 3.12: SynDEx syntax - Internals of the superblock in figure 3.11.

SynDEx Blocks (Algorithm model)

A SynDEx *block references* an *algorithm definition*. A reference corresponds to exactly one algorithm definition, while a given algorithm definition may correspond to several references (a 1:n relationship). An algorithm definition is a platform-independent representation of a sequence of atomic operations, that means SynDEx considers all modeled blocks (except hierarchical blocks) to be not interruptable.

The characteristics of an *algorithm definition* are:

- An algorithm definition *name*.
- *Input / Output port definitions* with corresponding data-types.
- *Precedence dependencies* setting the logical execution orders between blocks.
- *Data dependencies* stipulating the order of the data flow between blocks.
- *Execution durations*. An integer value specifying the time this algorithm requires on a corresponding execution environment (e.g. the WCET[7] in processor clock cycles).
- *Operation Period* setting the periodically execution of the algorithm.
- *Allocation constraints* unchangeably pre-defining the allocation of the algorithm onto an execution environment.

Depending on the *type* of the algorithm definition, appropriate input and output ports are declared (figure 3.11 holds an example of each type):

- *Sensor*. A sensor block defines an input operation to the algorithm. In the algorithm model these are the first blocks called.
- *Actuator*. Actuators are outputs of the algorithm and thus the last operations called.
- *Function*. Function blocks represent a computational operation.
- *Constant*. These blocks act as a fixed value input to the algorithm and are also part of the first blocks called.
- *Delay*. Delay blocks act as discrete memory blocks.

The scheme of a function-block can be obtained in figure 3.13. *Precedence dependencies* determine the order of block executions, thus they define if the block precedes or succeeds other blocks. *Data dependencies* (in SynDEx so called *strong precedence dependencies* assign the order of data-flows between

[7]Worst-Case Execution Time

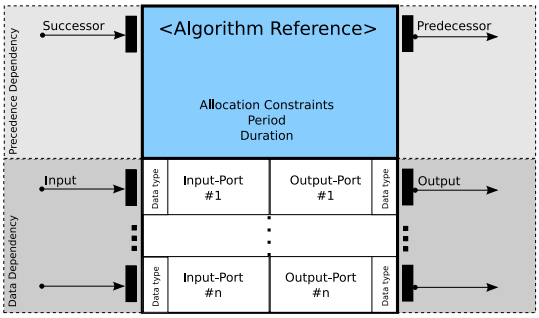


Figure 3.13: A SynDEx block.

blocks and by that also imply precedences. *Input-ports* and *output-ports* can be arbitrarily edited: Several ports can be declared with data types assigned to. *Allocation constraints* are used to force SynDEx placing an algorithm on a desired target. For example this would be useful if special hardware functionalities are only available for one type of targets. If an algorithm shall be periodically called, the *period* parameter of the block allows specifying execution periods. Before assigning periods or durations, a time-representation for the model has to be chosen: a relation between the time representation of the execution environment and the time representation in the SynDEx modeling environment. *Durations* of algorithms can be defined manifold, for each operator (target) a duration has to be defined; SynDEx needs that information to perform the optimization and distribution of the algorithms. Figure 3.14 depicts the general scheme of a SynDEx blocks appearance in the graphical SynDEx editor.

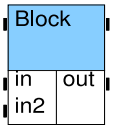


Figure 3.14: A SynDEx block as viewed in the graphical SynDEx editor with two inputs and one output port.

SynDEx Model Hierarchy and Conditions

Superblocks (the "superblock1", figure 3.11 and its contents in figure 3.12) can represent a set of blocks which lie on a lower level of the abstraction hierarchy. Finite automatons can be realized by superblocks with conditioned ports, that means an input port of the superblock realizes the state of the automatons like a switch statement (not only superblocks, but every block could be seen

as a finite automaton). Additionally, the introduction of superblocks inside an algorithm model can increase the readability of the model. This kind of a block is not considered to be atomic, that means the set of connected blocks (a algorithm sub-model) beneath it might be partitioned by the AAA-Methodology.

SynDEx Blocks (Architecture and Communication Model)

An architecture model is built by *operator blocks*, *memory blocks* (with or without arbiter) and *bus/mux/demux blocks* (with or without arbiter) connected by *communication blocks*. Operators execute operations sequentially, communicators execute communications sequentially.

The most important characteristics of an operator block (e.g. node0 in figure 3.15) are:

- *Operator Name*. An identification.
- *Operator Type*. The type of the operator, e.g. an Atmel ATmega128 microprocessor.
- *Communication Gates*. They are used to specify full-duplex ports which may be used for the connection to communication blocks.

A *communication block* is a model of a communication medium, which can be one of the two types: *Sequential Access Memory (SAM)* and *Random Access Memory (RAM)*. SAM can be defined as a *point-to-point* medium where only two operators are connected to each other, or a *multipoint* medium that allows several operators to be connected to each other via the communication medium.

An exemplary architecture model of a hardware setup containing four microcontrollers connected by a bus via the communication model *comA*, and a desktop pc connected to target node0 via *comB* is depicted in figure 3.15.

SynDEx Semantic (Algorithm model)

SynDEx blocks are connected via signals. Signals have the purpose of modeling data-flow and control-flow. Pure control-flow connections define rules for the ordering of the blocks in the final optimized SynDEx model while data-flow connections define data-flow (and imply control-flow) precedence dependencies. Blocks can only be connected, if all involved input and output ports are of the same data-type. This circumstance makes it clear how operations have to interface each other and therefore eliminates some issues of the error-prone process in data passing during the development process.

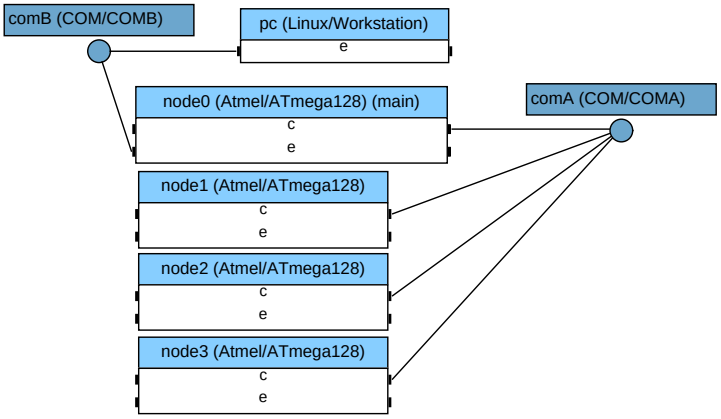


Figure 3.15: SynDEx architecture model. Four nodes (ATmega128 microcontrollers) and a desktop PC (Linux workstation) are connected via communication gates "c" and "e".

An algorithm model runs in a forever-loop, thus actuator blocks represent the last instances of a current iteration of the algorithm, while sensors and constant blocks stand for entry points at each iteration.

SynDEx Modeling

Software development methodologies with SynDEx have strong similarities to rapid application development and prototyping. The design flow with SynDEx can be imagined as follows (figure 3.16):

1. *Algorithm model design.* The application is modeled with blocks and corresponding dependencies. For each block periods, durations (depending on the type of the target operator) and allocation constraints (the placement of a block onto a specific target) are defined.
2. *Architecture and medium model design.* The architecture is modeled with operators connected by communication media.
3. *Algorithm adequation execution.* SynDEx' algorithm adequation processes the models - the result is a synchronized executive having partitions of the algorithm distributed among the targets.
4. *Assessment of the timing and scheduling model.* Blocks and their distribution onto the targets, timings and dependencies are graphically displayed. If the results are not satisfying the algorithm model has to be redesigned.

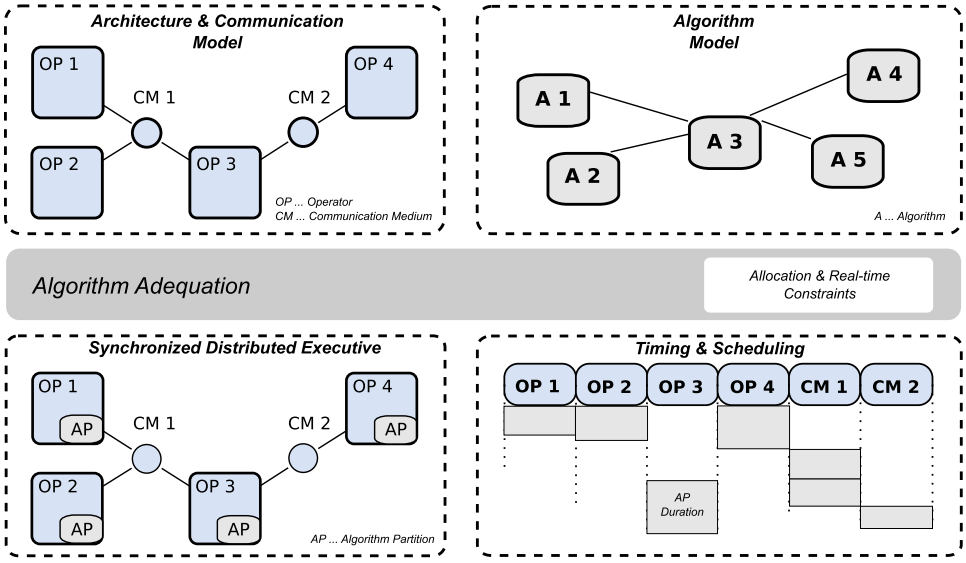


Figure 3.16: Design flow with SynDEx.

If the resulting SynDEx model is schedulable and satisfying the requirements, a MCM can be transformed from SynDEx model. The macro code itself might then be transformed into an executive model by customized code generation definitions.

SynDEx Scheduling

The following paragraphs start with a brief description of the SynDEx scheduling, especially the SynDEx scheduling tables. Later on a comparison of scheduling tables to real-world scheduling accompanied by a small discussion and proposals for the identified problems follows. Scheduling tables are results of the AAA methodology and presented graphically inside SynDEx. In general, time representations of a scheduling table in SynDEx include:

- Medium Channels in multiprocessor architectures - Communication processes require channel resources, e.g. sending a data packet from one processor to another via an UART protocol.
- Communication processes. Transfer of information from one processor to another.
- Processors - Atomic tasks are executed on the processor with their periods and durations.
- Tasks. Stateless, simple tasks with a duration and period. A simple task is a task that is executed until it's termination is reached without being

interrupted [Kop97].

Data is sent/received between targets by sender/receiver tasks connected by medium channels. In SynDEx every target is realized with its own communication thread running concurrently to the main routine, that means that the time consumption of a communication job includes two targets exchanging data (the sender and the receiver task). In fact there could be more targets involved that are simultaneously listening to the bus but discarding unintended packages.

The scheduling table does not hold the behavior of the algorithm during the init phase of the blocks. The scheduling table represents the timely behavior of the algorithm during the cyclic / periodic phases of the respective blocks, while init and end phases are not included. As depicted in figure 3.17, an algorithm is repeated after the *Inter-repetition Period*. Every action in the distributed executive is synchronized. Synchronization can be thought of a token, or more tokens, being passed around from finished jobs to others. Preventing data-depending jobs being executed too early is done by creating communication threads and main loops on each target controlled by semaphores in such a way that tasks will be called in a correct execution order. *Intra-repetition Synchros* order the execution of tasks within one whole execution of the algorithm, while *Inter-repetition Synchros* delay the execution of tasks, in the scope of two successive repetitions of the whole algorithm, until their execution is needed.

SynDEx Scheduling - Model versus Reality

SynDEx creates a scheduling which resides conceptually in an ideal world. In the ideal world, execution durations of tasks are constant, therefore the same is valid for the periods of tasks. In a SynDEx multiprocessor architecture, all processors have a synchronized start-up and do not have clock drifts. All these characteristics have to be assumed to keep a scheduling model simple, and in cases of higher complexity even calculable.

Every task requires the definition of durations (for the internal sequential calculations). In a general point of view, tasks can have various execution durations which are not preliminary known. Choosing the worst case execution time (WCET) for the task duration might be common practice because it is used to layout the system in the case of maximal load. Even though this sounds fairly reasonable, there are several issues arising when it comes to the realization of a scheduling model in the real world (see figure 3.18). The tasks in this figure shall be seen as stateless, simple tasks which are just executed after another without any external synchronization (e.g. a timer unit that ensures the tasks exactly being executed at modeled times). Modeling task durations and fitting

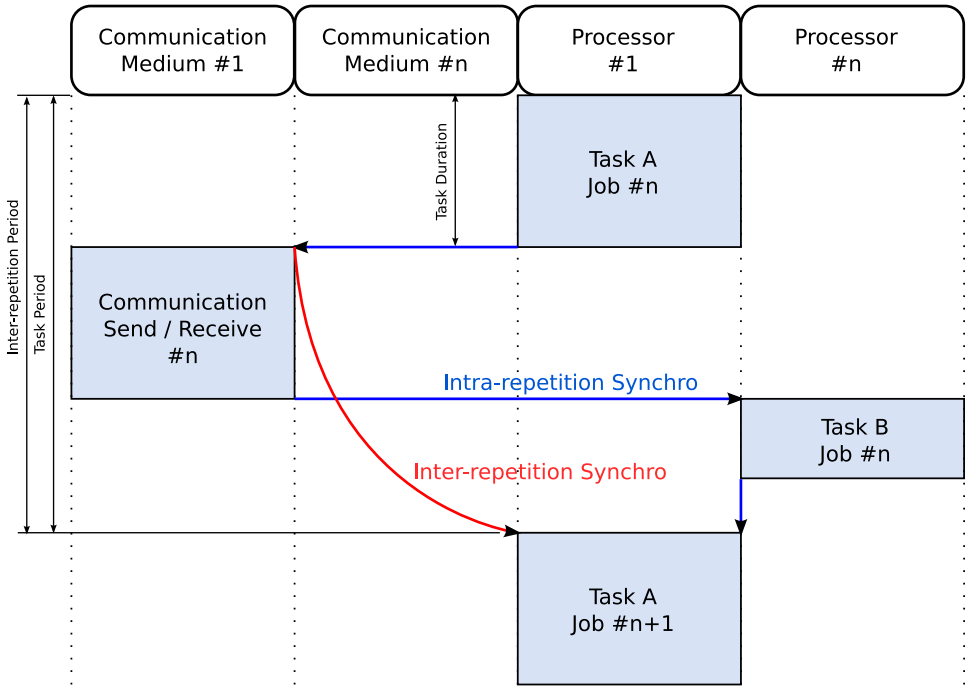


Figure 3.17: SynDEx - Scheduling table with two processors and two communication media.

those to the real world, requires a stronger way of modeling which considers various possible execution times of tasks - in SynDEx this is for sure a deviation between the model and the real world. The problem of tasks terminating at unknown times is not just relating to SynDEx models, but rather a general issue.

Period violations occur if there are no synchronizing units responsible for the regular execution of the task. In a pure event-driven task model there is no guarantee that the tasks will be executed exactly in their defined period, but might rather be executed earlier. Information are pushed in this kind of system. In a best-effort application these circumstances might be acceptable, but if an application with fault-tolerance is needed, e.g. taking sensor samples at fixed time periods, this kind of model might not be sufficient. Early execution of the tasks might even lead to some tasks being executed more times inside a time interval if there is no synchronization available.

If timer peripherals are provided on a target hardware, tasks with fixed periods can be realized, but still with the introduction of a deviation between model and reality. A fixed-period task has to include a hardware timer unit, which fires the task at desired time intervals and has to be blocking until the

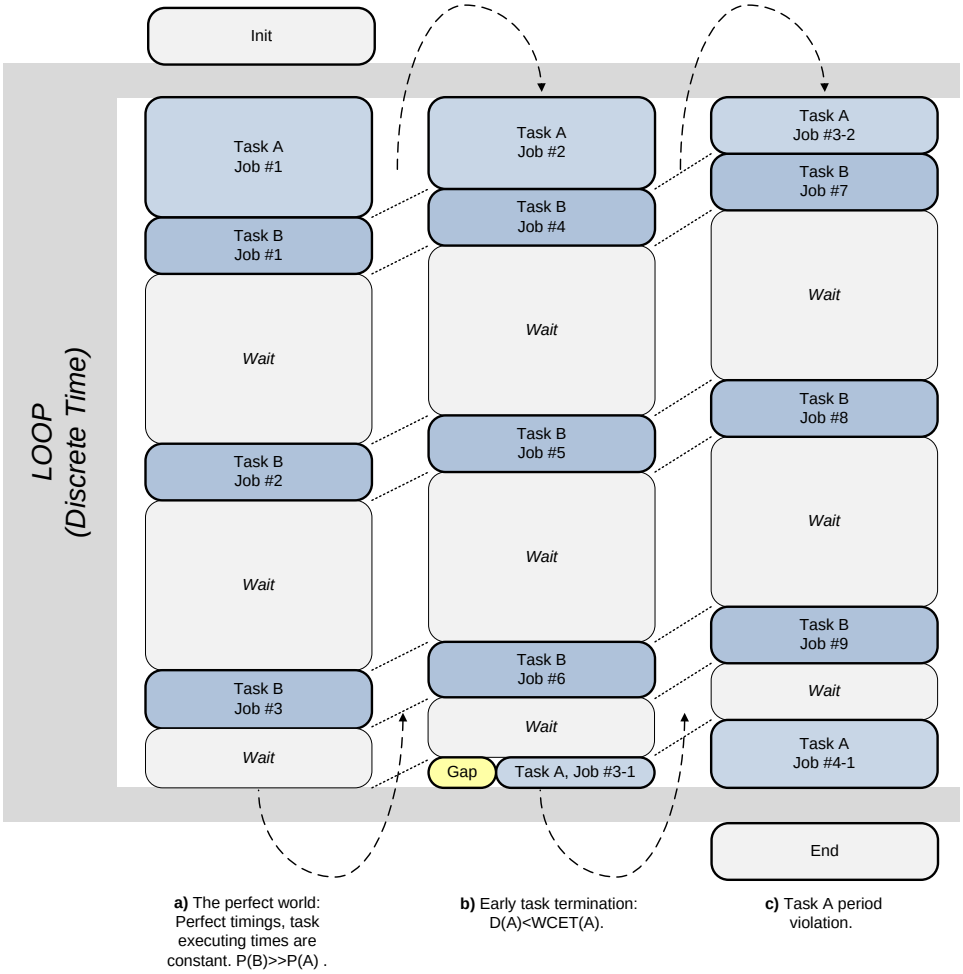


Figure 3.18: SynDEx - Scheduling with tasks of varying execution times. a) In the perfect (modeled) world, all task durations and periods are exactly known and do not change during execution. The period of Task A $\mathbf{P}(\mathbf{A})$ is much smaller than the period of Task B $\mathbf{P}(\mathbf{B})$. b) Executed on the target hardware, Task A terminates earlier than its WCET (duration of Task A $\mathbf{D}(\mathbf{A}) < WCET(A)$). Therefore a gap in the scheduling occurs, which makes Task A being executed earlier in the next repetition of the loop. c) Due to (b), the period of Task A is violated.

activation time is reached. Blocking is necessary to compensate in time for the early task terminations - this is not actually seen in the scheduling model, but is a method of realizing a fixed-period task (see figure 3.20).

Another way of synchronizing tasks is by introducing more operators into the hardware architecture model of SynDEx. These operators could model

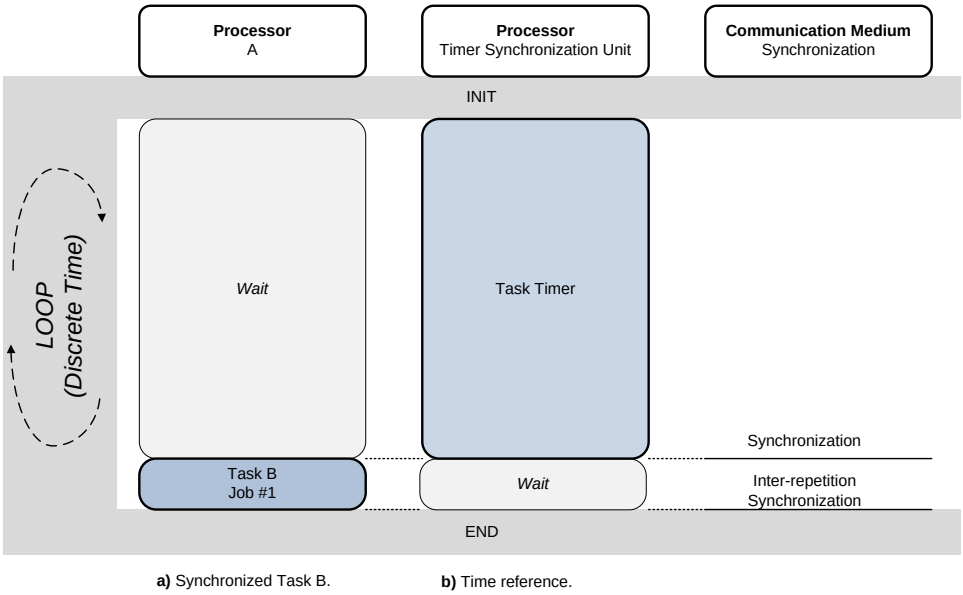
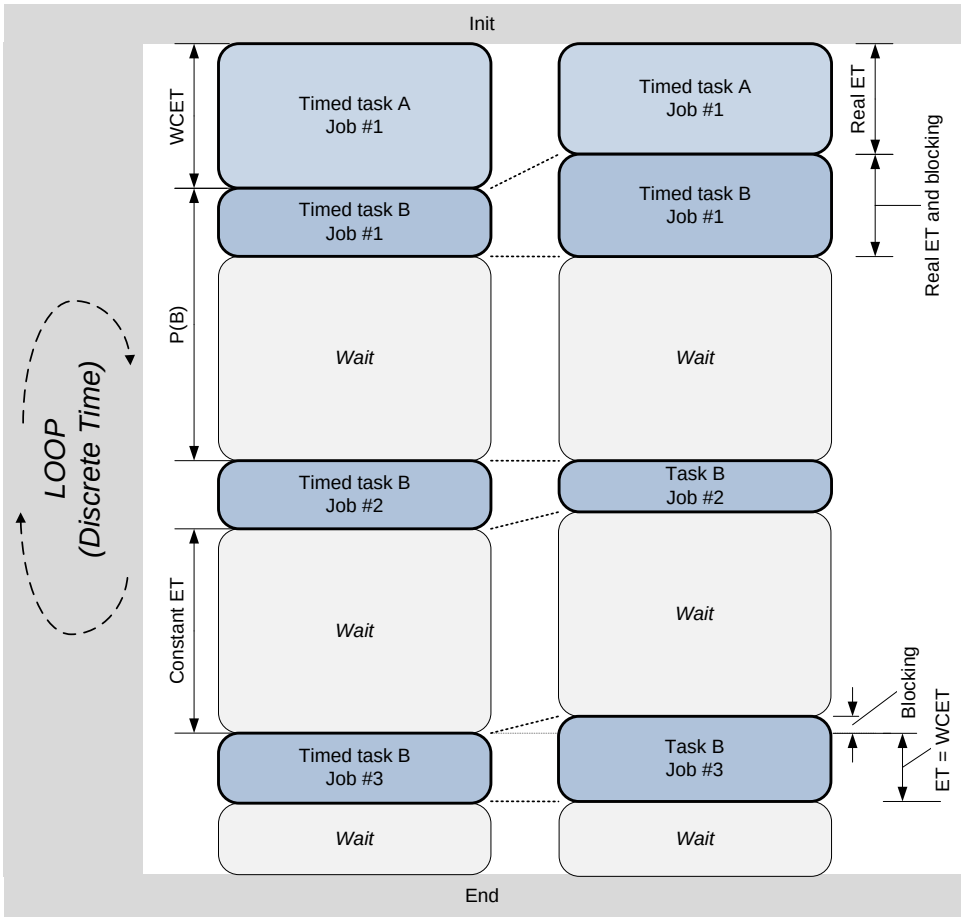


Figure 3.19: SynDEx - Synchronizing a task with a timing operator.

a timer unit on the processor itself and thus, by synchronization operations, maintain a fixed period of a task (see figure 3.19). This is possible in the model, but the macro code generation part of SynDEx at the current version used in this thesis (7.0.0) is not fit for this issue - this solution would require additional customized model transformation and code generation programs. In the case of a multi-processor architecture, meaning several targets have to communicate and work collaboratively, the varying task durations will tend to average out since synchronization calls are done between the targets (synchronization and communication).



a) Scheduling model. b) Real scheduling.

Figure 3.20: SynDEx - Model versus a real execution instance. a) In the SynDEx model execution times are constant, while b) at a real execution tasks may terminate earlier which is compensated by blocking timed tasks.

3.3 Scicos-SynDEx Interface

The model-based character of the Scicos/SynDEx framework can improve the system development process. The Scicos/SynDEx framework System does not only enhance the interaction between several phases, but also reduces the number of phases in the process model in a sense of aggregating several phases in a single phase where a great deal is done by the used model-based framework. Consider the software development sub-model of the V-Model 97 (figure 3.21) which provides a good example of showing the impact of introducing models in the system development process. This figure acts as an example how the model-based Scicos/SynDEx framework is applied to a document-centric, waterfall-model based software development model.

Scicos and SynDEx models are used as a formalized system representation in almost every phase of the process and thus providing a consistent view of the product throughout the whole software development life cycle. For example, a Scicos model might be used in meetings with project stakeholders and users in order to determine the requirements. The same model is also used in the system design phases and the verification phases in the development life cycle. As it might have been already noticed by the reader, a system process model like the one in figure 3.21 can not be executed as is with Scicos/SynDEx: A classical plan-driven, step-by-step proceeding is not what is happening when designing models, because during the design of a Scicos/SynDEx model, many phases are advanced at the same time, since the models contribute to all phases simultaneously: directly (model is the immediate representation of the system) or indirectly (Scicos provides the model for the SynDEx model which generates code in the implementation phase).

These circumstances lead to a new software development life cycle with Scicos/SynDEx (see figure 3.22). At the start of this SDLC hybrid system models are defined in the Scicos context, simulated and models refined. If the simulation results are satisfactory, the model (specification part modeled in Scicos) can be model-to-model (Scicos-to-SynDEx) transformed and be part of the system specification represented by SynDEx models, which is, in the next phase of the SDLC, formal verified and treated with temporal simulation by the SynDEx software. In case of failing the requirements, the hybrid system part might have to be redesigned, that means the Scicos model has to be edited again, a model-to-model transformation done and the resulting model used again for the simulations in SynDEx. Automatic implementation, code generation, for different execution environments after successful simulation can be done by the SynDEx software and the results be validated afterwards. Instead of transcending from the executable code to integration and system integration verification phases, the validation phase is directly hit. The simulations and

verifications done in the previous phases make it possible to skip, a now already integrated, additional verification phase (provided that there is enough trust in the software development tools). In case the results are positively validated the cycle can stop, otherwise the models have to be worked on again in the modeling phase.

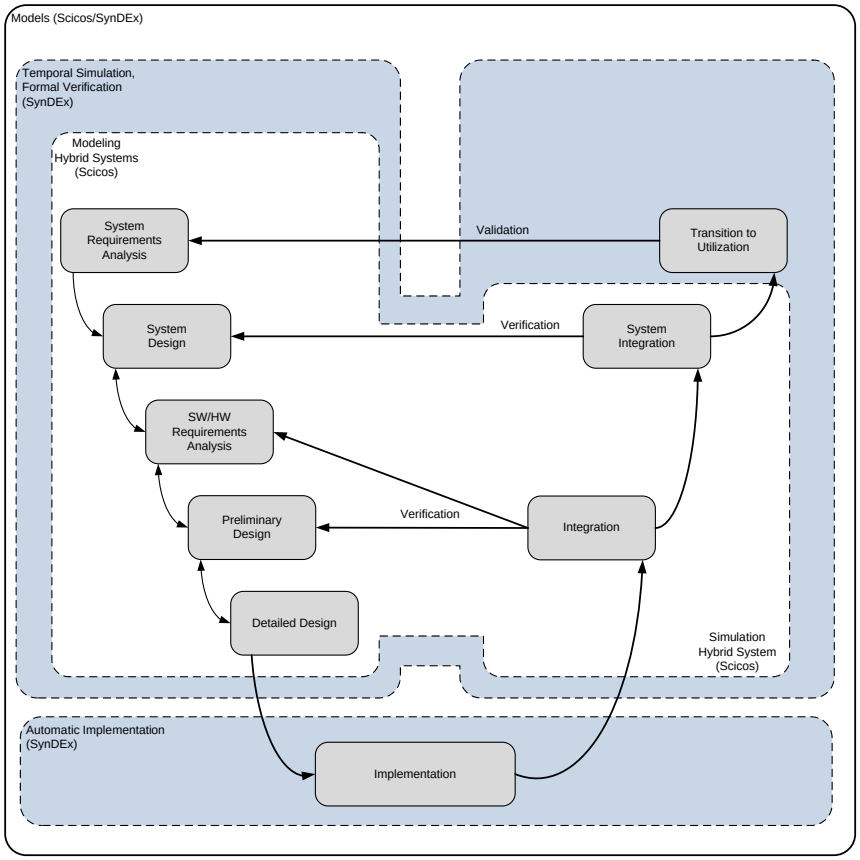


Figure 3.21: A Software Development Lifecycle (V-Model '97) with the Scicos/SynDEx framework.

Scicos-SynDEx Gateway

As mentioned above, the SDLC with the Scicos/SynDEx framework requires an intermediate step translating a Scicos model into a SynDEx model. This model-to-model transformation is done via the Scicos-SynDEx Gateway - an add-on tool-box of Scicos. With this gateway the hybrid system model designed in Scicos is transformed into a SynDEx model which can then be combined with other SynDEx models (see figure 3.23).

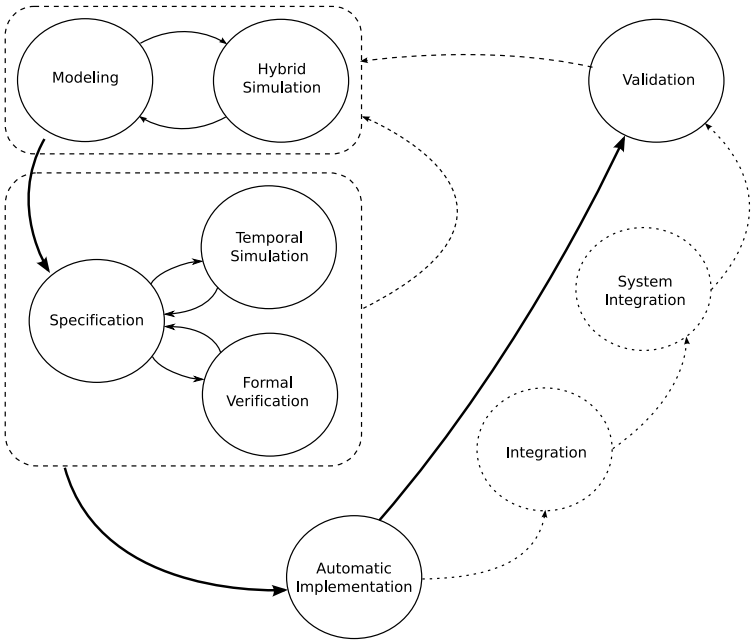


Figure 3.22: The software development lifecycle with the Scicos/SynDEx framework, source: [KS04] - modified illustration

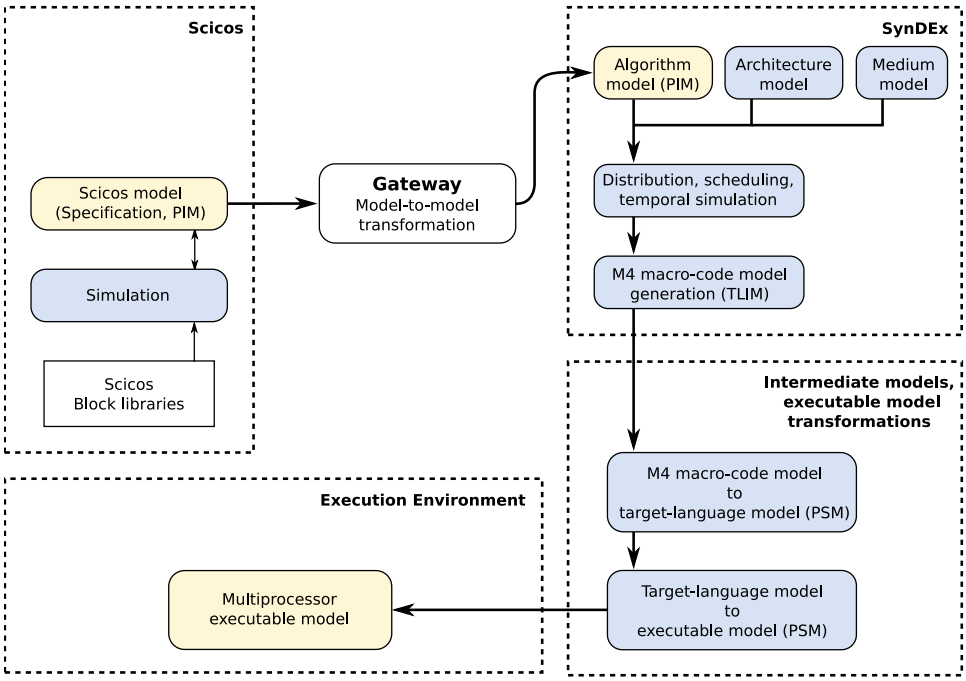


Figure 3.23: Scicos/SynDEx gateway.

4 Demonstrations of MBD with the Scicos/SynDEx Framework

Mapping the real world onto a model that retains all information of interest and represents models in a resource-friendly (memory, time) fashion should be the aim of any modeling software. The Scicos/SynDEx framework, Scicos for the modeling and simulation of discrete/continuous systems and SynDEx for the distribution and scheduling of models on a distributed system, might show a good way for modeling hybrid systems, simulation, optimization and scheduling (AAA Methodology) on a multi-processor system. How far and in which ways this framework can be applied to a distributed embedded system is shown by the implementation of two examples on a customized embedded system development board. One example implements a fan speed PID control circuit on a monoprocessor, using both, the Scicos and the SynDEx software tools, in combination. The second example implements a simple observer pattern on a multiprocessor system, where two observer nodes are notified about the temperature values from a sensor node. The architecture of the target hardware used for these examples is described in section 4.1, information about the software architecture of the examples can be found in sections 4.2 and 4.3 along with the corresponding Scicos and SynDEx models.

Measuring the capabilities of a model-based framework is not an easy task. Sufficiently understanding of the modeling software is a prerequisite for assessments. What kinds of requirements and specification can successfully be modeled with this specific model-based development framework? In which ways should the application be distributed among all the resources? What is an efficient solution? How is the scheduling of the algorithms done? How can one visualize the resource allocation (time and data) of the possible solution? What code metrics can be applied to measure the quality of the produced code? These are just a couple of questions which will at least partially be answered by the implementation of the two examples following in this section.

4.1 Hardware Architecture

The hardware used for the evaluation of the Scicos/SynDEx modeling capabilities is a development and educational printing circuit board with sensor/actuator peripherals (from now on to be referred as the "ESE-Board" - Embedded Systems Engineering Board) created by Kössler at the Technical University of Vienna (complete information regarding the board can be found in the creator's master thesis [Koe09]). Basically, this board consists of four 8-bit ATmega128 microcontrollers (Node0 - Node3) where each controller node is interconnected to the others by a bus. Furthermore, there are various Input/Output peripherals assigned to each microcontroller, such as light intensity and temperature sensors, as well as actuators like a fan, liquid crystal displays, LEDs and LED bar-graphs. The programming of the microcontrollers is done by an USB interface via an USB-to-Serial circuit. Thus, this PCB is a good evaluation board for hybrid systems. The microcontrollers realize the control part of the system, while the controlled peripherals implement a real-world plant (dense time, continuous data values). A very simplified layout of the board is shown in figure 4.1 and a real-world photography in figure 4.2. Note that only for the examples' most relevant ESE-Board parts are depicted.

4.1.1 The components of the ESE-Board

The nodes with their attached peripherals form a distributed embedded system, which can be accessed by a PC workstation through an USB interface. Several peripherals are connected to the microcontrollers, that includes two LEDs for each node. An LCD, a light bulb, a temperature sensor and a fan are connected to one specific microcontroller. Every node (Node0-Node3) is an ATmega128[1] microcontroller holding following basic characteristics (only the most important ones are listed):

- 8-bit microcontroller.
- 32 x 8 General Purpose Working Registers + Peripheral Control Registers.
- Fully Static Operation.
- Advanced RISC Architecture.
- 128KB In-System Programmable Flash Memory.
- 4KB EEPROM.

[1]The Atmel Corporation, http://www.atmel.com

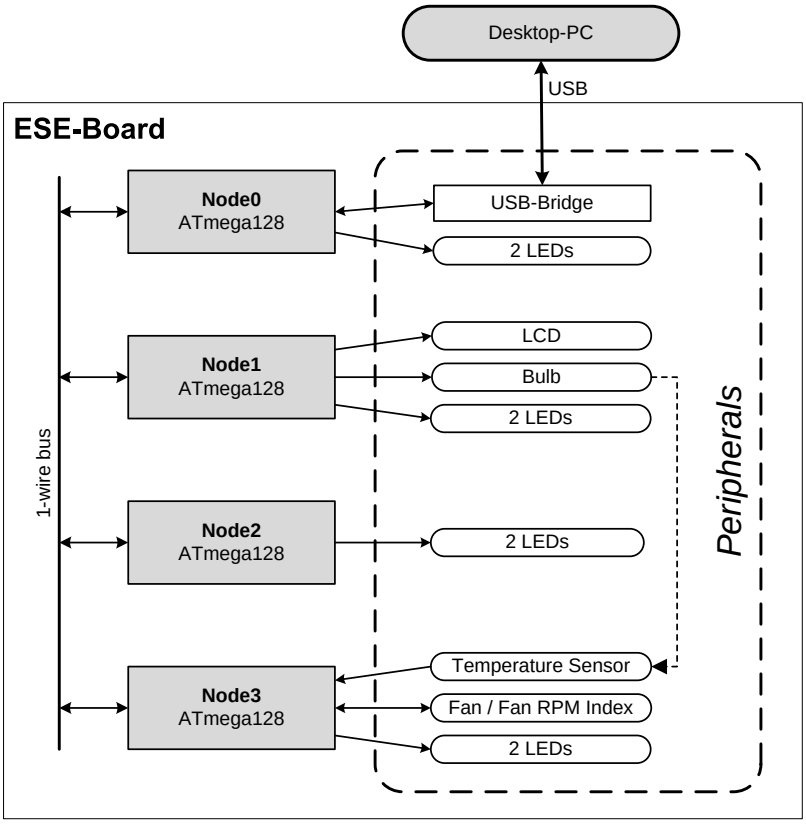


Figure 4.1: ESE-Board - Simplified layout.

- 4KB Internal SRAM.
- 133 Instructions, most of them take a single clock cycle for the execution.
- JTAG (IEEE std. 1149.1 Compliant) Interface.
- Two 8-bit Timer/Counters with separate prescalers and Compare Modes.
- Two Expanded 16-bit Timer/Counters with separate prescaler, Compare Mode and Capture Mode.
- Two 8-bit PWM Channels.
- 8-channel, 10-bit ADC.
- Dual Programmable Serial USARTs.
- Master/Slave SPI Serial Interface.
- 53 Programmable I/O Lines.

The time-source for the microcontrollers is an external clock block running with 14.745600 MHz. Every node is connected to the bus via a HW- or SW-USART (Hardware- or Software- *Universal Synchronous and Asynchronous*

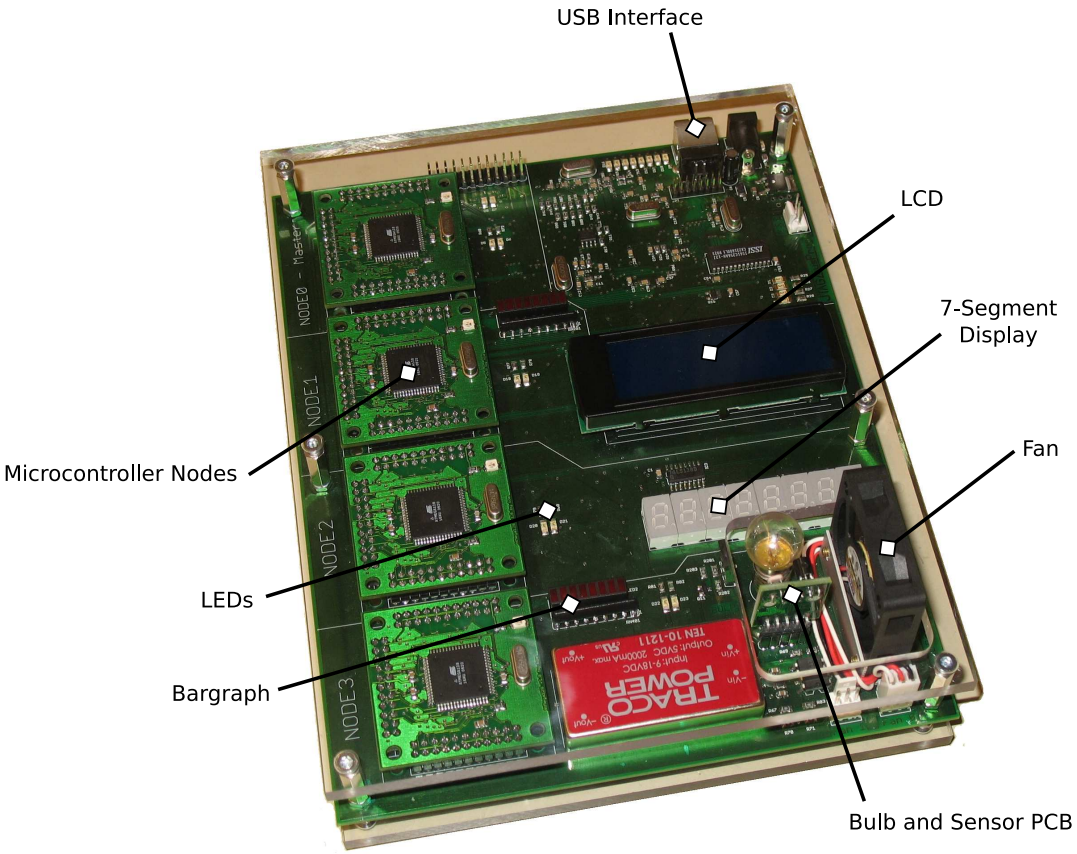


Figure 4.2: ESE-Board - Photography, source: [Koe09], modified illustration.

Receiver Transmitter). These USARTs are fully compatible with the AVR [Atm06] UART regarding baud rate generation, transmitter operation, transmit buffer functionality and receiver operation. The examples in this thesis, realized with the AVR tools, will only use the asynchronous mode of the USART, therefore they are depicted as UART. Following paragraphs describe only the parts of the nodes used for the examples in this thesis.

Node0

Node0 acts as the connector between the ESE-Board and an external device (usually an USB-wired desktop PC) and is connected to two LEDs. The UART1 block of the microcontroller is connected to the "USB to Serial" IC interfacing a possible external device.

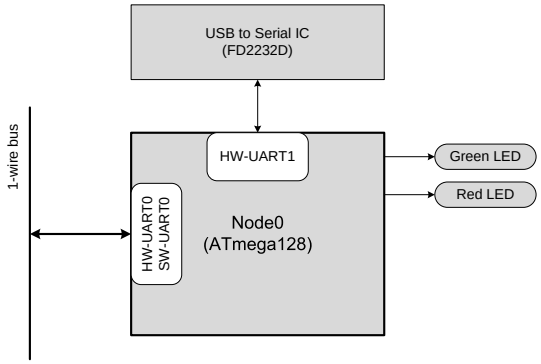


Figure 4.3: ESE-Board - Node0.

Node1

Node1 is responsible for controlling an LCD[2] device and a light bulb. The background light of the LCD is adjusted by PWM[3], the data I/O handled by SPI[4].

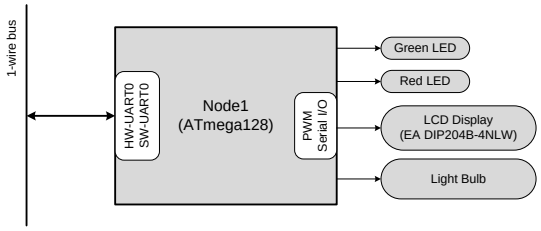


Figure 4.4: ESE-Board - Node1.

Node2

Node2 is connected to two LEDs.

Node3

Node3 controls a DC[5] fan (DA04010B12S-017), two LEDs, and acquires data from a temperature sensor (temperature sensitive Zener diode, National

[2]Liquid Crystal Display
[3]Pulse Width Modulation
[4]Serial Peripheral Interface
[5]Direct Current

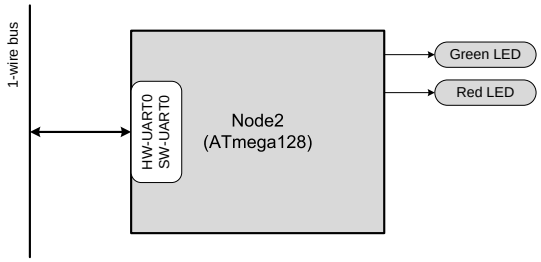


Figure 4.5: ESE-Board - Node2.

Semiconductor, LM135). The speed of the fan is controlled using PWM, the current RPM[6] are measured by observing the frequency generator output supplied by the fan (emitting two pulses per round). Temperature data values are converted via the ADC[7] unit of the microcontroller.

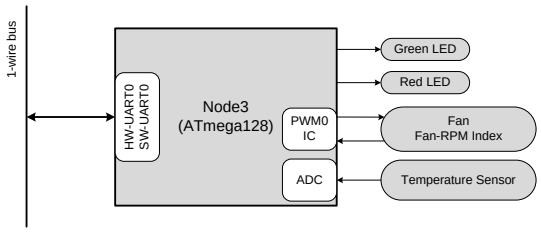


Figure 4.6: ESE-Board - Node3.

[6]Rounds per Minute
[7]Analog Digital Converter

4.2 Example Monoprocessor - A PID Controller with Scicos and SynDEx

This example exercises a development process with the combination of Scicos and SynDEx. A simple application is designed and run on a monoprocessor target. Node3 of the ESE-board will act as the execution environment of a digital PID control algorithm with anti-windup (confer to [Elm09, section 5.4.2]) that regulates the connected peripheral air fan. The fan (the plant) resides in the continuous domain (figure 4.7). The PID algorithm and the fan will be modeled and simulated with Scicos. After simulations, we demonstrate the temporal design of the PID algorithm and model it in SynDEx in order to support code generation for the hardware target.

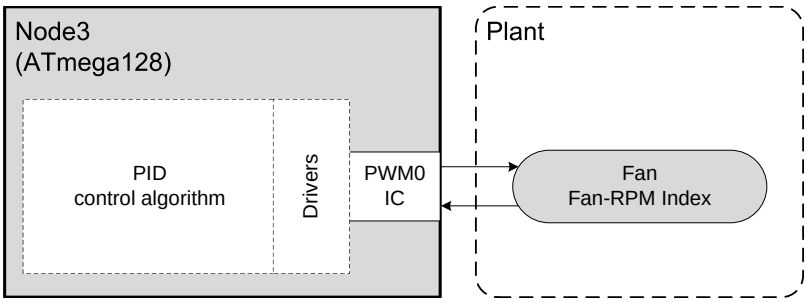


Figure 4.7: PID-Example: A PID algorithm placed on Node3 controlling the fan.

The peripherals connected to Node3 (an ATmega128 microcontroller) provide an excellent target to evaluate a PID control algorithm: The speed of the fan can be controlled by the PID algorithm through a customized fan driver, while speed data is obtained by a fan speed measurement driver.

The topic of designing a digital PID algorithm was chosen for several reasons: PID algorithms are very widely used control algorithms, have basic mathematics operations included, enforce the use of memory blocks for calculations, touch the topic of discrete PID implementation and their relation to continuous domain PID algorithms. Furthermore, the PID algorithm requires reading data from a sensor, the calculation of depending data, and writing of control values to an actuator. Through simulations of algorithm and architecture designs possible solution candidates for software algorithms and target hardware designs can be found. Latter is already realized by the ESE-Board hence the focus of this work lies mostly on software design.
Details about the target hardware architecture is found in section 4.2.1, while the software architecture, including a short requirements description and

the design of the models with their simulation and model transformations, is located in section 4.2.2. These sections include several comments and considerations which were posed during the development process.

The results of the example are helpful to reasonably answer following questions:

- How well is the Scicos/SynDEx framework suited to contribute to the development of a simple control algorithm?
- How much memory will the generated application use on the embedded target?
- How much effort is necessary (e.g. person hours) to develop such an application with the framework? Is the time spent reasonable? Does it need considerable time for a developer to understand/setup the framework?

The presented experiment will provide data allowing a reasonable statement about code size and quality for a typical embedded application.

4.2.1 Hardware Architecture (Execution Environment)

The chosen execution environment - plant and controller- consists of several ESE-Board components with following parts playing major roles:

- **Controller device:** Node3 (depicted in 4.7 and section 4.1.1).
- **Controlled peripherals:** The air fan connected to Node3 (actuator) takes the set fan speed as argument, and the fan sensor provides fan speed data. The air fan's input line is connected to the PWM0 output of the microcontroller, the speed index line to an input pin (IC - interrupt capture) of the microcontroller.

Note that this section contains the physical description of a target execution environment (hardware architecture), while the formally modeled hardware architecture is designed with SynDEx (following in this example). The hardware description could be done in more detail on the physical level, for example by modeling the hardware based on the blueprints of the board, but this is not intended for the design of the models at the chosen design abstraction level (keep the models simple). Simple models, dislodged from a very low level representation, are easier to understand and design, but at the same time they might hide details which could be crucial for designing a model representing the real world in adequate details.

4.2.2 Software Architecture

A digital PID control algorithm with anti-windup is designed in a generic way. This makes it possible to compare the results to other frameworks. For instance, the same algorithm could be implemented with the SCADE software in order to compare the results to this example with Scicos and SynDEx. Focusing on a simple example with only a PID algorithm leads to the idea of following rough requirements that just demand an air fan to be operated at a given speed.

Requirements

A fan (with a maximum of 10 000 RPM) is regulated by a digital PID control algorithm on a single processor. At system start, the speed of the fan is supplied with a constant percent value, for example a value of 100% results in 10 000 RPM, 50 % in 5 000 RPM, 0 % in 0 RPM. A speed tolerance of 15 % is acceptable, thus a speed input value of 50 % may result in $5\,000 \pm 750\,RPM$.

Considerations

Before going into solving the algorithmic problems, some economic factors have to be considered. Imagine oneself being in the role of the software company reading these requirements. Following thoughts could occur: There is a large variety of fans and control electronics available on the market. Which electronics equipment is the cheapest? How can such electronic components be evaluated in a fast way - is it possible to create a solution using these components? If necessary, would it be easily possible to replace a component by another model (e.g. the fan of choice could not be delivered anymore)?

A top-down model design approach using Scicos and SynDEx might lead to a working prototype. Scicos can be used to simulate the PID algorithm with a model of some different hardware fans in question. If the fan models used in the evaluation phase are considered to be accurate, the PID algorithm can be designed and simulated. Thus, the feasibility of a prototype can be confirmed or denied. Furthermore, information about necessary hardware components and algorithm designs can be gathered by including SynDEx in the design process. SynDEx can handle the temporal design of the application and automatically generate code for an electronic control logic of choice (e.g. an embedded microcontroller in this case).

Hybrid System Design and Verification

Obviously, the requirements describe a hybrid system: The fan is located in the real world with its continuous time and data characteristics. The design of the functional PID model and the fan model are implemented with Scicos. If the model of the real-world fan is implemented with sufficient accuracy, the parameters of the PID algorithm (that are located in a discrete data and time domain) can be calculated.

Scicos is a domain specific language and is used for modeling during the development and simulation phase. Scicos, as an internal DSL (which is based on Scilab), provides the means for hybrid systems modeling. Scicos models are meant to be platform-independent, thus their generic nature makes it possible to transform them into other artifacts or even executable models. Platform independence pertains mostly the controller algorithm part of the Scicos models - they are a part of the development which should be portable to different target hardware architectures. Whereas for the model of the plant (air fan) hardware independence is an issue which might not be realized easily. Every plant has different characteristics which need to be modeled in a sufficient level of detail - if only a very abstract representation of the plant is required, then the plant model could be considered to have some degree of platform independence - a real hardware fan could then be replaced by another model (or just a fan of the same model) without changing the plant model. The term *platform independence* might be seen not adequate for a plant model where no code will be executed on. Nevertheless, such models are needed for the simulation and verification of the controller part and moreover, their reuse in combination with various other plant instantiations would be of advantage.

PID Algorithm - Scicos Model

The Scicos model consists basically of two main components, the fan (plant) and the PID control algorithm (discrete controller). Both domains are connected via interface components (figure 4.8). Interface components are necessary since the drivers implemented on the microcontroller take integer values as arguments.

Before going towards the modeling of the plant and controller, some fundamental assumptions have to be made. In which detail should the fan be modeled? An ideal solution would be a model that represents real-world behavior. It might be common sense that an ideal solution is hardly possible, for several reasons, such as the fact that memory space and computation power are limited. This example does not focus on a sophisticated model, but on the methods and concepts in designing models with the Scicos/SynDEx framework.

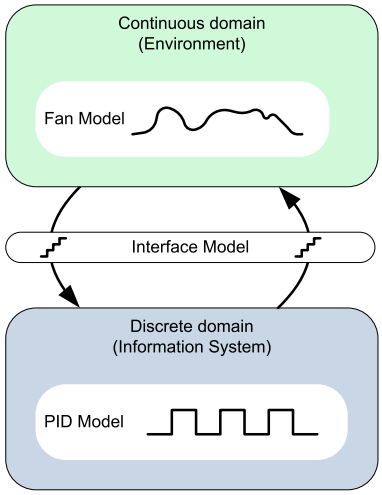
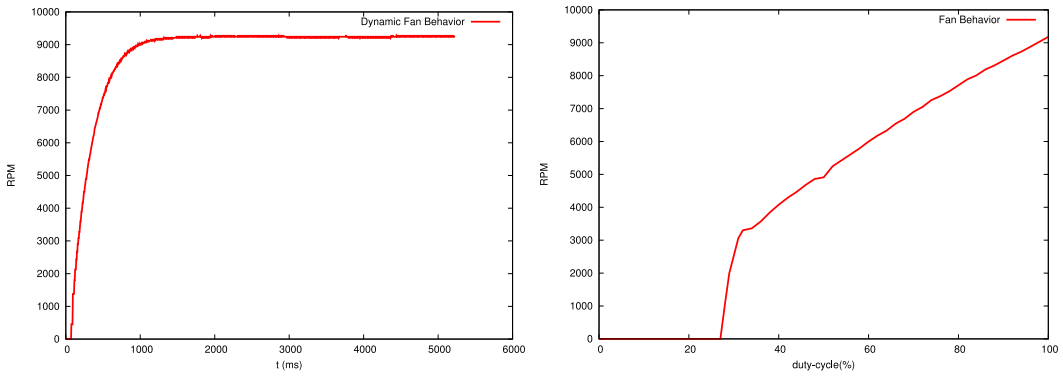


Figure 4.8: PID-Example - a hybrid system.

Plant Modeling

The behavior of the fan is modeled based on the dynamic and static behavior of the real hardware. The step response function (figure 4.9(a)) has to be combined with the PWM-RPM curve function (figure 4.9(b)).



(a) Step response.

(b) Static behavior.

Figure 4.9: Cooling fan behavior.

The PT1 behavior is approximately characterized by $x(s) = \frac{K_S}{1+s \cdot T_a}$ with $T_a = 330\ ms$ and $K_S = 0.925\ \frac{\%}{\%}$ (figure 4.10). The controller input is modeled as percentage of the maximum PWM input to the fan motor. The controller output is modeled as a percentage of the nominal maximum fan speed ($10\,000\ RPM$). Additionally, there is a deadtime of about $T_u = 75\ ms$,

modeled with $x(s) = \frac{1}{1+s \cdot T_u}$. The static behavior is a function correlating the set PWM duty-cycle (dc) and the fan's RPM (the signal dampening of about 7.75 % is already considered in the dynamic behavior):

$$f(x_{dc})\, RPM = \begin{cases} 0 & \text{if } x_{dc} < 28\% \\ PWM \cdot 9\,250 & \text{if } x_{dc} \geq 28\% \end{cases}$$

The corresponding Scicos model is depicted in (figure 4.11).

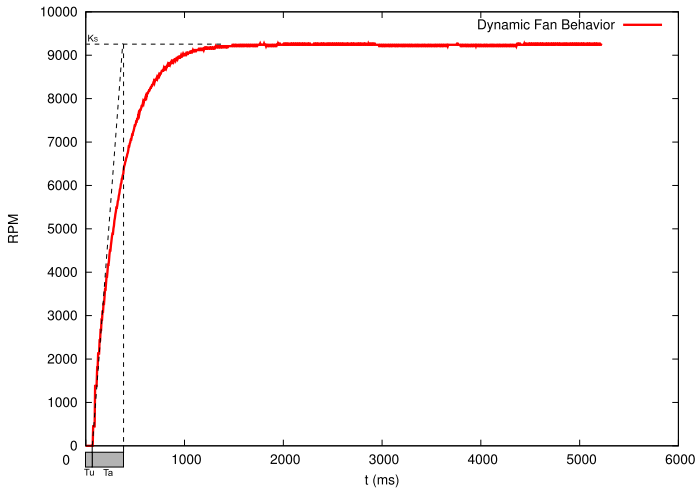


Figure 4.10: PID-Example - Approximation of the fan's PT1 and dead-time behavior.

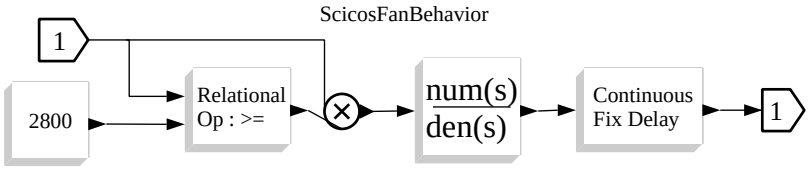


Figure 4.11: PID-Example - fan behavior Scicos model.

These characteristics are combined into a single Scicos model. The simulation results of the path are shown in figure 4.12.

Controller Modeling

The modeled controller is a typical discrete PID controller with anti-windup (figure 4.13). The anti-windup feature is necessary in order to avoid negative

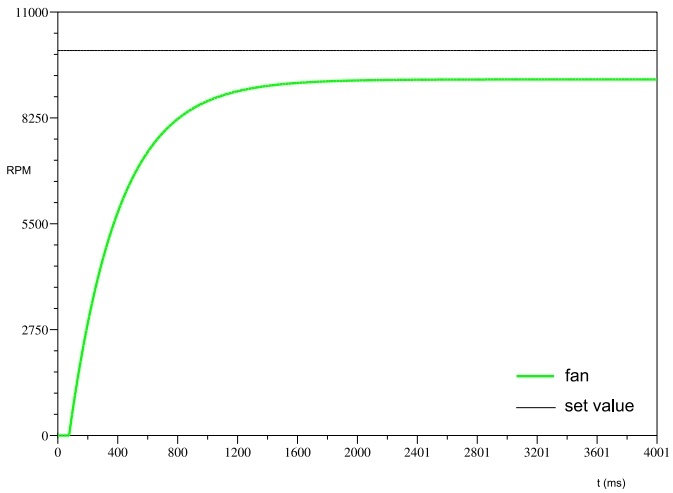


Figure 4.12: Fan, simulated Scicos model, $T_a = 330\,ms$, $T_u = 78\,ms$, $K_S = 0.925\frac{\%}{\%}$.

effects from the integrator component when the control output from the PID algorithm is pruned to the maximum output. The additional 1/Z block (memory) is inserted to avoid algebraic loops - it is triggered with a higher frequency than the rest of the algorithm.

Hybrid System Simulation

Controller and path are modeled as a closed loop and simulated. A top-level model shows the combination of discrete and continuous domain (figure 4.14). The PID controller takes the set point as argument and provides set values between 0 and 100. These values are quantized since the modular fan drivers take integers as arguments. The fan driver outputs PWM duty-cycle values that control the fan between 0 and 10 000 RPM (this is depicted by the following gain block). Now, in the continuous domain, calculated in Scicos with double values, the fan speed is measured with a fan sensor driver. The resulting RPM values are quantized since the modularly designed drivers provide only integer values. These values are scaled down by the factor 100 and forwarded to the modularly designed PID control block.

The system is simulated with the parameters K_P, K_I, K_S tuned according to the rules of Chien, Hrones and Reswick without overshoot (figure 4.15).

The simulation results, input/output values for the fan PID controller are

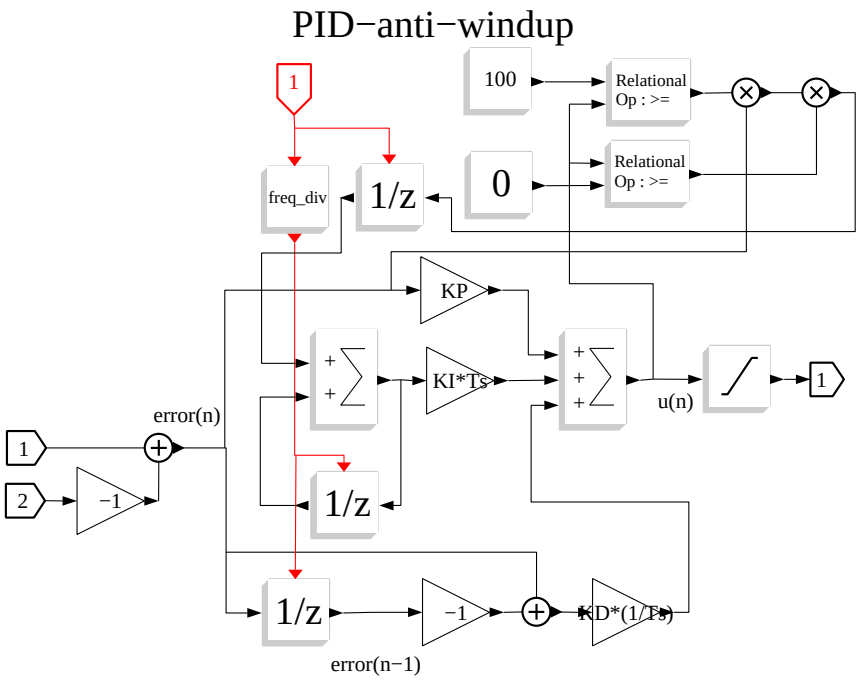


Figure 4.13: PID controller, Scicos model.

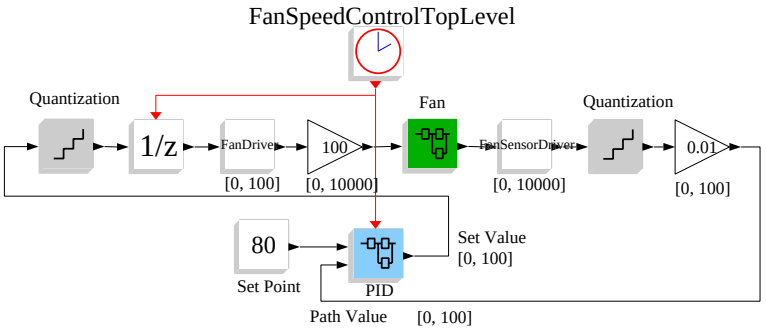


Figure 4.14: Plant and controller modeled in Scicos.

depicted in diagram 4.16). The differential weight factor is responsible for the abrupt changes in the set value.

PID Algorithm - Temporal Design and Automatic Implementation

After the Scicos simulation results were assumed being right, the Scicos

T_a		... effective compenstation time	=	$330\,ms$
T_u		... deadtime	=	$78\,ms$
K_S		... gain	=	$0.925\,\frac{\%}{\%}$
K_P	=	$\frac{0.6 \cdot T_a}{K_S \cdot Tu}$	=	$2.74\,\frac{\%}{\%}$
K_I	=	$\frac{K_P}{T_n}$	=	$0.00832\,\frac{\%}{\%}$
K_D	=	$K_P \cdot T_v$	=	$107.02703\,\frac{\%}{\%}$
T_n	=	T_a	=	$330\,ms$
T_v	=	$0.5 \cdot T_u$	=	$39\,ms$
S_P		... set-point	=	$80\frac{\%}{\%}$
T_S		... sampling time	=	$5\,ms$

Figure 4.15: PID control algorithm, parameters.

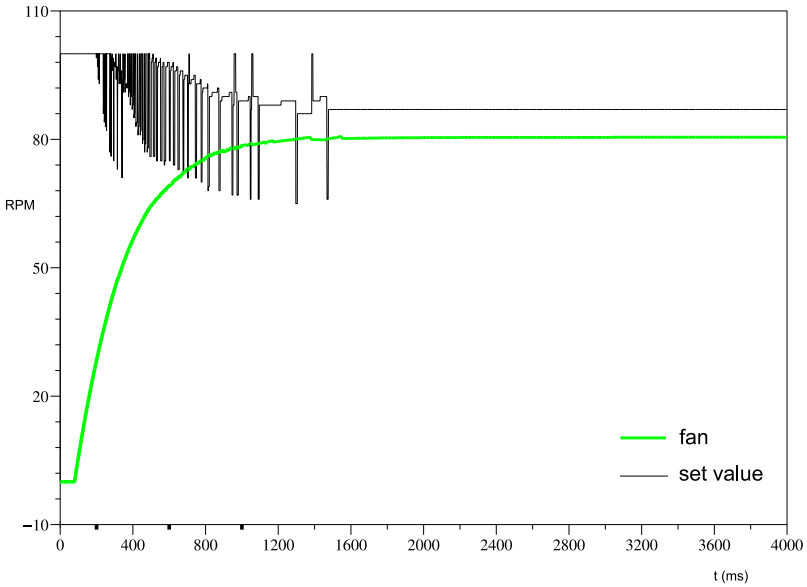


Figure 4.16: Plant and controller simulated in Scicos, Input/Output of the fan (continuous domain).

model is transformed to a SynDEx model via the Scicos-To-SynDEx gateway (a model-to-model transformation). SynDEx is then used to design the temporal requirements (the PID control algorithm is triggered every 5 ms), model the hardware architecture (and execution environment), simulate these together with the PID algorithm model and then automatically generate a macro code representation of the model (see figure 4.17). After that, the macro code model is expanded into an AVR-GCC compliant C code by the usage of the GNU M4 macro processor[8] in conjunction with the target-dependent macro expansion

[8] http://www.gnu.org/software/m4

files, which are consisting of rule-sets for expanding the macro code to C.

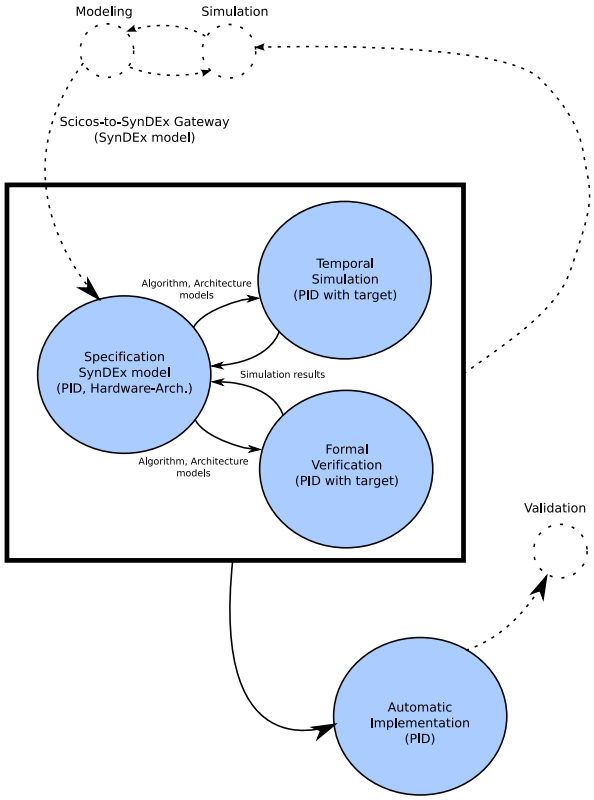


Figure 4.17: Partial software development process, PID-Example. The Scicos model is transformed into a SynDEx model acting as system specification followed by temporal simulations, formal verification, and automated code generation.

The SynDEx set of models for this example consists basically of two main models, the *hardware architecture* defining a model of the target hardware, and the *algorithm* defining a model of the PID algorithm in connection with the fan drivers.

Designing a SynDEx hardware architecture model. SynDEx requires a target hardware architecture being described by a model, for this purpose an ATmega128 microcontroller SynDEx block is constructed. The ATmega128 block is part of a customized Atmel SynDEx-library. Since a microcontroller on the target hardware can be connected to the bus and enable a communication line to the other nodes on the ESE-board, a communication gate (c) is added to the model. A node could also be connected to an external PC device via a Serial-To-USB circuit, therefore an additional communication gate (e) was

added. Since this is a monoprocessor example, the gates will not be used here. The model is depicted in figure 4.18 and the corresponding SynDEx code is listed here (file Atmel.sdx):

```
1 syndex_version : "7.0.0"
  application description : ""

  # Libraries

6 # Algorithms

  def operator ATmega128 :
  gate COMA c;
  gate COMB e;
11 code_phases:  loopseq initseq endseq;

  # Main Algorithm / Main Architecture

  # Extra durations
16
  # Software components

  # Constraints
```

Listing 4.1: ESE-Board - ATmega128 Node - SynDEx-Architecture

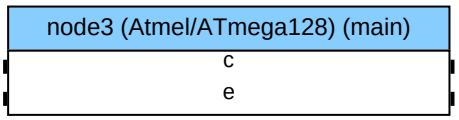


Figure 4.18: Node3, SynDEx architecture model.

Design and parametrization of the SynDEx algorithm model. The top-level SynDEx algorithm model is depicted in figure 4.19. The figure depicts a block for the control of the timer2 unit of the ATmega128 microcontroller. This block takes parameters that determine when to fire a timer event and activates the following top-level hierarchy block. Latter contains the PID controller algorithm connected to the fan drivers. One reason for choosing this solution was the lack of sufficient computation power of the used development system: If the period of a task is much higher than that of other tasks, the resulting scheduling table will be too large to display, too large to be sufficiently readable by the developer, and additionally tricky scheduling algorithms need to be added to the code generation part if such a scheduling needs to be implemented in the product. This solution, using the timer2 controller block and the following top-level time-triggered (confer to [EBK03]) block burns time every micro-period for the sake of scheduling. The price for improved scheduling, readability and an implementation without high effort scheduling code, is thus spending periodic time. A second reason, of no lesser importance, is the fact,

that a constant time instance for a periodic task is needed. If there was no timer, there would be a, more or less random, shift of the execution of the time-critical blocks earlier in time, because all time blocks in SynDEx are considered to be atomic and WCET - in fact, tasks may be finished before their specified WCET, pulling the execution of the periodic task forward in time.

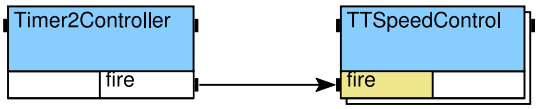


Figure 4.19: PID-Example. Top-Level SynDEx model.

The timer2 (time-triggered) block activates the fan sensor block, the PID controller block and the fan driver (see figure 4.20). The PID block was converted from the Scicos model into the SynDEx model and therefore its data types are double. Fan driver and fan sensor driver blocks are of the types uint8 and uint16 leading to the need of an intermediate interface layer wrapping the types: S2SWrapper-blocks convert double to uint8, uint8 to double and uint16 to double data types, which could mean a loss of information, but since the Scicos model was implemented by using quantization blocks (thus interfacing the real fans with integer values), these conversions make sense.

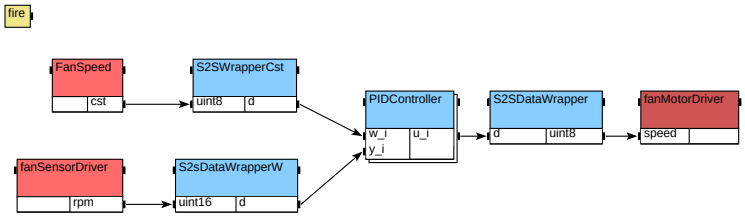


Figure 4.20: PID-Example. SynDEx model, Scicos-transformed PID algorithm interfacing fan sensor and fan drivers.

Finally, on the bottom level of the hierarchy, the PID algorithm, which was converted from a Scicos model into a SynDEx model, is shown in figure 4.21. The algorithm takes the set point w_i and the set value y_i as inputs and outputs the path value u_i. Note that all the blocks contained in this PID controller are bearing the data structure overhead from the Scicos model: The ability to simulate and change, automatically transform models between Scicos and

SynDEx introduces additional data memory costs on the target platform.

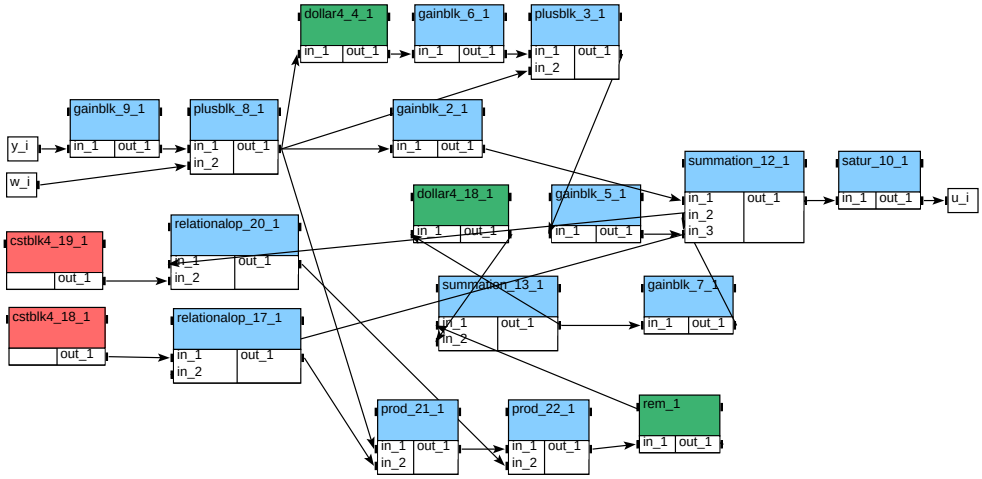


Figure 4.21: PID-Example. SynDEx model of the PID controller transformed from the Scicos PID model.

PID Algorithm - Scheduling of the SynDEx model

SynDEx takes the algorithm model and schedules the blocks by using a greedy algorithm under provided time constraints. The entire algorithm is supposed to be triggered about every 5 ms (even a Sampling Time of 8.5 ms was calculated, however the real-world properties make the fine-tuning of a PID almost always necessary) and this is accomplished basically in two steps:

- **Define the periods of each task** (similar to the Liu and Layland task model).
  The ATmega128 is clocked with a frequency of 14.745 6 MHz, that means 1 ms takes 14745.6 CPU cycles, 5 ms take 73 728 CPU cycles. The question raised now is how exactly the period must be chosen to meet the requirements, since this is a single periodic example. A period chosen of 74 STU will not interfere with the schedulability of other tasks (since there are no other tasks which periods would have to be n-times the period of this task). To keep it simple, the trigger time of the algorithm is modeled with 80 $STU = 5.42\,ms$, however the STUs are chosen, they will mostly imply an inaccuracy between model and real world (depends also on the clock of the MCU).

- **Define the durations of each task.**
  To keep the model human-readable, a standardized, constant duration for each block is chosen: 1 000 *cycles* of the ATmega128 shall represent 1 *STU* (SynDEx Time Unit in the model - this unit is made up for a better nomenclature). All the blocks in this example are executed a lot faster than that time (WCET), leading into an investment of resources into safety.

After SynDEx runs through the algorithm using the AAA methodology, a resulting scheduling table can be investigated (see figure 4.22). On the left hand side on the processor schedule, there is the timed controller running at the start of each CPU-period, followed by the main algorithm. After that, a wait statement is inserted to fill up the intended period of 80 *STU*. The right hand side is a time reservation slot which will be executed if the timer controller has not fired. If a task has to be executed every 3 *s* and another one every 5 *ms*, the costs for the calculation of the scheduling will be enormous, resulting in an obfuscated model (provided the calculating time of the model is smaller than the developers patience/time available). Therefore, we reserve time (wait the WCET of the whole algorithm) whenever the timer does not fire in this CPU period.

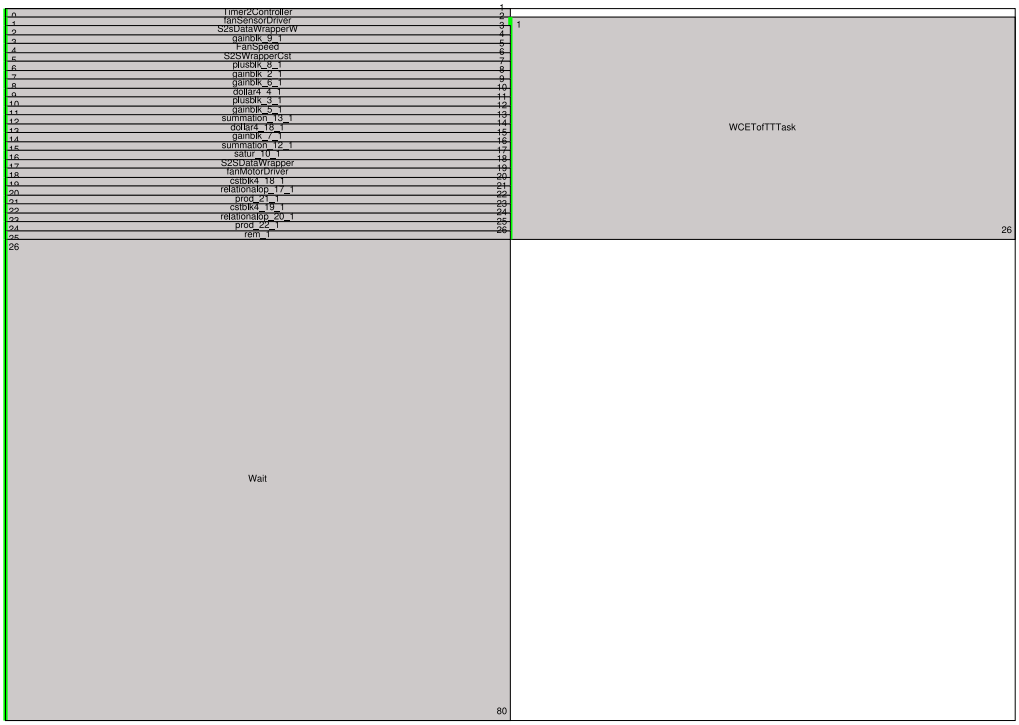


Figure 4.22: PID-Example. SynDEx scheduling after algorithm adequation.

PID Algorithm - Macro Code and Executive Code Generation

At this step the development process is at a stage where it is possible to generate a target language-independent macro code file for each involved node. Using target language-dependent macro definitions these macro code files are then translated into avr-gcc compliant C code which afterwards can finally be compiled, downloaded and executed on the hardware.

A look at the generated macro code explains what is needed to realize these last steps (node3.m4, listing B.1). In this file, after allocating global memory for variables, the *main_* function tag is immediately followed by an *proc_init_* tag. Latter tag is used to switch a flag in the macro code - this way a macro is then expanded to its corresponding section in the target-dependent definition file, thus the C code for the initialization for the block (if there is any) is inserted. An example of a macro expansion definition for the pid_gainblock is listed in the appendix (see listing B.2): The generated C code depends on the code phase in the node3.m4 algorithm file. After all initializations of the blocks are handled, the main loop of the algorithm expands to an infinite loop (by using the definition in a processor (ATmega128) specific language macro definition file, see listing B.3). In this main loop all the blocks are expanded by their corresponding main loop phase. After finishing the main loop, blocks are expanded according to an end phase. For example, the latter can be used to free resources. The resulting C code is listed in the appendix B.4 which can then be compiled by using the generated makefiles (see listing B.5 and listing B.6). The executable was downloaded onto the ESE-Board and the fan run approximately at the set speed.

A note about data types. It is possible to automatically map the data types to compiler conforming ones by adding rules to the target language expansion file (e.g. ATmega128.m4x). For example, the type *int* is mapped to *int16_t* without a redesign of the model. This is useful to support the use of different compilers or libraries, and adds value to the re-usability of models.

The interface data types of the modeled SynDEx blocks are defined by an identifier and the data type size (bytes):

```
typedef_(‘uint16’,2)
```

New data types can be defined:

```
typedef_(‘uint16_t’, 2)
```

The new data types can be mapped (note that also those which are not mapped to a new type need a mapping definition):

```
define(‘uint16_map’,‘uint16_t’)
```

SynDEx will create all data types with a ”_map” appendix when the ”basicAlloc_” definition is edited as follows:

```
define(‘basicAlloc_’,‘_($1_type_()_map() $1[$1_size_];)’)
```

PID Algorithm Example - Validation

Simulations of the hybrid system, temporal design and algorithm optimization were done. The question at this point in the development process, is if the models and simulations are a good representation of the life behavior. The algorithm was deployed onto the target architecture and some monitoring code inserted. The velocity data of the fan was sent via UART to node0, which itself forwards all the communication data via USB to the development PC. The collection of velocity data on the target did not interfere with the PID code running on it, since the UART baud rate and communication calls were designed carefully with the help of the SynDEx scheduling model: The *Wait* time interval in the scheduling model was used for communication purposes.

Figure 4.23 shows the real behavior of the PID controlling the fan. The fan was running approximately at a speed of 80 % for some seconds. The life behavior differs from the simulated behavior - there might be several reasons for that, however, they are no object of further investigations right here. A PI controller might be more suitable for this situation.

Comparing these live data (figure 4.23) to the Scicos simulation results (figure 4.16) shows different behavior of model and real world. Because of the simple plant model, as well as its roughly calculated parameters, the live data differs. The high fluctuations are explained by the K_D factor and the small fan speed spikes provided by the sensor. If the model was required to be more accurate, a re-modeling with Scicos/SynDEx would be necessary (figure 4.24). This example escapes the software development cycle with Scicos/SynDEx at this point.

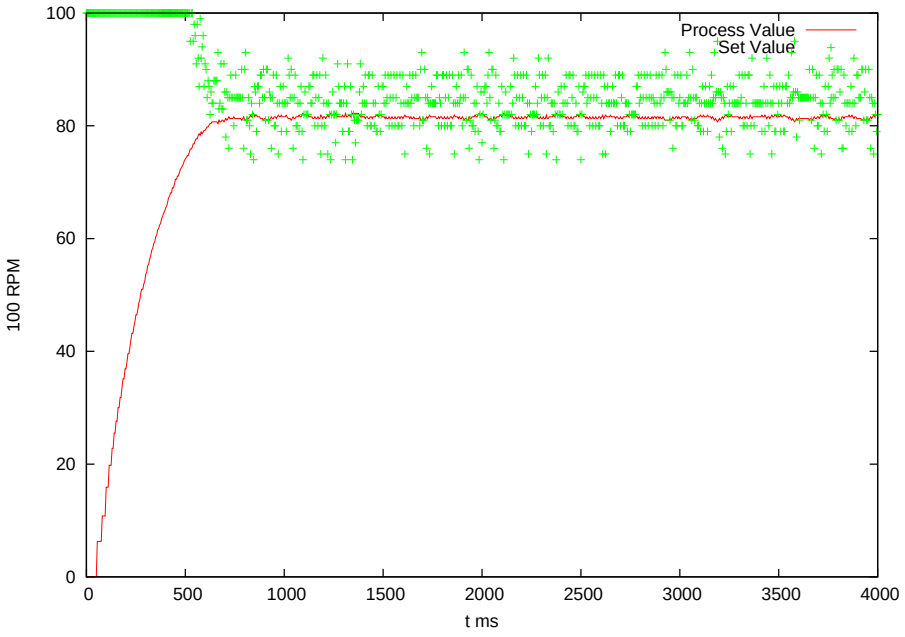


Figure 4.23: PID-Example. Live data diagram of the PID controller and the fan.

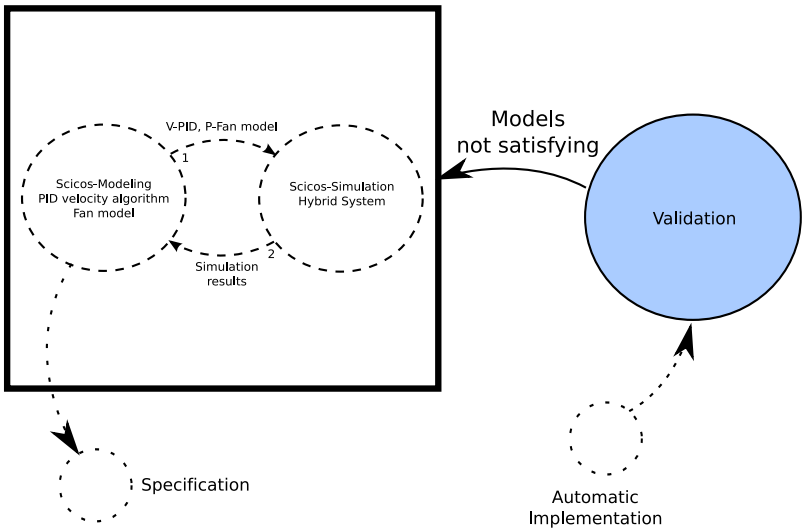


Figure 4.24: PID-Example. Model redesign and new parameters in case of unsatisfying results.

4.3 Example Multiprocessor - Data Observation and Communication with SynDEx

The aim of this example is to model a multiprocessor application in SynDEx with the ESE-Board as target hardware. Temperature values are periodically acquired at node3. These temperature values will be sent to node1 and displayed on its LCD peripheral. Furthermore, the temperature data is sent via node0 (acting as a repeater) to the development PC and displayed there (figure 4.25).

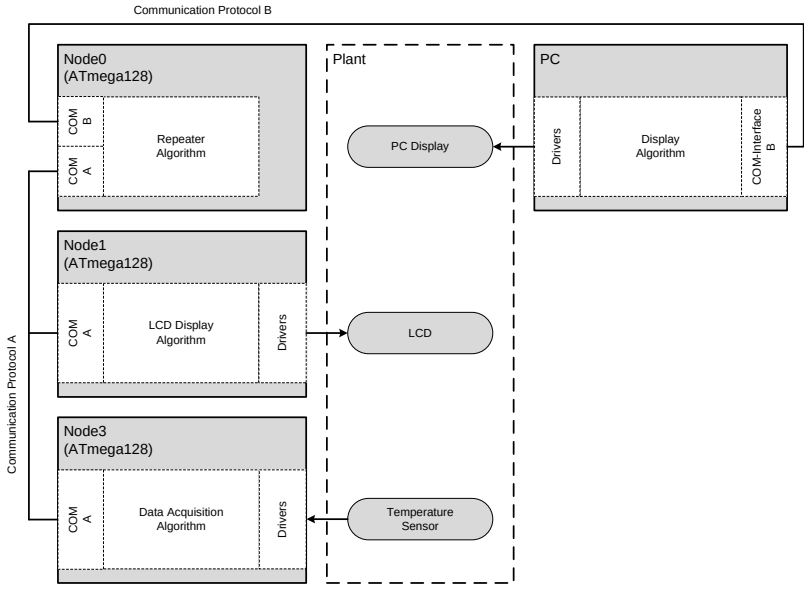


Figure 4.25: Multiprocessor-Example. Distribution and display of temperature information.

Expected results of this example are statements about the feasibility and scope of using SynDEx models on a microcontroller multiprocessor architecture.

4.3.1 Hardware Architecture (Execution Environment)

The execution environment consists of several ESE-Board components:

- **Microcontroller device:** Node0 - Receive temperature data and forward it to the Desktop PC.
- **Microcontroller device:** Node1 - Receive temperature data and control LCD peripheral.
- **Microcontroller device:** Node3 - Sample temperature data and send it to Node0 and Node1.

- **Microcontroller peripheral** Node1: The liquid crystal display for presenting temperature values.
- **Microcontroller peripheral** Node3: The temperature sensor for data sampling.
- **Desktop computer device:** PC - display temperature data received.

4.3.2 Software Architecture

The principle of this example is to realize an application where information is gathered periodically and distributed within a pre-determined timespan to other targets. In terms of an industrial application, this example could be described like a car-application: The break pedal state is checked regularly and its state sent to the brake system within one millisecond.

Requirements

Temperature values are measured periodically with 20 *STU* and displayed by the LCD display on Node1. Furthermore the measured data is sent to the PC at the same time period. The temperature data measured has to be sent to the LCD within a time of 15 *STU*.

SynDEx - Hardware architecture model

The hardware architecture model designed for this example includes four microcontroller nodes (node0, node1, node2, node3) connected by communication medium "comA", and one desktop PC connected to node0 by communication medium "comB" (figure 4.26). Every communication medium is defined as SAM multipoint without broadcast.

SynDEx - Algorithm model

The algorithm model (figure 4.27) consists of several interacting blocks. Located on Node3, a timer block (*TemperatureTimer2Controller*) periodically triggers the temperature acquisition blocks (*TTTTemperature*; returns 0 when not fired, otherwise the temperature) is the first instance in the algorithm. After the temperature data is gathered, it is sent to the *Repeater* block on Node0 which forwards it to the *PCDisplay* block. The data returned from the *TTTTemperature* block is passed to the *LCDShow* block located on Node1. *NoSpeedValue* is supplied to the *LCDShow* block if there are no data available.

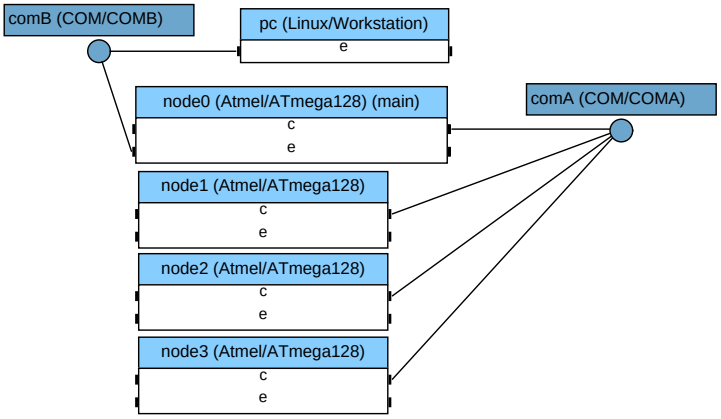


Figure 4.26: SynDEx architecture model.

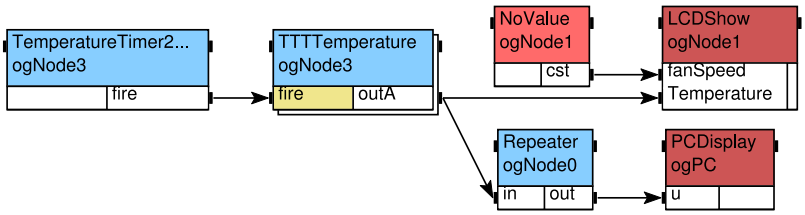


Figure 4.27: A SynDEx algorithm model with four operators.

All blocks in the model have a defined period of 20STU, the durations of the operator-blocks were defined with very high values for the sake of the readability of the model (otherwise the fraction of communication-durations/operator-block-durations might be very high). The resulting SynDEx scheduling of the algorithm can be found in figure 4.28 and a brief explanation what happens in this scheduling table is given here:

- **node0 (a)**. The *Repeater* block is executed after the temperature data from block *CondO0* arrived via communication medium *comA*.
- **node1 (b)**. Initially a no-value block is executed followed by a wait statement until temperature data is received on *comA* from *CondO0* on node3. With the data available, information is displayed on the *LCDShow* block. This node holds also the conceptual trigger for the next inter-repetition synchronization (b1) - it is supposed to hold an additional wait statement to fill out the period of 20 STU (note: in the generated m4 macro code no such statements could be found).
- **node2**. This node is not used for this example.
- **node3**. The timer controller block triggers the temperature sensor block

on time.

- **pc (c)**. The *PCDisplay* block gets data from the *Repeater* block via *comB*.
- **comA**. The occupation of the communication channel is displayed. First, data is sent from *CondO0* (node3) to *Repeater* (node0). Second, temperature data from *CondO0* (node3) is sent to *PCDisplay* (PC).
- **comB**. The *Repeater* (node0) sends data to the *PCDisplay* (PC) using communication protocol B.

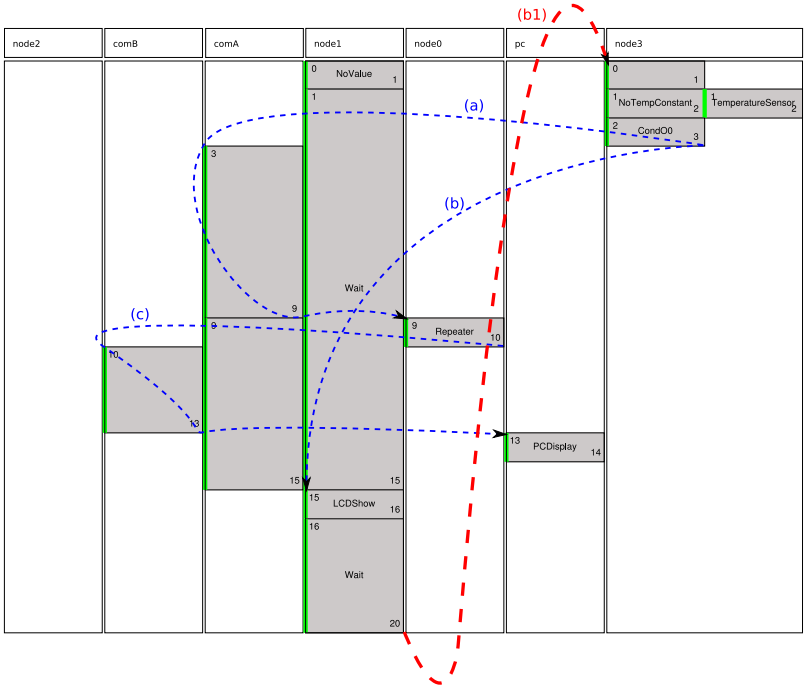


Figure 4.28: SynDEx scheduling for the multiprocessor example.

SynDEx - Macro Code Structure and Communication

The generated M4 macro code consists of a main loop surrounded by initialize and finalize placeholders for each block. Every communication medium is realized with a separate thread running concurrently to the main loop. The challenge here is to make this code structure fit for a target without supported threads. Therefore a customized program is built (M4-Builder) to transform these threads to send/receive functions which can be called from within the main-loop blocks (figure 4.29).

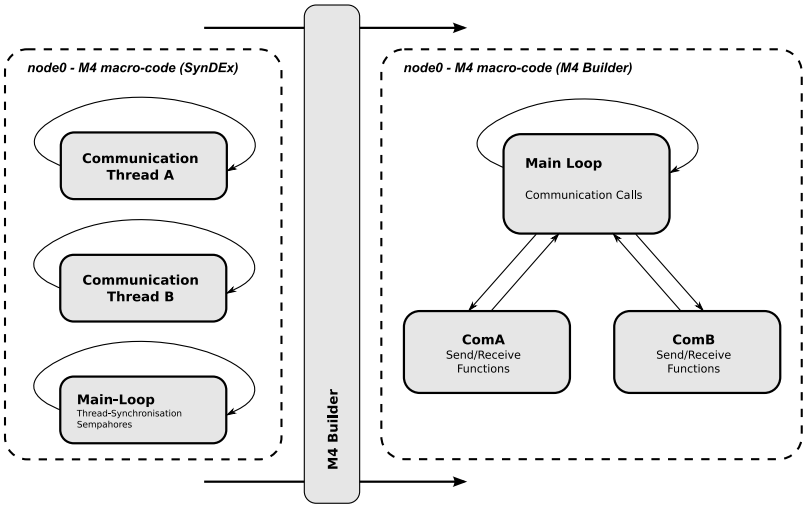


Figure 4.29: Re-structuring of the SynDEx M4 code.

Consider listing 4.2 which is a portion of the M4 code generated for node0. A thread is assigned to every communication medium: thread_(COMA,c,node0,node1,node2,node3) for communication medium comA and thread_(COMB, e, node0, PC) for comB: The parameters defined are the communication medium name, the communication gate (c, e) and the nodes connected by it (e.g. node0 and the PC are connected by comB). Each communication thread holds some semaphores responsible for maintaining the right order of the block execution (Pre0, Suc0, Pre1, Suc1). For example, node0 might only send data to the PC if the repeater block is filled with data.

```
thread_(COMA,c,node0,node1,node2,node3)
  loop_
    Suc1_(_comMultiProcessor_TTTTemperature_CondO0_o_node0_c_empty,,
        _comMultiProcessor_TTTTemperature_CondO0_o,empty)
    recv_(_comMultiProcessor_TTTTemperature_CondO0_o,ATmega128,node3,node0)
    Pre0_(_comMultiProcessor_TTTTemperature_CondO0_o_node0_c_full,,
        _comMultiProcessor_TTTTemperature_CondO0_o,full)
  endloop_
endthread_

thread_(COMB,e,node0,pc)
  Pre0_(_comMultiProcessor_Repeater_out_node0_e_empty,,_comMultiProcessor_Repeater_out,
      empty)
  loop_
    Suc1_(_comMultiProcessor_Repeater_out_node0_e_full,,_comMultiProcessor_Repeater_out,
        full)
    send_(_comMultiProcessor_Repeater_out,ATmega128,node0,pc)
    Pre0_(_comMultiProcessor_Repeater_out_node0_e_empty,,_comMultiProcessor_Repeater_out,
        empty)
  endloop_
endthread_

main_
  proc_init_
    spawn_thread_(c)
  spawn_thread_(e)
  eBoardDrivers_eRepeater(_comMultiProcessor_TTTTemperature_CondO0_o,
      _comMultiProcessor_Repeater_out)
  Pre1_(_comMultiProcessor_TTTTemperature_CondO0_o_node0_c_empty,c,
      _comMultiProcessor_TTTTemperature_CondO0_o,empty)
  loop_
    Suc0_(_comMultiProcessor_TTTTemperature_CondO0_o_node0_c_full,c,
        _comMultiProcessor_TTTTemperature_CondO0_o,full)
```

```
26      Suc0_(_comMultiProcessor_Repeater_out_node0_e_empty,e,_comMultiProcessor_Repeater_out,
            empty)
        eBoardDrivers_eRepeater(_comMultiProcessor_TTTTemperature_CondO0_o,
            _comMultiProcessor_Repeater_out)
        Pre1_(_comMultiProcessor_Repeater_out_node0_e_full,e,_comMultiProcessor_Repeater_out,
            full)
        Pre1_(_comMultiProcessor_TTTTemperature_CondO0_o_node0_c_empty,c,
            _comMultiProcessor_TTTTemperature_CondO0_o,empty)
    endloop_
31  eBoardDrivers_eRepeater(_comMultiProcessor_TTTTemperature_CondO0_o,
        _comMultiProcessor_Repeater_out)
    wait_endthread_(Semaphore_Thread_c)
    wait_endthread_(Semaphore_Thread_e)
    proc_end_
endmain_
```

Listing 4.2: SynDEx communication threads

With the m4Builder the communication threads are transformed to communication calls (see listing 4.3 for the transformed node0 macro code) while the blocking behavior (e.g. block until data was received or sent) lies in the responsibility of the communication protocol design. The blocking behavior is necessary to maintain the synchronization within the whole algorithm. Without it, data packets would most probably be missed and the display of the scheduling would differ more from the real-world execution. As seen, every communication medium thread becomes a function with a switch block providing the choice of operation to be done. Semaphores were removed and in the main loop replaced by appropriate communication functions calls. Note that this transformation **changes the scheduling table** and is only allowed if there are no subsequent operations following on the processors during the communication duration.

```
#define   _comMultiProcessor_Repeater_out_node0_e_full  0
#define   _comMultiProcessor_TTTTemperature_CondO0_o_node0_c_full  1

4 alloc_(uint16,_comMultiProcessor_TTTTemperature_CondO0_o,1)
  alloc_(uint16,_comMultiProcessor_Repeater_out,1)

  void comCOMA(uint8_t comActions){
    switch(comActions){
9   case _comMultiProcessor_TTTTemperature_CondO0_o_node0_c_full:
      recv_(_comMultiProcessor_TTTTemperature_CondO0_o,ATmega128,node3,node0)
    break;
    default:
      /* never reached */
14  break;
    } /* end Switch */
  } /* end comNAME */

  void comCOMB(uint8_t comActions){
19  switch(comActions){
    case _comMultiProcessor_Repeater_out_node0_e_full:
      send_(_comMultiProcessor_Repeater_out,ATmega128,node0,pc)
    break;
    default:
24    /* never reached */
    break;
    } /* end Switch */
  } /* end comNAME */

29 main_
    proc_init_
    eBoardDrivers_eRepeater(_comMultiProcessor_TTTTemperature_CondO0_o,
        _comMultiProcessor_Repeater_out)
    loop_
      comCOMA(_comMultiProcessor_TTTTemperature_CondO0_o_node0_c_full)
34    eBoardDrivers_eRepeater(_comMultiProcessor_TTTTemperature_CondO0_o,
          _comMultiProcessor_Repeater_out)
      comCOMB(_comMultiProcessor_Repeater_out_node0_e_full)
    endloop_
```

```
    eBoardDrivers_eRepeater(_comMultiProcessor_TTTTemperature_CondO0_o,
        _comMultiProcessor_Repeater_out)
    proc_end_
39 endmain_
```

Listing 4.3: SynDEx communication functions

SynDEx - Communication Protocol Design

The nature of the macro code allows the realization and the replacement of communication protocols. A protocol which does not violate the visualized scheduling model too much needs certain characteristics. Every send or receive operation must be blocking to maintain the synchronized integrity of the whole algorithm. If it was not blocking, communication would occur randomly and the data integrity would not be given. For example, if node1 is busy with executing the display operations and node 3 sends data to node1 at the same moment, the data would be lost. Another model violation would be a phase shift of the single scheduling between the nodes - that might even be acceptable for some applications. How the communication protocol works internally, like using handshakes or checksums is free of choice. One thing the protocol does need is an acknowledge mechanism telling the sender that the data was successfully received and it can proceed with its next operation.

Each communication transmission must contain an addressing part. Therefore it is possible for the receiving node to verify if the packet is meant to be read from the communication medium. Again, this is necessary because of possible early executions of tasks which could result in two nodes trying to send at the same time which could result in data collisions. Depending if and how bus collisions are handled in this case determines which packet arrives first at the receiver. Imagine two nodes sending temperature values at the same time to the same node and a bus arbitration decides which temperature data will arrive before the other one. If all blocks would have the same real-world behavior as modeled, then addressing would not be necessary leading to a slimmer protocol.

Necessary characteristics of the protocol:

- Blocking. Send and Receive operations must block to maintain synchronization within the algorithm.
- Finished transmission notification. Terminate the communication procedure by acknowledging a successfully received transmission - the sender may continue on its schedule.
- Addressing. Identify receiving and sending nodes, validation of the received data.

Next to the mandatory characteristics of the protocol, a good time representation has to be found. Since it has been pointed out that it is hard to maintain a good cognitivity of the model if the execution times of the blocks differ by a magnitude of 10^2, it might be hard to combine fast executed blocks, which take for example just 50 microcontroller cycles, with a protocol having a baud rate of 57 600 bits/sec. An approach to handle this is to reserve more time for every fast block for the price of performance.

4.4 Results

Several aspects about this conceptually model-driven development method resulted from the examples: How a development process is performed by concrete examples, how much space this particular model-driven approach requires on the ESE-Board target architecture in comparison to hand-written code, statements about the complexity and cognitivity of the macro code structure, and the Scicos/SynDEx models with their differences to real behavior. These results are presented and some statements about modeling with Scicos/SynDEx in combination with a microcontroller-based distributed embedded system are given.

4.4.1 Code Size

The executable code (an ELF file) was analyzed with avr-objdump, avr-size and a memory evaluation library (*MemEval*) [Elm10] for an approximation of the used dynamic memory: At program start, a bit-pattern is written onto the SRAM between the end of the .bss-section and the stack-pointer. After dynamic memory allocations, the SRAM area is searched for the largest untouched bit-pattern and its size returned/displayed. The focus of the measurement lies on the overhead produced by Scicos blocks - therefore the heap and stack memory consumptions before the main-loop are of interest. The main-loop consists basically just of Scicos block calls which have a clear structure and are comparable to hand-written code calling components.

A pseudo-code clarification how the dynamic memory was estimated is listed in algorithm 1, while the memory usage is found in table 4.1 (note: The listed dynamic memory usage applies not for the dynamic memory consumption of the whole algorithm - it describes only the dynamic memory usage of the introduced Scicos blocks). For comparison, a rough, hand-written PID algorithm which uses modularly designed drivers like in the SynDEx example, could use about 5 000 *bytes* Flash and 200 *bytes* static memory.

Memory type	Memory total (*bytes*)	Memory used (*bytes*)	
		Scicos/SynDEx	**Hand-written**
Flash	131 072	34 796	4 922
SRAM (static)	4 096	976	226
SRAM (dynamic)		1 596	

Table 4.1: Memory usage measured in the monoprocessor examples.

The ELF file was produced by avr-gcc (version 4.2.2) and avr-ld (version 2.17) with the optimization for code size flag (CFLAGS= -Os). The static

Algorithm 1 The concept of memory measurement

```
1: Includes, Defines.
2: Prototypes, Functions, Globals.
3: main(){
4: MemEval → Initialize memory.
5: Allocate Scicos blocks (includes dynamic memory allocations).
6: MemEval → Analyze memory.
7: MemEval → Return largest unused memory area.
8: loop
9:    run PID algorithm.
10: end loop
11: }
```

memory usage is calculated by adding the .data and .bss section sizes (e.g. displayed by an avr-objdump of the ELF file), while the dynamic memory usage is determined by subtracting the unused memory and static memory from the total SRAM capacity.

$$\begin{aligned} Flash &= .text + .data \\ &= 34\,024 \;+\; 772 \;=\; 34\,796\, bytes. \\ SRAM(static) &= .data + .bss \\ &= 772 \;+\; 204 \;=\; 976\, bytes. \\ SRAM(dynamic) &= SRAM(total) - MemEval(unused\, area) - SRAM(static) \\ &= 4\,096 - 1\,524 - 976 \;=\; 1\,596\, bytes. \end{aligned}$$

4.4.2 Code Structure - Differences between automatically generated and hand-written code

In general, hand-written code can have any structure possible. Latter was constructed here in respect to reusability and modular design of the components: Every functionality is designed by a module with appropriate interfaces including at least an init, main, and a end (finalize) function. These modules are used for automatically generated and hand-written applications.

Three different situations for implementing applications are compared: Scicos/SynDEx, only SynDEx, and a pure hand-written implementation. A design

Development setup	Scicos blocks	SynDEx auxiliary functions/globals	Cyclomatic complexity	Include customized modules (drivers)
Scicos/SynDEx	√	√	cc_{SynDEx}	√
SynDEx	¬	√	$cc_{SynDEx} = cc_{HW} + cc_{AuxF}$	√
Hand-written	¬	¬	cc_{HW}	√

Table 4.2: Code structure comparison.

with Scicos/SynDEx results in a scicos_block structure generated for every designed block (see listing B.7). This is necessary to maintain the simulation capability in Scicos. Furthermore SynDEx generates a global variable for each output of the blocks in SynDEx and additionally, customized helper code is needed for designing period-true tasks (e.g. auxiliary code for blocking, timed tasks). This is the case for designing with Scicos/SynDEx and SynDEx only. In the case of hand-written code, no global variables might be necessary in the main module, which they are not in this comparison. In the main-loop of the application, the blocks are first called by their init functions (e.g. resource allocations), then their main functionality is called, followed by a finalization function call (e.g. free resources) - these steps are part of every situation, with/without Scicos/SynDEx or SynDEx (see figure 4.30).

Code complexity and cognitivity

Scicos and SynDEx do not add any extra complexity compared to hand-written code, but some complexity is caused by needed auxiliary functions. The cyclomatic complexity is the same for the Scicos/SynDEx and SynDEx situation but differs when compared to a hand-written solution by the extra auxiliary functions needed for SynDEx (cc_{AuxF}, see table 4.2). The Scicos PID example model remains human-readable, as well as the SynDEx model. A scheduling table provides a visual representation of all tasks, including their dependencies to each other (successors, predecessors) and this makes it possible to spot unused resources and to optimize the design. Problems occur, if real durations for the tasks are modeled - they are resulting in a scheduling table which is hard to perceive due to optical proportions.

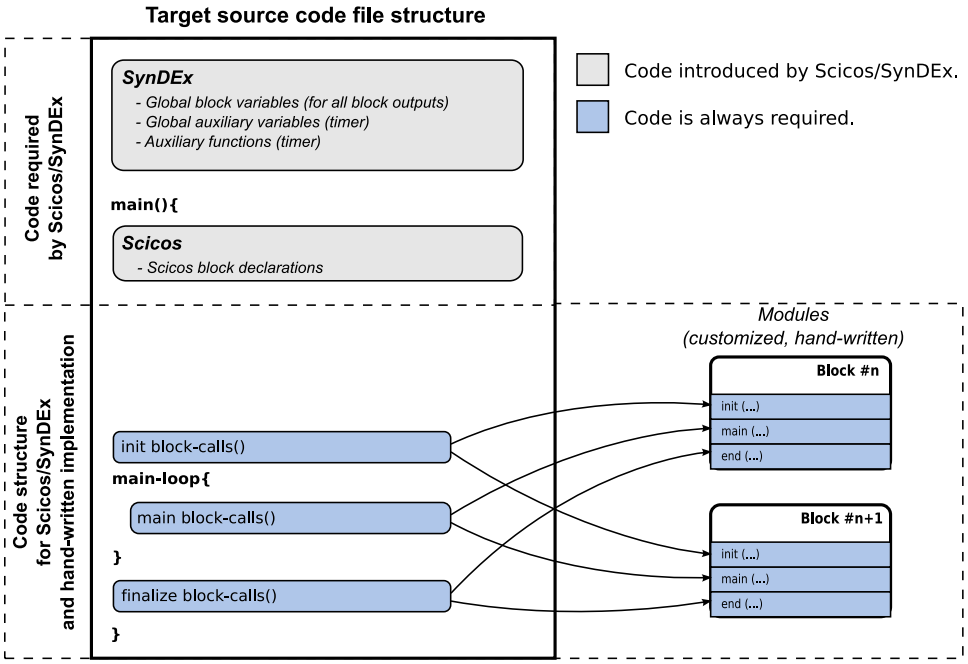


Figure 4.30: Scicos/SynDEx vs. hand-written code structure.

4.4.3 Effort

The effort for designing a PID algorithm with Scicos/SynDEx is broken down into several key actions (see table 4.3). The person hours (ph) listed are estimates made during development and are based on following presumptions: The developer is familiar with the working environment, and has expert knowledge about the problem domain. Additionally, the complete tool-chain is prepared and in working condition.

Before the Scicos/SynDEx framework can be used in combination with a customized hardware (ESE-Board), several scripts, programs and adaptions of the framework have to be done. The activities in the hereinafter given list require a great manifold of the time required implementing the examples in table 4.3. This list does not claim to be complete, several minor workarounds are not included and only issues encountered during the examples are taken into consideration. Note that stating the effort for these steps is difficult for several reasons: a comprehensive documentation for the framework was only partially found, new documentation was published during this work, the requirements for changing the structure of the macro code is dependent on the targets, and many more.

#	Development tasks	Model-driven	Hand-written	Effort (ph)
1	Design Scicos PID model and simulation	√	¬	3
2	Design SynDEx model and temporal design	√	¬	3
3	Implement driver modules (target language code)	√	√	30
4	Implement Scicos libraries (simulation blocks)	√	¬	5
5	Implement SynDEx libraries (code generation)	√	¬	10
6	Implement SynDEx libraries (algorithm & architecture)	√	¬	5
7	Implement PID algorithm by hand	¬	√	2

Table 4.3: Development effort with an already working tool-chain: model-driven vs. hand-written approach.

Clarifications:

1. Design and simulation of the hybrid system in Scicos until the results were satisfying.

2. Integration of the PID model in the SynDEx top-model, application of timer blocks for periodic tasks, determination of a proper time resolution, definition of constraints, obtaining and validating the scheduling.

3. These drivers are the same used for the hand-written and model-driven solution. The effort contains testing and modular design of timers, fan speed and sensor drivers.

4. Special blocks modeling real-world items, like the environment fan (envFan) and the environment fan sensor (evnFanSensor), have to be implemented.

5. Implementation of macro expansion files to generate C code (the target language definition): Timers, drivers, auxiliary blocks/functions, type mappings. In this case this is quite a fast to accomplish task since SynDEx is delivered with a C operator target language definition which was adapted for the ATmega128 microcontroller (the resulting target language was not optimized nor complete, and only implemented as needed for the examples).

6. Interface design (formal) for every additional SynDEx block which could not be automatically integrated from Scicos: Timers, type conversions, etc. Design the architecture model library.

7. A solution without Scicos/SynDEx, modular designed drivers are used, the algorithm is written by hand in a source file without intense testing.

1. **Installation of the framework.** Install the software framework on a development PC (this point is mentioned because the installation of the framework turned out to be sophisticated where critical information for a working framework was only given in a French description - the appendix of this work includes additional information about that).
2. **Become familiar with Scicos/SynDEx.** Understand how it works (from the GUI to the underlying scripts), what it can and can not do, search for documentation.
3. **Implement a m4x builder.** The macro code generated by SynDEx might be not directly useful for the given target hardware, e.g. threads for communication, semaphores and download labels. The m4x builder changes and cleans the structure of the macro code (depicted in figure 4.29).
4. **Implement a m4 builder.** Some strengths of a model-driven approach can only apply if components can be reused in several designs. Combining several Scicos-to-SynDEx transformed designs and integrating them into SynDEx is not possible. A m4 builder for combining several models is most probably be necessary (originally the PID example of this thesis consisted of several Scicos-to-SynDEx transformed components and a m4 builder was provisionally implemented).
5. **Adapt SynDEx templates.** Some of the generic templates of SynDEx might not be appropriate - therefore they need to be arranged even if it is not recommended by the SynDEx authors (e.g. the Makefile templates).

4.4.4 Model versus Reality

1. **Scicos PID model and real fan behavior.** The fan model was designed in Scicos with a PT1 behavior which turned out to be similar (according to the shape of the function) to the real fan behavior after a manual adaption of the PID parameters (compare figures 4.16 and 4.23). One reason contributing to this problem is for sure the large time base chosen (80 STU instead of 74 STU - even the 74 STU are already a rounded value). Creating a fitting model could turn out to be very sophisticated - extra effort and expert knowledge would be required. Another reason are the data types chosen - differences occur due to the quantification of the plant behavior (the model of the plant behavior is an estimated interpolation of the measured real plant behavior). The fan driver was implemented based on software pulse-width-modulation. A timer is regularly causing interrupts based on the set duty-cycle - no possibilities of modeling such a circumstance were found.

2. **Scheduling and time representation.** There were two choices for a representation of time in the models.

   First, a time base represented by operator cycles brings the advantage of a more precise model, but introduces following disadvantages: It requires more effort to model, the cognitivity of the scheduling table in SynDEx is reduced. Therefore, to scale the model in time, a virtual unit (*STU* - SynDEx Time Unit) representing n cycles had to be introduced - the price for this is the reservation of additional time-intervals which are not consumed, and the extra differences between model and real-world behavior. If $1\ STU = 1\ cycle$ then the modeled WCET times of blocks would only differ by the clock drift of the real operator.

   Second, a natural time base, e.g. microseconds, could be chosen as time base. In this case blocks with a WCET smaller than the time base will be modeled as being too time consuming, which leads to differences between model and reality. For both cases, a maximal block execution time can be modeled, but an operator might not be able to realize what was required by the application. For example, imagine a task should be executed every $4\,ms$, the time is translated to discrete SynDEx time units, and the algorithm is executed on the target platform. What was modeled might not be feasible on a discrete-time microcontroller target that calculates time based on a clock source and prescalers.

3. **Periodic tasks.** Durations and periods of tasks can be designed in SynDEx. Just entering the corresponding parameters in an algorithm block results in a task that will be called at least with this period - periods modeled are worst-case. However on the average case, due to the modeled WCET durations of prior called algorithms which terminate earlier than it can be seen in the scheduling, the period of the task will be smaller. This might be desired if a task has to be executed at a given period or better. If true-period tasks are desired, the proposed workaround for timed-execution of tasks is required (details are found in the monoprocessor example). Latter modeling method also introduces differences between the visual presentation of the model and the real-world execution where the timed task has blocking behavior to compensate for the early execution of prior tasks.

4.4.5 Scicos/SynDEx and a distributed embedded system with microcontrollers

During the design of the examples some key differences about what can be modeled with Scicos/SynDEx and what would be needed for designing distributed

embedded systems emerged. Of course, there are probably many more requirements - discoveries by the examples are mentioned.

Design requirement	Directly supported by Scicos/SynDEx	Proposed solution
Type mappings	model re-design required	add rules to target language definition (see PID example, 4.2).
Best effort period tasks	√	-
True-period tasks	¬	proposed algorithm design with blocking behavior (section 4.2).
Communication (macro code)	communication threads	transform via m4x-Builder (section 4.3).
Component reusability (SynDEx)	simple blocks only (no parameters)	hierarchical blocks have to be hard-coded. -
Component reusability (combine transformed Scicos models)	¬	combination by a m4-Builder.
Fault tolerance (redundancy)	samples by broadcast	com-medium modeled as broadcast.
Communication protocol	theoretically replaceable	blocking, synchronization and addressing (see section 4.3).

Table 4.4: Scicos/SynDEx on a microcontroller based distributed embedded system.

Some additional comments clarify the contents of table 4.4:

Type mappings. Blocks are designed with particular input/output types. If they had to be changed afterwards, e.g. include another math-library with different data types, the model would have to be redesigned.

Component reusability. In SynDEx, only simple blocks can be designed and reused in other models. Parameters, e.g. periods are not saved. The nesting of blocks is not supported directly, therefore hierarchical components can only be hard-coded and then be reused in other models/designs. Scicos-to-SynDEx transformed models need to be treated if two or more are required in a SynDEx model. An solution is a customized M4 builder which combines these files through previously inserted section tags.

Fault tolerance. A setup: three different temperature sensors take a synchronized snapshot by a broadcast communication medium. The data is then sent to a microcontroller containing a voter unit.

5 Conclusion

In this work, theoretical foundations for modeling embedded systems were presented and a modeling-framework (Scicos/SynDEx) was examined and evaluated. The fitness of this particular model-driven approach for a distributed embedded system with microcontroller targets depends on individual needs and requirements. Required design characteristics, priorities, tool support and many more aspects in systems development are depending very much on the particular problem domains. Thus, it is hardly possible to create a fitness reference covering all thinkable required aspects, however an attempt of creating some valuable statements about advantages and disadvantages of using a model-driven approach in this particular case follows.

The combined use of Scicos/SynDEx results in a high demand on dynamic memory introduced by the Scicos block structures. These results (table 4.1) cover only the dynamic memory consumption by the Scicos blocks. A comparison of the overall dynamic memory consumption between Scicos/SynDEx and hand-written code is hard to qualify for the general case, since hand-written solutions can be carried out in many variations. A suggestion for an approximate comparison is to assume that the hand-written solution uses the same libraries (drivers) like the model-driven solution - therefore this part of the used memory should be the same for both solutions. A malicious fact playing against a good comparison of the code size is the nature of compilers - the optimization abilities and differences of several compilers might diffuse results. Thus, the results can only be considered with caution. In the case of developing with Scicos/SynDEx, the static memory consumption of the model-driven approach is significantly higher than that of the hand-written solution, while in the case of using only SynDEx, the memory consumption can be considered as equal to the hand-written solution.

Next to the code size, the cognitivity of the designed models is of importance - the better the representation of the model the earlier aspects like bad designs or unused resources can be spotted. The code generated by Scicos/SynDEx is not downgrading the readability of the code structure in comparison to hand-written code. Additional structures are inserted by the framework, they are consuming memory, but they do not include obfuscating

characteristics, e.g. deep nestings of functions or confusing variable/function names.

The development effort of using the Scicos/SynDEx framework is signiffi-cantly higher than that one caused by a hand-written solution, but this might be only valid for the simple one-time applications (which are in fact not that simple, as seen in the examples) made in this thesis. The installation and adaption work necessary for a working, automated framework must not be underestimated - especially the additional scripts and tools for model transformations have to be carefully designed and in the case of designing a critical embedded system these tasks should only be accomplished by experts. Next to establishing a working framework, the design process with a ready-to-use framework requires much more time than the hand-written approach (table 4.3): $56 person\,hours$ versus $32 person\,hours$ (handwritten). Latter results are, of course, not quantitative statements, since these numbers represent a unique case and depend on the skill and preferences of the developer. Advantages of the model-driven approach might show up for designing complicated systems, product families, and when the price for setting up the framework could be neglected and components reused. The communication macro code structure generated by SynDEx in the case of a multiprocessor application opens an unanswered, fundamental question: Why are there communication threads created, if the characteristics of the designed models are atomic and the scheduling is non-preemptive? Threads imply the presence of a scheduler, context switches and additional resource consumption. Even a decoupling of the communication unit from the processor requires coordination resources. These aspects seem to be missing in the models.

Building models of real-world entities with a discrete machine causes always differences in the behavior, diminishing these differences results usually in more complex models. Creating an adequate model of the fan behavior did not turn out to be an easy task - the chosen design was not powerful enough to ensure an adequate execution of the simulated models on the target hardware - manual adaptions had to be made for a satisfying PID controller. Since the focus of this thesis was not about model construction, the fan and plant model were kept simple and were redesigned just one time. Expert knowledge and re-modeling of the fan is required to sharpen models. The design process would be improved if manufacturers/vendors included Scicos models within the delivery of the hardware - such models could be integrated in Scicos by the developer - instead of worrying about a customized design and its adequacy (this would also encourage the use of the free Scicos software). The differences between models and real applications seem to be mainly depending on: the

chosen representation of time (only a representation of $n \cdot cycles$ was possible, due to representation problems of the scheduling), the real execution time of tasks (they were modeled with WCET), and the introduction of true-period tasks. The modeling of true-period tasks is only possible with work-arounds - adding differences between model and real application.

A major advantage of the model-driven approach (and the idea of Scicos/SynDEx) is the safety of the design concerning deadlocks and resource allocations/optimizations. Such tasks can be error-prone and very complicated if done manually. Another benefit of the framework is that free resources can be easily spotted by the visual representation of the models - this could lead to several benefits as, for example, reduced hardware costs, space for additional features, or just reserved resources for an ongoing evolution of the software. The possibility of model simulation has shown to be of great value during the design of the first PID algorithm (velocity PID). The wrong design was detected early and a redesign started - in the case of a hand-written solution, the parameters of the wrong design could in some cases be refined until a solution for this single version of the application is doable - this could be disastrous if the software requires minor changes, for example if the fan was replaced by another model. This would be followed by confusion and uncertainty about the previously designed solution. Simulation capabilities narrow the path of possible project advancements towards expected solutions earlier than a traditional hand-written approach (figure 5.1). This might lead to reduced costs, development time, and risks.

Working with Scicos/SynDEx shows some characteristics of agile development. Models and simulation results are represented visually and system changes, e.g. system enhancements or product evolution, can often be realized faster than with a handwritten solution (which might require additional tests and design checks). Customer collaboration is supported by the models contributing to, or being part of, the specification (this is the idea, but an accurate model is not easy to accomplish, as the examples have shown) - this fact might reduce the need for exhaustive project documentation and reduced contract negotiations.

In which scenarios should Scicos/SynDEx be preferred to a hand-written solution? Based on the comparisons made (table 5.1) it should be valid to say that the Scicos/SynDEx framework can unfold its strengths in product families when a basic platform is established and the effort for modeling the components pays off (components are reused). SynDEx, as software, is supposedly meant for rapid prototyping. In the case of the combined use with

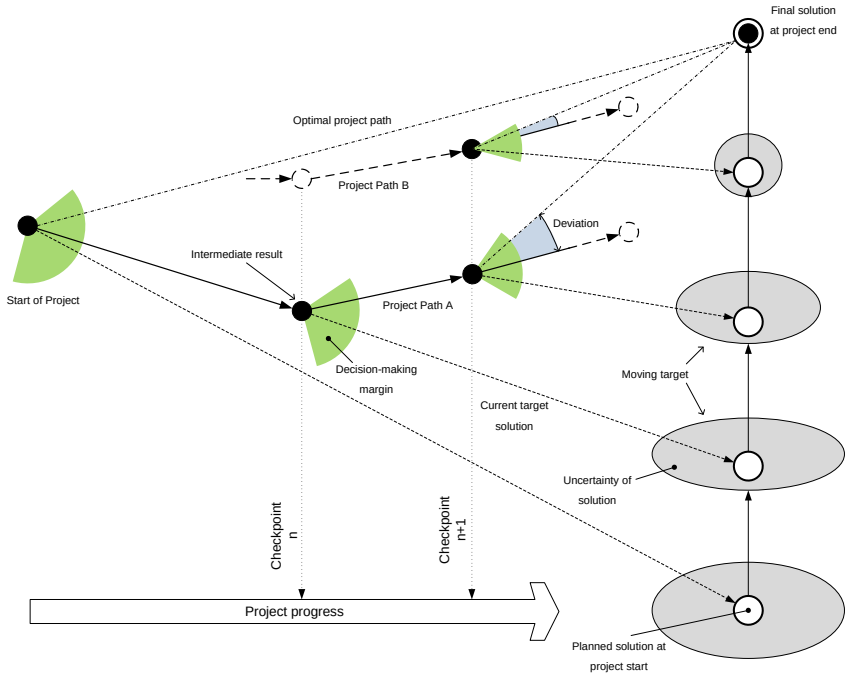


Figure 5.1: Scicos/SynDEx vs. hand-written solution. Simulation and modeling capabilities (Project Path B) decrease the size of the detour in comparison to hand-written projects (Project Path A). The uncertainty about the solution is reduced earlier, the deviation from the optimal project path diminished.

Comparison	Model-driven solution	Hand-written solution
Code size	34 796 *bytes*	4 922 *bytes*
Memory consumption		
(static)	976 *bytes*	226 *bytes*
(dynamic - Scicos)	$\approx \frac{100\ bytes}{1\ block}$	-
Code structure	extra variables/functions	-
Code cognitivity	human-readable	human-readable
Effort		
(PID, development)	56 *person hours*	32 *person hours*
(establish tools)	high effort	-
Simulation	early detection of mistakes	¬
Optimization	automatically, safe	by hand
Resources (display)	visual representation	source code only
Respond to changes	lower effort, quick	higher effort
Documentation	model as specification	formless

Table 5.1: Fitness of the model-driven approach compared to a hand-written solution.

Scicos on the ESE-Board, this can only be true, if the effort for setting up the tool and the underlying automated build process is neglected. Designing the libraries might still take more time for simple applications than the hand-written approach, but has the advantage of reuse and supports an easy redesign of the application. For instance, if the hardware fan, like the one in the PID example, should be replaced by another model it should be quite comfortable to build the model and then just compare these two models by the simulation results (with the assumption of accurate models). Additionally, models and simulation results might be directly discussed with the customer. Scicos/SynDEx should only be used if there is no shortage of memory capacity on the target platform. SynDEx itself does basically not introduce extra memory costs, however SynDEx does not support hybrid system simulation and modeling of hybrid systems.

Designing systems with the framework has shown that it is not only crucial to determine WCETs of tasks, of even the same importance are models which consider the minimal and possible durations of tasks. As seen in this thesis, an early termination of a task leads to gaps in the scheduling model, which might throw over the real behavior of the system. An approach to handle this would be to time-trigger all critical tasks, this would require either a lot of timers or a customized scheduler which can not be modeled with the framework. It might be, that this is not the idea of a synchronized distributed executive, it seems that best-effort applications are lying in the focus of the framework. Modeling time characteristics of a critical, distributed system might be better done with a time-triggered architecture where the problems of early task terminations are diminished (confer to [Pau04]).

The idea of a framework for model-driven development of distributed embedded systems with the support of hybrid system design and resource allocation/optimization sounds promising. Scicos/SynDEx introduces extra memory costs but can pay off in complex systems design. SynDEx itself handles the temporal design and algorithm allocations automatically for distributed systems while basically not requiring extra memory space nor increasing the complexity of the model. The framework is not of a commercial nature, which might be the reason for its immaturity concerning the usability (see the appendix for hints) - the implementation focus lies clearly on applying the underlying concepts of the AAA-Methodology. Implementing a complete model-driven development environment from scratch is an interdisciplinary challenge. Expert knowledge in several fields like model building, formal methods, scheduling, code generation, compiler design, DSL design, dependable system design, communication protocol design, hardware knowledge, software architectures in general, and many more are required. The effort for creating a model-driven development environ-

ment can therefore be labeled very demanding. Adapting the Scicos/SynDEx framework requires also a relatively high effort. Thus, it might be reasonable to consider the use of commercial software with already provided hardware/software libraries. The support for the Scicos/SynDEx framework seems to have stopped since no updates for the Scicos-to-SynDEx-Gateway working with the latest Scicos distributions could be found. SynDEx is still under development and with its evolution, based on the fundamental AAA approach for embedded system design, it might turn out to be a very valuable asset in the future.

5.1 Outlook

In this thesis we have seen that the real behavior of systems deviates in different shapes from the modeled systems. First, time and data were abstracted which lead to deviations. Second, tasks were modeled with WCET, but in reality the real task durations can not be predicted. It might be possible to classify the errors caused by discretization and use this information for a better predictability in systems modeling, that means drawing a connection from hybrid systems timings to distributed systems scheduling. Furthermore it would be of interest if there are advantages if several task duration parameters, such as worst-case execution time and best-case execution time, are considered in a scheduling model. In that way the running-away of tasks, as it is the case in single-processor applications in SynDEx, might be prevented.

Building a seamless modeling tool-chain with the Scicos/SynDEx framework and the implementation of some examples have shown some potential improvements for the concepts realized in the SynDEx software:

- The scheduling for communication calls could be integrated into the operator scheduling and thus serving the generation of code for microcontrollers (only one sequencer for the operator scheduling).
- Support the realization of fixed-period tasks rather than only best-effort tasks.
- Include code-size indications and constraints which allow a prediction of the code-size for a given hardware target.
- Support of automatic type-mappings within the graphical tool.
- Automatic implementation of a small static-scheduler that provides a a way of combining long and short period tasks effectively.

Regarding component-based development the effort for design could be reduced by a great deal if manufacturers would supply standardized models for simulation and implementation for their products. This might already be true

for some suppliers in combination with certain tool-chains, but still, a commonly accepted format for models is required.

Bibliography

[Atm06] Atmel Corporation. *ATmega128*, 2006. Rev. 2467N–AVR–03/06. Available online at http://www.atmel.com/products/avr/.

[Boe88] Barry Boehm. A Spiral Model of Software Development and Enhancement. *Computer*, 21(5):61–72, May 1988.

[Bro04] Alan W. Brown. Model Driven Architecture: Principles and Practice. *Software and Systems Modeling*, 3:314–327, 2004. 10.1007/s10270-004-0061-2.

[CE00] Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000.

[CNC06] Stephen Campbell, Ramine Nikoukhah, and Jean-Philippe Chancelier. *Modeling and Simulation in Scilab/Scicos*. Springer Science+Business Media, Inc., 2006.

[Col01] Defense Systems Management College. *Systems Engineering Fundamentals : Supplementary Text / Prepared by the Defense Acquisition University Press* . The Press, Fort Belvoir, Va. :, 2001.

[CPPSV06] Luca Carloni, Roberto Passerone, Alessandro Pinto, and Alberto Sangiovanni-Vincentelli. Languages and Tools for Hybrid Systems Design. *Foundations and Trends in Design Automation*, 1(1):1–204, 2006.

[EBK03] W. Elmenreich, G. Bauer, and H. Kopetz. The Time-Triggered Paradigm. In *Proccedings of the Workshop on Time-Triggered and Real-Time Communication Systems*, 2003.

[EK08] Khaled El Emam and Akif Günes Koru. A Replicated Survey of IT Software Project Failures. *IEEE Software*, 25:84–90, 2008.

[Elm09] W. Elmenreich, editor. *Embedded Systems Engineering*. Vienna University of Technology, Austria, Vienna, Austria, 2009. ISBN 978-3-902463-08-1.

[Elm10] W. Elmenreich. Evaluating the static and dynamic memory consumption for AVR microcontroller programs. Networking Embedded Systems Blog, October 2010. Available at http://netwerkt.wordpress.com/2010/10/06/memeval.

[Fau06] Cyril Faure. Traducteur Scicos/SynDEx - Installation et Utilisation, v2.2.2, April 2006. INRIA, Project Eclipse.

[fRiCSC03] INRIA The French National Institute for Research in Computer Science and Control. Launch of the Scilab Consortium Dedicated to Scientific Computing, May 2003. Available online at http://www.scilab.org/aboutus/pressroom/press_release/pr_20030522.

[GLS99] Thierry Grandpierre, Christophe Lavarenne, and Yves Sorel. Optimized Rapid Prototyping For Real-Time Embedded Heterogeneous Multiprocessors. In *Proceedings of 7th International Workshop on Hardware/Software Co-Design, CODES'99*, pages 74–78, Rome, Italy, May 1999.

[Gro01] Object Management Group. *Model Driven Architecture (MDA)*, July 2001. Rev. ormsc/2001-07-01. Available online at http://www.omg.org/cgi-bin/doc?ormsc/2001-07-01.

[GS03] Jack Greenfield and Keith Short. Software Factories: Assembling Applications with Patterns, Models, Frameworks and Tools. In *OOPSLA '03: Companion of the 18th annual ACM SIGPLAN Conference on Object-oriented programming, systems, languages, and applications*, pages 16–27, New York, NY, USA, October 2003. ACM.

[IEE90] IEEE. Standard Glossary of Software Engineering Terminology. IEEE Std 610.12-1990. Technical report, IEEE Computer Society Press, 1990.

[IEE00a] IEEE. IEEE 100 The Authoritative Dictionary of IEEE Standards Terms Seventh Edition. *IEEE Std 100-2000*, 2000.

[IEE00b] IEEE. IEEE 1471-2000 Recommended Practice for Architectural Description for Software-Intensive Systems. Technical report, IEEE Computer Society, 2000. Available online at http://ieeexplore.ieee.org/servlet/opac?punumber=7040.

[iGfiSm03] iteratec Gesellschaft für iterative Softwaretechnologien mbH. Kurzbeschreibung iteratec Vorgehensmodell, March 2003. Available online at www.iteratec.de/download/iteratec_Vorgehensmodell.pdf.

[Koe09] Alexander Koessler. A Platform for Teaching and Research on Distributed Real-Time Systems. Master's thesis, Technical University of Vienna, Institute for Computer Engineering, Treitlstr. 3/2/182-2, 1040 Vienna, Austria, March 2009.

[Kop97] Hermann Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Kluwer Academic Publishers, Norwell, MA, USA, 1997.

[KS04] Rémy Kocik and Yves Sorel. A Methodology to Reduce the Design Lifecycle of Real-Time Embedded Control Systems. In *Proceedings of European Simulation and Modelling Conference, ESM'04*, Paris, France, October 2004.

[NAN03] Masoud Najafi, Azzedine Azil, and Ramine Nikoukhah. Implementation of Continuous-Time Dynamics in Scicos. In *15TH European Simulation Symposium and Exhibition*, Delft, The Netherlands, October 2003.

[NR68] Peter Naur and Brian Randell. Software Engineering: Report of a conference sponsored by the NATO Science Committee, Garmisch, Germany, Brussels, Scientific Affairs Division, NATO, October 1968. Available online at http://homepages.cs.ncl.ac.uk/brian.randell/NATO/.

[OMG03] OMG. MDA Guide Version 1.0.1. Technical Report omg/2003-06-1, OMG, June 2003. Available online at http://www.omg.org/cgi-bin/doc?omg/03-06-01.

[Pau04] Christian Paukovits. Modellierung und Scheduling von flexiblen, zeitgesteuerten Kommunikationsprotokollen. Bachelor's thesis, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 3/3/182-1, 1040 Vienna, Austria, 2004.

[PCM01] Arjun Panday, Damien Couderc, and Simon Marichalar. AIL: Description of a Global Electronic Architecture at the Vehicle Scale. In *DATE '01: Proceedings of the Conference on Design, Automation and Test in Europe*, page 112, Piscataway, NJ, USA, 2001. IEEE Press.

[Roy70] Winston W. Royce. Managing the Development of Large Software Systems: Concepts and Techniques. In *Technical Papers of Western Electronic Show and Convention (WesCon)*, 1970.

[Sci10] *Scilab Manual*, April 2010. Rev. 5.2.2. Available online at http://www.scilab.org/product/man/index.html.

[Sel03] Bran Selic. The pragmatics of model-driven development. *IEEE Software*, 20:19–25, 2003.

[SK98] Gerhard-Helge Schildt and Wolfgang Kastner. *Prozeßautomatisierung*. Springer, Wien, 1998. ISBN 3-211-82999-7.

[Som04] Ian Sommerville. *Software Engineering*. Addison-Wesley, Harlow, England, 7 edition, May 2004. ISBN 0-321-21026-3.

[Sor94] Yves Sorel. Massively Parallel Systems with Real Time Constraints, the Algorithm Architecture Adequation Methodology. In *Proceedings of Conference on Massively Parallel Computing Systems, MPCS'94*, Ischia, Italy, May 1994.

[SPF07] Paul F. Smith, Sameer M. Prabhu, and Jonathon Friedman. Best Practices for Establishing a Model-Based Design Culture. Technical report, The Mathworks, 2007.

[SVEH07] Thomas Stahl, Markus Völter, Sven Efftinge, and Arno Haase. *Modellgetriebene Softwareentwicklung*. dpunkt.verlag, May 2007.

Acronyms

ADC Analog Digital Converter. An ADC is an IC that quantifies analog to digital values.

AIL Architecture Implementation Language. The AIL is a description language that allows for an internal representation of the architecture and acts as a connection with tools to simplify the construction, planning, verification, capitalization, and documentation of an architecture.

AOSTE is one of INRIA's research teams focusing on high-level modeling, transformation and analysis and implementation onto embedded platforms

CAD Computer Aided Design

DC Direct Current

DSL Domain-Specific Language. A DSL is a language well suited to describe a specific domain.

IC Integrated Circuit

IEEE Institute of Electrical and Electronics Engineers, Inc. IEEE is a non-profit organization and the world's leading professional association for the advancement of technology.

INRIA Institut National De Recherche En Informatique Et En Automatique. INRIA stands for the French National Institute for Research in Computer Science and Control.

LCD Liquid Crystal Display. A LCD is a display based on liquid crystal technology.

LOC Lines of Code

MARTE UML Profile for Modeling and Analysis of Real-time and Embedded Systems. MARTE is the specification of an UML profile adding model-driven development capabilities for real-time and embedded systems to UML.

MDA Model Driven Architecture

MEM Micro-Electro-Mechanical System

MOF Meta Object Facility. The MOF is the meta-meta-model and core of the MDA defined by the OMG.

NATO North Atlantic Treaty Organization. The NATO is an alliance of 26 countries from North America and Europe fulfilling the goals of the North Atlantic Treaty signed on 4 April 1949.

OMG Object Management Group. The OMG is an open consortium for improving interoperability and portability of software systems by defining manufacturer - neutral standards.

PWM Pulse Width Modulation. PWM is a signal modification technique based on the adjustment of ratio between high-/low- time of the signal line.

RPM Rounds per Minute. RPM stands for the cycles per minute of a rotating device.

SPI Serial Peripheral Interface. The SPI bus is a full duplex, synchronous serial data link standard.

UML Unified Modeling Language. The UML is a modeling language based on the OMG's MOF.

UNIX Unix is a computer operating system originally developed in 1969 by a group of AT&T employees at Bell Labs.

WCET Worst-Case Execution Time

A Notes

A.1 Scicos/SynDEx - Configuration

The setup for exercising the examples as well as important Scicos/SynDEx installation notes are discussed as follows in this section.

Scicos is a toolbox of the Scilab package and its development lies in the responsibility of the "Scilab Consortium". Initially it was developed and maintained by INRIA [1]. It's distributed freely, including the source code.
Scilab - Version: 4.1.1

SynDEx has been developed by INRIA in the Rocquencourt Research Unit France by the AOSTE team. It's free for non-commercial use.
SynDEx - Version: 7.0.0

The avr-gcc compiler packet and tools.
avr-gcc - Version: 4.2.2-1
binutils - Version: 2.17
avr-libc - Version: 1:1.4.7-1

A debugging/programming tool interfacing the GDB and JTAG.
avarice - Version: 2.6

Experiment (ESE-Board) Monitor.
JTagZeusProg - Version: 0.2

Scilab and Scicos-SynDEx Installation Notes

First of all, it is important to only use the specified versions of the software. Up to the date this thesis was written, the only setup of software versions listed above seemed to allow Scicos working together with SynDEx via the Scicos-SynDEx interface.

Scicos

Necessary files:
scilab-4.1.1-src.tar.gz - Only the source version of the scilab-4.1.1 package provides the necessary resources for the Scicos-SynDEx gateway.

[1] INRIA - The French National Institute for Research in Computer Science and Control

Compilation notes:
For the compilation there is, amongst others, a Fortran compiler necessary - here, the "GNU Fortran - 4.2.4" compiler was used. Additionally some "-dev" packages might also be required: libXext-dev, libxmu-dev, libxmuu-dev, linux-libc-dev, xutils-dev.

SynDEx

Necessary files:
syndex-7.0.1-linux-i586.tar.gz

Extract the contents of "syndex-6.8.5-linux.tgz" and start. "GNU m4" has to be installed to execute the GNUmakefile generated by SynDEx.

Scicos-SynDEx Gateway

Necessary files:
ScicosToSynDEx.tar
s2s_scicosFiles.tar

The integration of the Scicos-SynDEx Gateway requires some script file editing and files moved into the right directories:

- Create a folder named "SCICOS_SYNDEX" in the Scilab-install directory.
- Extract "ScicosToSyndex.tar" into the "<Scilab-install-dir>/SCICOS_SYNDEX".
- Edit the file "scilab.star" in the <Scilab-install-dir>: Attach the line "exec SCI/SCICOS_SYNDEX/dot.scilab" at the end of the file. Scilab will now prompt about the loaded SynDEx module (see figure A.1). With the installation of the Scicos-Syndex Gateway an additional entry inside the graphical interface of Scicos is added: Object→To SynDEx (see figure A.2).
- Extract "s2s_scicosFiles.tar" into the "<SynDEx-install-dir>".

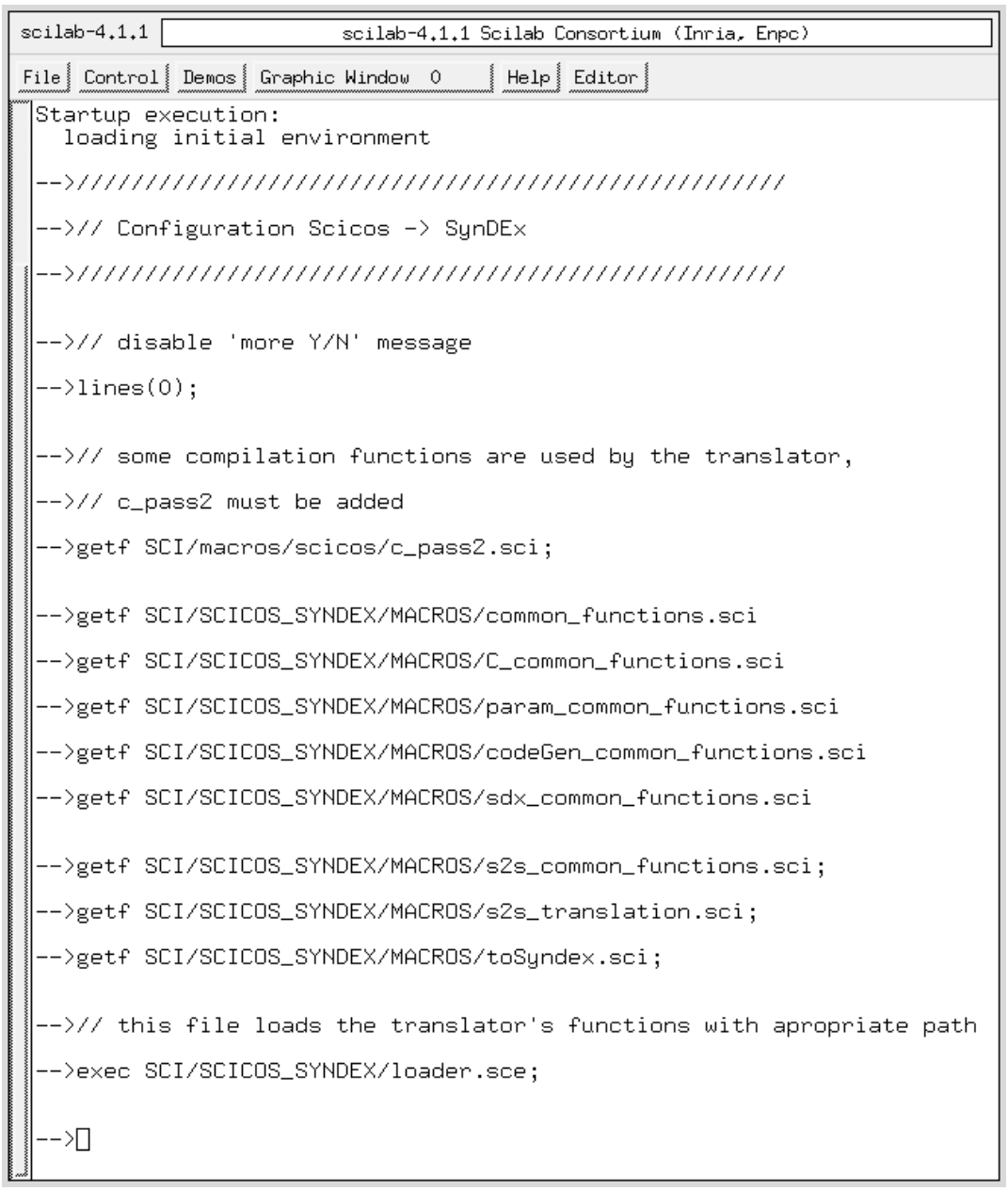


Figure A.1: Screenshot of Scilab with the installed SynDEx gateway module.

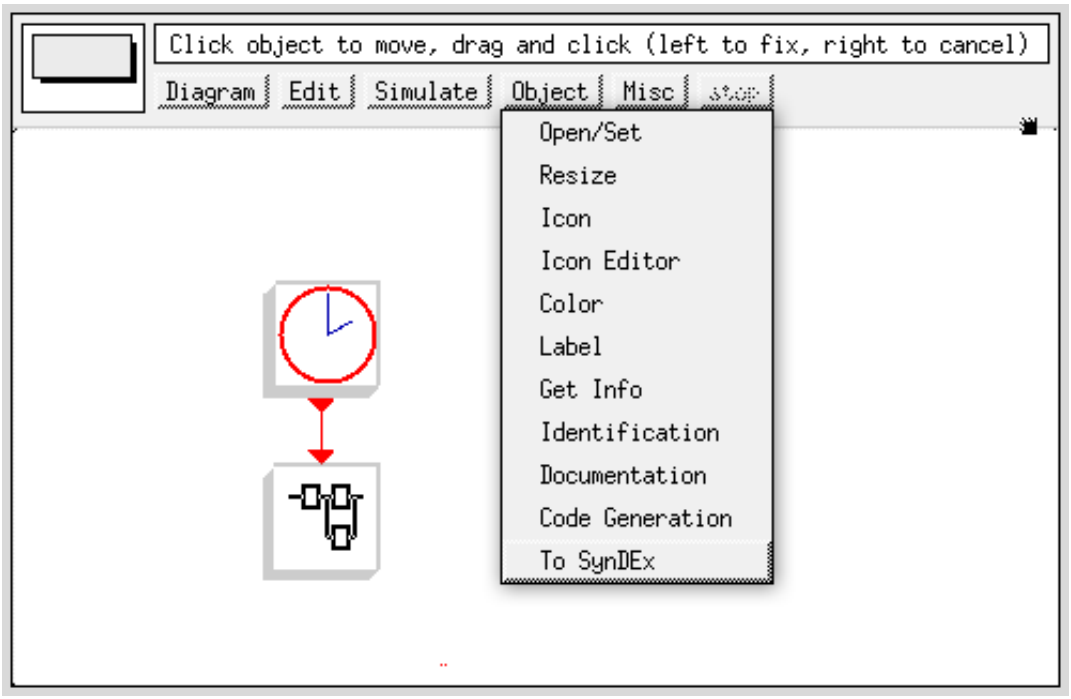


Figure A.2: Screenshot of Scicos with the installed SynDEx gateway module.

A.2 Scicos/SynDEx - Documentation

Scicos

Creating new blocks in Scicos

The basic steps of creating blocks for Scicos are documented in this section (see also [CNC06]). Each Scicos-Block consists of two parts: an "interfacing function" and a "computational function". The interfacing function handles the interaction with the graphical editor. The computational function defines the behavior of the block during the simulation. Both have to be designed, compiled and "linked" to the Scicos software. Following procedure is suggested:

1. Implement the computational function, e.g. IO_LEDBar_8bit.c. In general, the structure of a block's computation function coded in C looks like the following:

   ```
   #include "scicos\_block.h"
   #include <math.h>
   void my_block(scicos_block *block, int flag){
   ...
   }
   ```

   Examples for computational functions can be obtained in the <Scilab-installation-directory>/routines/scicos (e.g. summation.c).
2. Implement the interfacing function, e.g. IO_LEDBar_8bit.sci. Examples for interfacing functions can be obtained in the <Scilab-installation-directory>/macros/scicos_blocks (e.g. SUMMATION.sci).
3. For the compilation of the files, several Scilab commands have to be executed. The man pages of Scilab should also be checked out [Sci10]. To make the job easier, generate a builder file named "builder.sce" with following (example of building the library "ESE" with a 8 bit LED bargraph) content:

   ```
     comp_fun_lst=['IO_LEDBar_8bit']; // ['foo1','foo2',...]
     c_prog_lst=listfiles('*.c'); // c files in directory
     prog_lst=strsubst(c_prog_lst,'.c','.o');
   // generate loader and compile
     ilib_for_link(comp_fun_lst,prog_lst,[],'c','Makelib','loader.sce','ESE');
     genlib('lib_ESE',pwd()); // compile macros and generate lib
   ```

   Execute this script by starting Scilab, change to the directory where the files are stored (use the command "cd") and execute the script

("exec builder.sce"). Several files are generated: <some library files>, loader.sce, Makelib (a makefile), lib, and some more.

4. (Optional Step) Create a local scilab script file in the directory of the library files: <myScilabScript>.scilab. It can be useful for customizing commands for the treatment of the libs and shall at least contain the call of the loader.sce file, e.g.:

```
// this file loads the library functions with appropriate path
  exec /home/exa/DA/code/scicos/eBoardEnv/eBoardEnv.scilab;
```

5. Load the library into Scicos. To make Scicos load the new blocks at start-up the line "exec <PATH>/<MYSCRIPT>.scilab" (e.g. "/home/exa/DA/code/scicos/eBoardEnv/eBoardEnv.scilab") shall be appended to the file <Scilab-Installation-Directory>/scilab.star. If no <myScilabScript>.scilab is used, just call the loader.sce directly.

6. The new block is now ready to be used. Start Scicos, in the menu click "edit"→"Add new block" and enter the name of the block in the popped up dialog field. Now the block can be placed into the Scicos diagram.

Creating new palettes in Scicos

Creating a new palette makes the editing process of diagrams more comfortable. To do this, use Scicos to create a new palette.

1. Create the palette, e.g:

```
create_palette('/home/exa/DA/code/scicos/eBoardEnv')
```

   This will create the palette eBoardEnv.cosf in the chosen directory (which must contain all the necessary interfacing functions).

2. Add the new palette to Scicos, e.g:

```
load('<lib file absolute>');
scicos_pal($+1,1)='<Name of the Palette>';
scicos_pal($+1,2)=strcat(<Path to the library files>,'<the palette>.cosf');
```

   For the sake of some working comfort, these lines should be attached to a Scicos script, e.g. to eBoardEnv.scilab:

```
eBoardEnvPath='/home/exa/DA/code/scicos/eBoardEnv';
eBoardEnvLoader=eBoardEnvPath+'/'+'loader.sce';
eBoardEnvLib=eBoardEnvPath+'/'+'lib';
eBoardEnvPal=eBoardEnvPath+'/'+'eBoardEnv.cosf';

// this file loads the library functions with appropriate path
  exec(eBoardEnvLoader);
```

```
// load the palette and add it to the Scicos' palette menu
  load(eBoardEnvLib);
  scicos_pal($+1,1)='eBoardEnv';
  scicos_pal($+1,2)=eBoardEnvPal;
```

Scicos-SynDEx Gateway

The full documentation, publications and tutorials can be found on the SynDEx homepage [2]. For starters, with Scicos and SynDEx, the gateway documentation [Fau06] is recommended.

After the design of an synchronous diagram in Scicos, this diagram can be translated into a SynDEx conform diagram (Note: An asynchronous scicos diagram can't be translated to SynDEx - trying that causes an "error 10000" (something like a clkRoot error)). This is done by making a superblock of the Scicos diagram and then translate it to SynDEx by choosing following entries in the menu: Object→To SynDEx. This will start the translation process with the pop-up of a parameter window (see figure A.3).

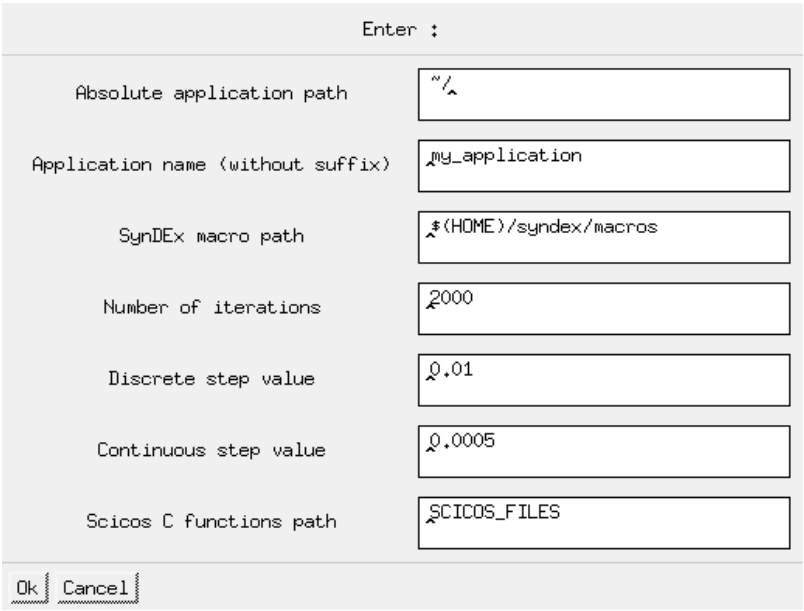


Figure A.3: SynDEx parameter-window screenshot of the Scicos-to-SynDEx gateway.

[2] http://www.syndex.org/scicosSyndexGateway/index.htm

The parameters are explained as described on the SynDEx homepage:

- Absolute application path: the path where the SynDEx files will be generated.
- Application name (without suffix): the application name.
- SynDEx macro path: the path where the SynDEx macros are located (usually the path to SynDEx + "/macros").
- Number of iterations: the number of steps the final application has to process.
- Discrete step: the discrete step value (useful when continuous blocks are present).
- Continuous step: the integration step value (useful when continuous blocs are present).
- Scicos C functions path: the path where the Scicos C functions are located.

If the Scicos-SynDEx gateway has successfully translated the diagram, the generated files were stored in the <Absolute Application path>. Following files were created using a simple test-diagram (A.4, Scicos Superblock: A.5):

- demo.sdx - Contains the transformed graph which is readable by SynDEx.
- demo.m4x - Structures and functions for the mechanism of the Scicos-SynDEx macros.
- demo.m4 - Contains architecture properties, for example rootOperator and processor-architecture.
- demo.m4m - Contains the definition of the "rootOperator_hostname" property.
- RootOperator.m4x - Defines the root operator (for example the microcontroller-application).In a single processor application, like in this demonstration application, the demo.m4m is included inside this file.
- GNUMakefile - ""

The relationship of the tools and generated artifacts is given in figure A.6.

SynDEx

Notes for the first tries with SynDEx are found in the Scicos-SynDEx gateway documentation [Fau06]. This paragraph describes how to use SynDEx to generate code for a mono-processor setup which was already described in

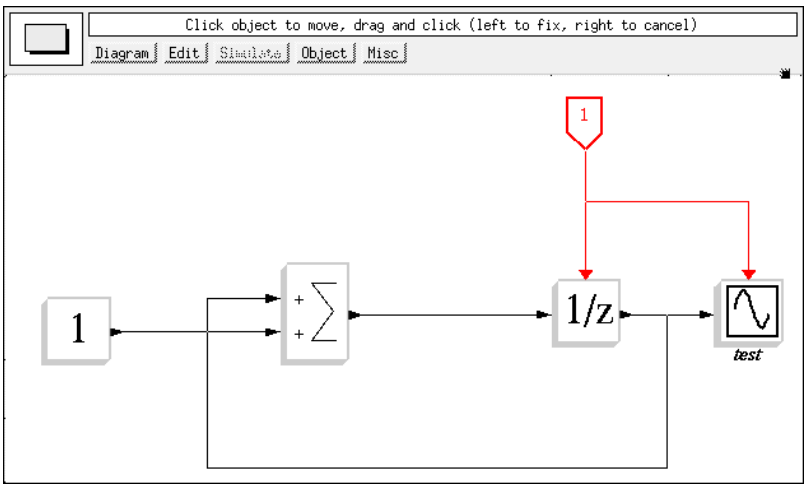


Figure A.4: Demo - diagram used for the demonstration of the Scicos2SynDExGateway.

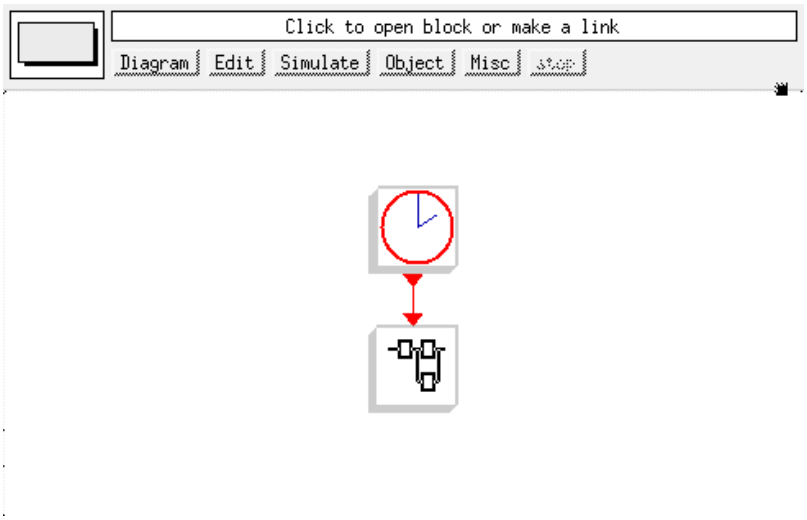


Figure A.5: Demo - diagram transformed to a Scicos Superblock.

french [Fau06, section 1.4.1], however some additional notes are required for a successful usage.

Section 1.4.4, Pts. 1-3

When SynDEx is started via command-line, it is mandatory to pass the -libs parameter, otherwise SynDEx will not find even the libraries inside its own distribution: "syndex -libs <SynDEx-install-dir>/libs". Now it should be

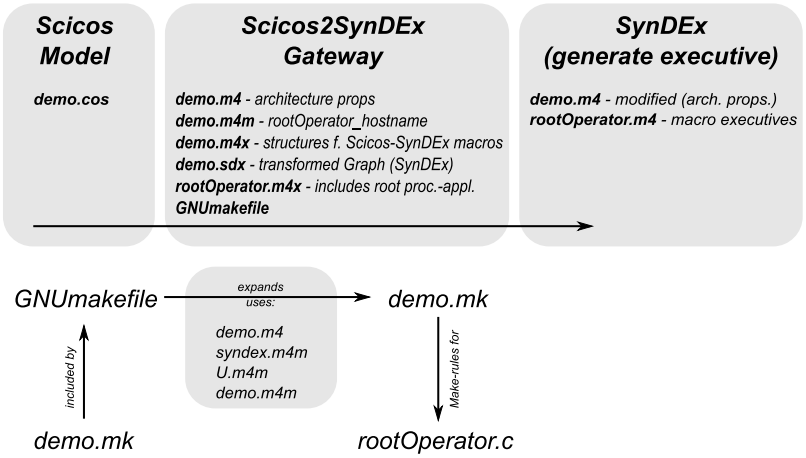


Figure A.6: Scicos2SynDEx - Generated artifacts.

possible to open a *.sdx file within SynDEx. For a first start of SynDEx, a test diagram translated by the Scicos-SynDEx Gateway was used (see figure A.4). After starting SynDEx the demonstration file "demo.sdx" was opened using the context menu (see figure A.7).

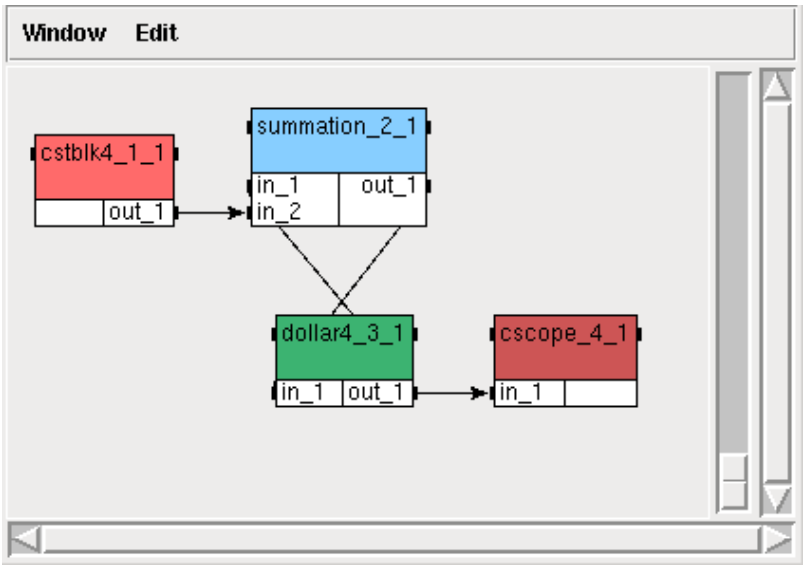


Figure A.7: SynDEx - Screenshot, the test diagram.

Generating code requires, first of all, the target-architecture to be chosen. For test purposes the "U" processor type (within the libs of the distribution) was chosen: "Architecture"⇒"Edit Operator Definition", see figure A.8. The automated code generation of SynDEx should generate code for the init and

loop phase of the algorithm. The cscope block is not defined in SynDEx, for a successful code generation libraries for this block have to be implemented.

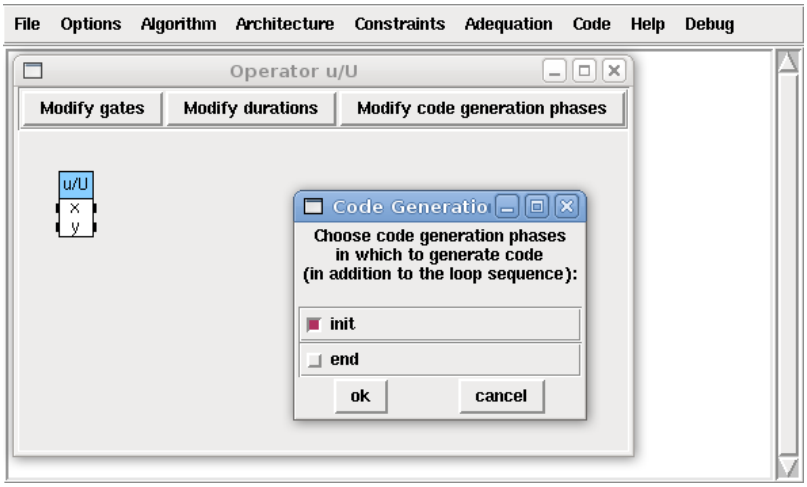


Figure A.8: SynDEx - Screenshot, choosing the target architecture.

Section 1.4.4, Pt. 4

Start the Adequation, for example: "Adequation"→"No Flatten".
Start the code generation by: "Code"→"Generate Executive". Entering this command will produce an intermediate artifact, not the finished target-architecture code. The file "rootOperator.m4" is created and the file "demo.mk" is changed (in this case a line with "dnl" after the "include..." is inserted, as well as an "endarchitecture_" at the end of the file; additionally the SynDEx version used for the code generation is inserted).
Another note: If the diagram is saved in SynDEx, a demo.sdc file is created - it contains just some miscellaneous version information.

Section 1.4.4, Pts. 6-7

Changing the name of the target machine in demo.m4m is necessary for the "rsh" connecting to the localhost. The "GNUmakefile" might need some editing, for example to set the right m4 macroprocessor installed on the develop-pc, or to set some path variables. Executing the GNUmakefile will create the demo.mk file, which is included at the end of the GNUmakefile itself. The demo.mk file in this example is used for two things. First, to create (macro-expand) the rootOperator.c file. Second, to compile, link and generate the executive. Note: Anything related to the "rsh" command inside the makefiles can be discarded, since their only purpose is to automate the compilation of the sources on a different target machine.

Files created by the GNUmakefile (+ demo.mk) are demo.mk, rootOperator.c, rootOperator.rootOperator.o, rootOperator.

Scicos-SynDEx Gateway developer notes on the PID Controller example

Consider the PID superblock from the mono-processor example (figure 4.14 in section 4.2). This block is now converted using the Scicos-to-SynDEx gateway resulting in the following files: FSC.m4, FSC.m4m, FSC.m4x, FSC.sdx, GNUmakefile, rootOperator.m4x. The FSC.sdx file can now be opened in SynDEx (see figure A.9).

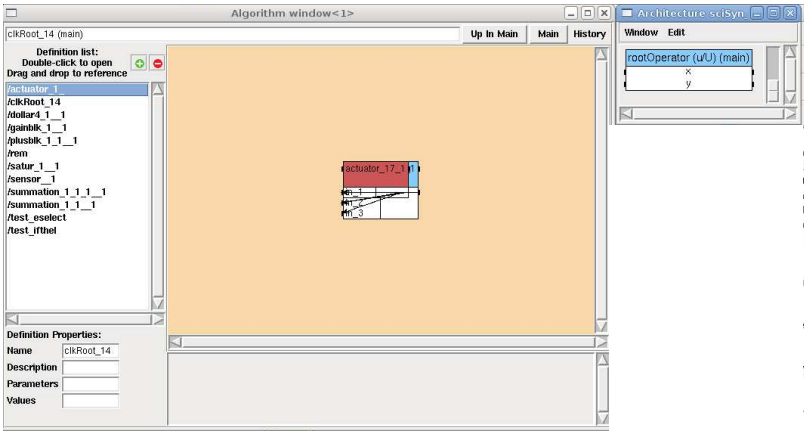


Figure A.9: SynDEx - Opening of an example application.

After rearranging the blocks in a human-readable way, following steps will make the algorithm ready to be used in another SynDEx diagram. Only the FSC.sdx file is needed by SynDEx, and the macro-expansion file FSC.m4x is needed for future steps. The FSC.sdx is edited by a text editor (that's more convenient than fighting with the SynDEx bugs in the graphical editor) and all the architecture dependencies and durations removed or commented with "#". After that the real durations on the target architecture may be inserted (don't forget to include the necessary files if needed). The PID should be able to take two inputs of the type "double": Setpoint (w_i), Actual Input(y_i). Additionally the PID will need one "double" output: Actual Output(u_i). To do this, following lines need to be inserted (the next line right after "def algorithm pidFanController :"):

```
? double[1] w_i 1;\\
```

```
? double[1] y_i 2;\\
! double[1] u_i 1;\\
```

Now the file is renamed to "pidFan.sdx" since this algorithm will be a PID block suitable for the fan (RPM - scaling is included in the block, this is necessary for a more comfortable automated edit- and build-process). Additionally, the algorithm is baptized by replacing the "def algorithm clk_root..." tag by a name of choice. Now either copy the new algorithm in the "<SYNDEXHOME>/libs" or make a symbolic link to it in that directory. The PID controller block is now ready to be used in SynDEx.
The next step proposed is to create a new project folder. If there are some home-made development scripts, then copy these files in the project folder (that could be m4x-builder scripts, a customized GNUMakefile or something else).
Start a new SynDEx project. The new library is found in the SynDEx menu: "File"→"Included Libraries"→"pidFan". Start the design process with the use of the new PID algorithm. In the PID example all the blocks have to be called with equal periods. To do this, click on EVERY single block and enter the period. Note: If at least one is forgotten and the algorithm adequation called, a "division by zero" error message occurs: In this case close all files and SynDEx, reopen it (it is good NOT to save after entering periods, because the file will be malformed, not readable, and a start from scratch is required) and do everything again from the last saved point (hopefully before any periods were entered); all blocks previously shifted from one hierarchy level into another must be re-edited as well as all the lost block parameters... Double/Tripple/Multi-check all periods before adequation (don't forget to handle all switch-blocks branches)!
After the design of the algorithm, chose the target architecture (e.g. the home-made "eBoard"→"monoProcNode3") in the "Architecture" menu. Now it should be possible to successfully call the algorithm adequation, obtain the scheduling and generate macro-executive code. Note: Even though it seems that everything was done in the right way, a new message might occur when starting the adequation: "Uncaught exception: File "algorithm/transform.ml", line 287, characters 7-13: Assertion failed" - Why, where, what is wrong? SynDEx won't tell. After some tests it was figured out that a connection the new I/O ports within the block (they were hidden behind a block) had been forgotten - well, let's start from the last saved point again and enter all periods, move hierarchical blocks and re-enter parameters in the previously inter-hierarchy-level moved blocks). The adequation might require some extra durations for some blocks to be entered and don't bother(!?) if the adequation seems to stop half way, or seems not to start or finish. With a diminished trust in adequation results, keep trying some times and it might start / finish

or ask for further block durations.
If the scheduling is satisfying, macro code can be generated: "Code"→"Generate Executive(s)". This will generate the files FSCMonoProcessorStage03.m4 and node3.m4, both containing macro code.
The next step is the macro code into C code expansion phase. Therefore the GNUmakefile has to be adjusted properly to your needs. Note: Don't forget to touch at least an empty <ApplicationName>.m4m file - SynDEx does not provide conclusive error messages!
In order to successfully expand to C code, self-made m4 definitions files are needed (for this example macro expansion files for the ESE-board, task-timings and the ATmega128 were written). To automatically generate customized Makefiles a customized syndex.m4m is needed. With the customized GNUmakefile in a Unix shell type: "make expand". That will generate the customized Makefile for node3 (FSCMonoProcessor.mk). "make node3.c" will finally generate C code, but things have to be considered. The macro-expansion file (.m4x) generated by the Scicos-to-SynDEx gateway is needed since it holds macro-expansion rules for most of the needed blocks (remember to remove the default sensor, actuator and maybe other example .c includes in this file - otherwise they will be included in the generated C code file). Replace the m4-include calls with C includes, e.g.: Replace...

```
include(/home/exa/DA/syndex-6.8.5/SCICOS_FILES/s2s_common.c)\\
include(/home/exa/DA/syndex-6.8.5/SCICOS_FILES/gainblk.c)\\
include(/home/exa/DA/syndex-6.8.5/SCICOS_FILES/plusblk.c)\\
include(/home/exa/DA/syndex-6.8.5/SCICOS_FILES/dollar4.c)\\
include(/home/exa/DA/syndex-6.8.5/SCICOS_FILES/satur.c)\\
include(/home/exa/DA/syndex-6.8.5/SCICOS_FILES/summation.c)\\
```

with...

```
#include "s2s_common.h"\\
#include "gainblk.h"\\
#include "plusblk.h"\\
#include "dollar4.h"\\
#include "satur.h"\\
#include "summation.h".\\
```

Note: Before the macros can be expanded, they might need a rename in a way that already written .m4x macro expansion rule definitions match (use scripts for that, e.g. monoclean.sh). Another note: Be careful when including

more than one Scicos-to-SynDEx generated file since some declarations (parameters) will mix up (for this sense I used a self-made m4x-builder java-program). Yet another note: Remove the line

```
"enum flag_type {
flag_init = 4, flag_updateOutputs = 1,
flag_updateZstates = 2, flag_updateSstates = 0,
flag_reinit = 6, flag_end = 5};"
```

and the line

```
"#include <stdio.h>"
```

from the pidFan.sdx file. Additionally, a remark about memory blocks: Dollar blocks have to be defined with double in this example (otherwise mismatching data-types): e.g. replace alloc_(int,dollar4_4_1_buf,1) with alloc_(double,dollar4_4_1_buf,1). About the number of iterations in the main-loop: In the pidFan.m4x file the number of iterations in the main-loop can be changed to forever by removing the "NBITERATIONS" definition. When doing this, maybe also remove the other comment macros:

```
dnl define(‘NOTRACEDEF’)
dnl define(‘NBITERATIONS’,‘‘1000’’)
dnl define(‘dnldnl’,‘‘// ’’)
dnl define(‘    # ’,‘‘// ’’)
```

Now it should be possible to compile it using avr-gcc: "make".

B Listings

B.1 SynDEx PID-Example, Node3.m4

```
include(syndex.m4x)dnl
sinclude(TaskTiming.m4x)dnl
sinclude(eBoard.m4x)dnl
dnl
processor_(ATmega128,node3,fscFinal6b,
SynDEx-7.0.0 (C) INRIA 2001-2009, 2010-10-03 03:02:02)

alloc_(int,Timer2Controller_fire,1)
alloc_(uint16,fanSensorDriver_rpm,1)
alloc_(double,S2sDataWrapperW_d,1)
alloc_(double,gainblk_9_1_out_1,1)
alloc_(uint8,FanSpeed_cst,1)
========== Snip ==========
...
========== Snip ==========
alias_(dollar4_4_1_out_1,dollar4_4_1_buf,0,1)
alias_(dollar4_18_1_out_1,dollar4_18_1_buf,0,1)
alias_(rem_1_out_1,rem_1_buf,0,1)

main_
  proc_init_
  HWTimer2Controller(1000,25,Timer2Controller_fire)
  eFanSensorDriver__1(fanSensorDriver_rpm)
  TTTaskReservation(1)
  eUint16ToDouble(fanSensorDriver_rpm,S2sDataWrapperW_d)
  gainblk_1__1(11,4,0,0,0,1,0,0,0,1,5,0,0,0,0,1,1,S2sDataWrapperW_d,gainblk_9_1_out_1)
  eConstantUint8(50,FanSpeed_cst)
  eUint8ToDouble(FanSpeed_cst,S2SWrapperCst_d)
  plusblk_1_1__1(12,2,0,0,0,0,0,0,0,1,5,0,0,0,0,2,1,gainblk_9_1_out_1,S2SWrapperCst_d,
      plusblk_8_1_out_1)
  gainblk_1__1(16,4,0,0,0,1,0,0,0,0,1,0,0,0,0,1,1,plusblk_8_1_out_1,gainblk_2_1_out_1)
========== Snip ==========
...
========== Snip ==========
  loop_
    HWTimer2Controller(1000,25,Timer2Controller_fire)
    switch_(Timer2Controller_fire)
      case_(0)
        TTTaskReservation(1)
      endcase_
      case_(1)
        eFanSensorDriver__1(fanSensorDriver_rpm)
        eUint16ToDouble(fanSensorDriver_rpm,S2sDataWrapperW_d)
        gainblk_1__1(11,4,0,0,0,1,0,0,0,1,5,0,0,0,0,1,1,S2sDataWrapperW_d,gainblk_9_1_out_1
            )
========== Snip ==========
...
========== Snip ==========
      endcase_
    endswitch_
    wait_(54)
  endloop_
  HWTimer2Controller(1000,25,Timer2Controller_fire)
  eFanSensorDriver__1(fanSensorDriver_rpm)
  TTTaskReservation(1)
  eUint16ToDouble(fanSensorDriver_rpm,S2sDataWrapperW_d)
  gainblk_1__1(11,4,0,0,0,1,0,0,0,1,5,0,0,0,0,1,1,S2sDataWrapperW_d,gainblk_9_1_out_1)
  eConstantUint8(50,FanSpeed_cst)
  eUint8ToDouble(FanSpeed_cst,S2SWrapperCst_d)
========== Snip ==========
...
========== Snip ==========
  proc_end_
endmain_

endprocessor_
```

Listing B.1: Node3 - SynDEx macro code

B.2 SynDEx PID-Example, Scicos-Syndex Gain Block

```
define(`gainblk_1__1',`ifelse(
MGC,`MGC',```',
MGC,`INIT',```/* init phase for block gainblk_1__1  */
pidBlocks[$1-1].type = $2;
pidBlocks[$1-1].ztyp = $3;
pidBlocks[$1-1].ng = $4;
pidBlocks[$1-1].nz = $5;
pidBlocks[$1-1].nrpar = $6;
pidBlocks[$1-1].nipar = $7;
pidBlocks[$1-1].nevout = $8;
pidBlocks[$1-1].nmode = $9;
pidBlocks[$1-1].z = &(pidz[$10]);
pidBlocks[$1-1].rpar = &pidRPAR1[$11];
pidBlocks[$1-1].ipar = &pidIPAR1[$12];
pidBlocks[$1-1].x = &(pidxt[$13]);
pidBlocks[$1-1].xd = &(pidxtd[$14]);
pidBlocks[$1-1].nx = $15;
pidBlocks[$1-1].nin = $16;
pidBlocks[$1-1].nout = $17;
pidBlocks[$1-1].nevprt = 0;
if ((pidBlocks[$1-1].evout = calloc(pidBlocks[$1-1].nevout,sizeof(double)))== NULL ) return 0;
if ((pidBlocks[$1-1].insz = malloc(sizeof(int) * pidBlocks[$1-1].nin))== NULL ) return 0;
if ((pidBlocks[$1-1].inptr = malloc(sizeof(double*) * pidBlocks[$1-1].nin))== NULL ) return 0;
if ((pidBlocks[$1-1].outsz = malloc(sizeof(int) * pidBlocks[$1-1].nout))== NULL ) return 0;
if ((pidBlocks[$1-1].outptr = malloc(sizeof(double*) * pidBlocks[$1-1].nout))== NULL ) return 0;
pidBlocks[$1-1].work=NULL;
pidBlocks[$1-1].inptr[0] = $18;
pidBlocks[$1-1].insz[0] = '$18_size_';
pidBlocks[$1-1].outptr[0] = $19;
pidBlocks[$1-1].outsz[0] = '$19_size_';
gainblk(&pidBlocks[$1-1], flag_init);
gainblk(&pidBlocks[$1-1], flag_reinit);
'',
MGC,`LOOP',```/* loop phase for block gainblk_1__1  */
pidBlocks[$1-1].inptr[0] = $18;
pidBlocks[$1-1].outptr[0] = $19;
gainblk(&pidBlocks[$1-1], flag_updateOutputs);
gainblk(&pidBlocks[$1-1], flag_updateZstates);
'',
MGC,`END',```/* end phase for block gainblk_1__1  */
pidBlocks[$1-1].inptr[0] = $18;
pidBlocks[$1-1].outptr[0] = $19;
gainblk(&pidBlocks[$1-1], flag_end);
'')')
```

Listing B.2: SynDEx Gain Block macro code

B.3 Atmel ATmega128 macro expansion definitions file

```
/* =====> ATmega128.m4x */
dnl ATmega128.
dnl Purpose: Macro Expansion File for the Atmel's ATmega128 with data type mapping.

========== Snip ==========
...
```

```
========================= Snip ===================================

#define(`lang_',`C')

#############
# DATA TYPES

# --------
# typedef_(name, size) defines a new type with its size in address units:
# interfacing data-types:
typedef_(`bool',  1) # MUST be defined
typedef_(`double', 4)
typedef_(`int', 2)
typedef_(`uint8',1)
typedef_(`uint16',2)

# mapped target data-types:
typedef_(`uint8_t', 1)
typedef_(`uint16_t', 2)
typedef_(`int8_t', 1)
typedef_(`int16_t', 2)

# MAPPING
# --------
## also types which aren't mapped have to be defined:
define(`double_map',`double')
define(`bool_map',`bool')

## the types to be mapped to different ones:
define(`int_map',`int16_t')
define(`uint8_map',`uint8_t')
define(`uint16_map',`uint16_t')
define(`int8_map',`int8_t')
define(`int16_map',`int16_t')

####################
## MEMORY ALLOCATION

## ------------
## basicAlloc_(label, memoryBank)
#define(`basicAlloc_',`_($1_type_ $1[$1_size_];)')
define(`basicAlloc_',`_($1_type_()_map() $1[$1_size_];)')

# ------------
## basicAlias_(newLabel,oldLabel[,offset=0])
define(`basicAlias_', `define(`$1_base_', $2_base_)dnl for `basicCopy_'
`#define' $1 ifelse($3,,`$2',`($2+$3)')')


####################
## CONTROL STRUCTURES

## --------
## basicIf_(buffer,value, tagforElseOrEndif)
define(`basicIf_', `_(if ($1[0]==$2) { /* $3 */)')

========================= Snip ===================================
...
========================= Snip ===================================

divert
divert``'dnl ------------------------ End of file ---------------------------
```

Listing B.3: SynDEx ATmega128 macro expansion definition file

B.4 SynDEx PID-Example, final C code

```
/* =====> eBoard.m4x */

#include "eFanDriver.h"
#include "eFanSensorDriver.h"
========================= Snip ===================================
...
========================= Snip ===================================

/*****************************************************************************/
/***** ADDITIONAL GLOBALS ****************************************************/
/*****************************************************************************/
```

```
uint8_t HWTimer0Fire;
uint8_t HWTimer1Fire;
uint8_t HWTimer2Fire;
uint8_t HWTimer3Fire;

/*****************************************************************************/
/***** ADDITIONAL COMPUTATIONAL FUNCTIONS ************************************/
/*****************************************************************************/

/* Called when Timer reached destination Time (n * OC-1) */
void HWTimer2Handler(void){
  HWTimer2Fire = 1;
}

/* Reset the Fire-global after 1 repetition */
uint8_t HWTimer2ResetFire(uint8_t fire){

  static uint8_t fireMem;

if(fire==1&&fireMem==1){
    HWTimer2Fire=0;
    fireMem=0;
  }
  if(fire==1&&fireMem==0){

    fireMem=1;
  return 1;
  }
  if(fire==0){
  return 0;
  }

  /* never reached */
  return 0;
}

========================= Snip ===================================
...
========================= Snip ===================================


/* SynDEx-7.0.0 (C) INRIA 2001-2009, 2010-10-03 03:02:02,
    application fscFinal16b, processor node3 type=ATmega128 */
/* =====> ATmega128.m4x */

/* 'alloc_(int,Timer2Controller_fire,1)' */
int16_t Timer2Controller_fire[1];
/* 'alloc_(uint16,fanSensorDriver_rpm,1)' */
uint16_t fanSensorDriver_rpm[1];
/* 'alloc_(double,S2sDataWrapperW_d,1)' */
double S2sDataWrapperW_d[1];
/* 'alloc_(double,gainblk_9_1_out_1,1)' */
double gainblk_9_1_out_1[1];
========================= Snip ===================================
...
========================= Snip ===================================

/* 'main_()' */
int main(int argc, char* argv[]) { /* for link with C runtime boot */

========================= Snip ===================================
...
========================= Snip ===================================

/* init phase for block gainblk_1__1  */
blocks[11-1].type = 4;
blocks[11-1].ztyp = 0;
blocks[11-1].ng = 0;
blocks[11-1].nz = 0;
blocks[11-1].nrpar = 1;
blocks[11-1].nipar = 0;
blocks[11-1].nevout = 0;
blocks[11-1].nmode = 0;
blocks[11-1].z = &(z[1]);
blocks[11-1].rpar = &RPAR1[5];
blocks[11-1].ipar = &IPAR1[0];
blocks[11-1].x = &(xt[0]);
blocks[11-1].xd = &(xtd[0]);
blocks[11-1].nx = 0;
blocks[11-1].nin = 1;
blocks[11-1].nout = 1;
blocks[11-1].nevprt = 0;
if ((blocks[11-1].evout = calloc(blocks[11-1].nevout,sizeof(double)))== NULL ) return 0;
if ((blocks[11-1].insz = malloc(sizeof(int) * blocks[11-1].nin))== NULL ) return 0;
if ((blocks[11-1].inptr = malloc(sizeof(double*) * blocks[11-1].nin))== NULL ) return 0;
```

```
    if ((blocks[11-1].outsz = malloc(sizeof(int) * blocks[11-1].nout))== NULL ) return 0;
    if ((blocks[11-1].outptr = malloc(sizeof(double*) * blocks[11-1].nout))== NULL ) return 0;
    blocks[11-1].work=NULL;
101 blocks[11-1].inptr[0] = S2sDataWrapperW_d;
    blocks[11-1].insz[0] = 1;
    blocks[11-1].outptr[0] = gainblk_9_1_out_1;
    blocks[11-1].outsz[0] = 1;
    gainblk(&blocks[11-1], flag_init);
106 gainblk(&blocks[11-1], flag_reinit);
    ========================== Snip ================================
    ...
    ========================== Snip ================================

111   /* 'loop_()' */
      for(;;){ /* loop_2 */
      {int i;for (i=0;i<1;i++)  rem_1_out_1[i] = prod_22_1_out_1[i];}
                  Timer2Controller_fire[0]=HWTimer2ResetFire(HWTimer2Fire);
        /* 'switch_(Timer2Controller_fire)' */
116     switch (Timer2Controller_fire[0]) { /* switch_3 */
          /* 'case_(0)' */
          case 0 : /* case_4 */
            eNOP(26); //1
          /* 'endcase_()' */
121         break; /* case_4 */
          /* 'case_(1)' */
          case 1 : /* case_5 */
            /* loop phase for block eFanSensorDriver__1  */
                  eFanSensorDriver(fanSensorDriver_rpm);
126         /* loop phase for block eUint16ToDouble  */
                S2sDataWrapperW_d[0] = (double)fanSensorDriver_rpm[0];
            /* loop phase for block gainblk_1__1  */
    blocks[11-1].inptr[0] = S2sDataWrapperW_d;
    blocks[11-1].outptr[0] = gainblk_9_1_out_1;
131 gainblk(&blocks[11-1], flag_updateOutputs);
    gainblk(&blocks[11-1], flag_updateZstates);
    ========================== Snip ================================
    ...
    ========================== Snip ================================
136   /* 'endcase_()' */
            break; /* case_5 */
        /* 'endswitch_()' */
          } /* switch_3 */
        eNOP(54);
141   /* 'endloop_()' */
      } /* end loop_2 */

      /* end phase for block gainblk_1__1  */
    blocks[11-1].inptr[0] = S2sDataWrapperW_d;
146 blocks[11-1].outptr[0] = gainblk_9_1_out_1;
    gainblk(&blocks[11-1], flag_end);

    ========================== Snip ================================
    ...
151 ========================== Snip ================================
    /* 'endmain_()' */
      return 0;
    } /* end of main */

156 /* 'endprocessor_()' */
```

Listing B.4: SynDEx PID-Example C code

B.5 SynDEx PID-Example, GNUMakefile

```
    A=fanSpeedControlMonoProcessorStage03
    M4=m4

  4 # these paths have to be modified by user
    sdx_MACRO_PATH=/home/exa/DA/syndex-7.0.0/macros

    eBoard_MACROS_PATH=/home/exa/DA/code/syndex/macros
    eBoard_MACROS_ALGORITHMS_PATH=$(eBoard_MACROS_PATH)/algorithms
  9 eBoard_MACROS_ARCHITECTURES_PATH=$(eBoard_MACROS_PATH)/architectures
    eBoard_BUILDENV_PATH=/home/exa/DA/code/eBoardBuildEnv
    eBoard_DRIVERS_SRC_PATH=$(eBoard_BUILDENV_PATH)/drivers/src/syndex
    eBoard_DRIVERS_INC_PATH=$(eBoard_BUILDENV_PATH)/drivers/include/syndex
    eBoard_CONFIG_PATH=$(eBoard_BUILDENV_PATH)/config
 14
```

```
# Include general properties for the environment
include $(eBoard_BUILDENV_PATH)/Makefile.set

# Algo_Macros_Path: Path of the algorithm specification files (.m4x and .m4m).
# Archi_Macros_Path: Path of the architecture specification files (.m4x and .m4m).
# S2s_Files_Path: User-made Scicos blocks must be copied in here (.c).
# eBoard_Drivers_Src_Path: User-made Scicos/Syndex computational function blocks (.c and .h
    ).

export syndex_Macros_Algorithms_Path=$(sdx_MACRO_PATH)/algo_libraries
export syndex_Macros_Architectures_Path=$(sdx_MACRO_PATH)/archi_libraries

export eBoard_Macros_Algorithms_Path=$(eBoard_MACROS_ALGORITHMS_PATH)
export eBoard_Macros_Architectures_Path=$(eBoard_MACROS_ARCHITECTURES_PATH)
export eBoard_Drivers_Src_Path=$(eBoard_DRIVERS_SRC_PATH)

export S2s_Files_Path=/home/exa/DA/syndex-6.8.5/SCICOS_FILES

export M4PATH=$(syndex_Macros_Algorithms_Path):$(syndex_Macros_Architectures_Path):$(
    S2s_Files_Path):$(eBoard_Macros_Algorithms_Path):$(eBoard_Macros_Architectures_Path)

VPATH=$(M4PATH)

#CFLAGS = -DDEBUG -lm -Wall -pedantic

.PHONY: all expand cleanall cleanit cleanc

# replace rootOperator.c by $(A).run in order to run the application with a simple make
    call
#all : node3.c
all : expand

#clean ::
# $(RM) \#* *~ *.o *.a *.mnt node3 $(A).mk

expand: $(A).mk

node: node3.c

debugPaths:
  @echo "M4PATH="$(M4PATH)

cleanit : cleanc
  $(RM) $(A).mk

cleanall :
  $(RM) node3.m4

cleanc :
  $(RM) node3.c

$(A).mk : $(A).m4 syndex.m4m ATmega128.m4m $(A).m4m
  $(M4) $< >$@


node3.libs = $(eBoard_DRIVERS_SRC_PATH)/cstblk4.o \
 $(eBoard_DRIVERS_SRC_PATH)/eFanDriver.o \
========================= Snip ===================================
...
========================= Snip ===================================

node3.inc = $(eBoard_DRIVERS_INC_PATH)

# the $(A).mk file is generated by the makefile process, do not edit/modify unless you know
    exactly what you are doing
sinclude $(A).mk
```

Listing B.5: SynDEx PID-Example GNUMakefile

B.6 SynDEx PID-Example, .mk Makefile

```
# SynDEx-7.0.0 (C) INRIA 2001-2009, 2009-10-13 03:54:51
# Makefile for application fanSpeedControlMonoProcessorStage03

# $(M4) must be the GNU macroprocessor m4
# $(Archi_Macros_Path) must be the path to the generic *.m4? macro-files
# $(VPATH) is searched by make for dependent files not found in $(PWD)
# VPATH = $(Archi_Macros_Path)
```

```
.PHONY: fanSpeedControlMonoProcessorStage03.all fanSpeedControlMonoProcessorStage03.run
     clean
fanSpeedControlMonoProcessorStage03.run : fanSpeedControlMonoProcessorStage03.all # load
     and run fanSpeedControlMonoProcessorStage03:
        # (command args = processors in loading order)
  ./$(fanSpeedControlMonoProcessorStage03.root)

# processor node3 type=ATmega128:
fanSpeedControlMonoProcessorStage03.all : node3
fanSpeedControlMonoProcessorStage03.root += node3

#START-procr- ---------------------------------- START OF PROCESSOR MAKEFILE
# ATmega128.m4m
# Author: exa
# Purpose: Macro expansion file for Atmel's ATmega128.
# Description: Generates the architecture specific makefile.

node3.c : node3.m4 syndex.m4x ATmega128.m4x eBoard.m4x
  $(M4) $< >$@

node3.OBJS        = node3.o
node3.SUBOBJS     = $(node3.libs)

# define variables containing the names of the hex & eep file
node3.FLASH_FILE = node3.hex
node3.EEP_FILE    = node3.eep
node3.ELF_FILE    = node3.elf

========================= Snip ======================================
...
========================= Snip ======================================

procr_SUBDIRS     = $(dir $(node3.SUBOBJS)) # just the dirs "Sub1/ Sub2/ ..."

========================= Snip ======================================
...
========================= Snip ======================================
# list of all non-file targets
.PHONY: all makesub lc lmc clean mostlyclean distclean install debug


# 1st the main target
all: $(DREQUIRED) makesub node3.hex node3.eep


# include just existing D-files
ifneq "$(strip $(DFILES))" ""
  include $(DFILES)
endif


# declare implicit rules
%.hex:  %.elf
  $(OBJCOPY) -O $(FORMAT) $< $@

%.eep:  %.elf
  -$(OBJCOPY) $(EEPFLAGS) -O $(FORMAT) $< $@

%.elf:  $(node3.OBJS) $(node3.SUBOBJS)
  $(CC) -Wl,-Map=$*.m $(LDFLAGS) -o $@ $(node3.OBJS) $(node3.SUBOBJS) $(LIBOPT)

========================= Snip ======================================
...
========================= Snip ======================================

%.o:  %.c
  $(CC) -I$(node3.inc) $(CFLAGS) -Wall -c -o $@ $<

%.o:  %.S
  $(CC) -I$(node3.inc) $(CFLAGS) -Wall -c -o $@ $<

%.i:  %.c
  $(CC) -I$(node3.inc) $(CFLAGS) -E -o $@ $<

%.s:  %.c
  $(CC) -I$(node3.inc) $(CFLAGS) -S -o $@ $<

========================= Snip ======================================
...
========================= Snip ======================================
include $(eBoard_CONFIG_PATH)/node3.deploy
#END-procr- ------------------------------------ END OF PROCESSOR MAKEFILE
```

Listing B.6: SynDEx PID-Example .mk Makefile

B.7 Scicos Block Struct, .h Header

```
#ifndef __SCICOS_BLOCK_H__
#define __SCICOS_BLOCK_H__

#include <stdint.h>
#include <stdlib.h>

typedef void (*voidg)(void);

typedef struct {
  int16_t nevprt;
  voidg funpt ;
  int16_t type;
  int16_t scsptr;
  int16_t nz;
  double *z;
  int16_t nx;
  double *x;
  double *xd;
  double *res;
  int16_t nin;
  int16_t *insz;
  double **inptr;
  int16_t nout;
  int16_t *outsz;
  double **outptr;
  int16_t nevout;
  double *evout;
  int16_t nrpar;
  double *rpar;
  int16_t nipar;
  int16_t *ipar;
  int16_t ng;
  double *g;
  int16_t ztyp;
  int16_t *jroot;
  char *label;
  void **work;
  int16_t nmode;
  int16_t *mode;
} scicos_block;



/* extern void do_cold_restart();
int get_phase_simulation(void){return 1;}
double get_scicos_time();
int16_t get_block_number();
void set_block_error(int16_t);
void set_point16_ter_xproperty(int16_t* point16_ter);

void * scicos_malloc(size_t );
void scicos_free(void *p);
*/

#define max(a,b) ((a) >= (b) ? (a) : (b))
#define min(a,b) ((a) <= (b) ? (a) : (b))

/*
extern int16_t s_copy();
extern int16_t s_cmp();
*/
enum flag_type { flag_init = 4, flag_updateOutputs = 1, flag_updateZstates = 2,
     flag_updateSstates = 0, flag_reinit = 6, flag_end = 5};

#endif /* __SCICOS_BLOCK_H__ */
```

Listing B.7: Scicos Block Structure .h Headerfile

B.8 Textual SynDEx PID algorithm, pidFan.sdx

```
syndex_version : "7.0.0"
application description : "Algorithm generated by the Scicos To SynDEx translator"

# Libraries
include "Atmel.sdx";
include "TaskTiming.sdx";
include "eBoard.sdx";
include "pid.sdx";
include "eBoardDrivers.sdx";

# Algorithms
def algorithm FanSpeedControlMonoProcessor  :
conditions: true;
references:
 eBoardDrivers/eFanSensorDriver__1 fanSensorDriver @-188,133;
 eBoardDrivers/eFanDriver_1_<0> fanMotorDriver @379,71;
 eBoardDrivers/eDoubleToUint8 S2SDataWrapper @242,71;
 eBoardDrivers/eUint16ToDouble S2sDataWrapperW @-44,133;
 eBoardDrivers/eConstantUint8<50> FanSpeed @-135,38;
 eBoardDrivers/eUint8ToDouble S2SWrapperCst @-7,38;
 TaskTiming/TTTaskContainer TTSpeedControl @193,-39;
 TaskTiming/HWTimer2Controller<1000;25> Timer2Controller @-21,-39;
 pid/pidControllerFinal PID @135,66;
dependences:
 strong_precedence_data fanSensorDriver.rpm -> S2sDataWrapperW.uint16;
 strong_precedence_data S2SDataWrapper.uint8 -> fanMotorDriver.speed;
 strong_precedence_data S2sDataWrapperW.d -> PID.y_i;
 strong_precedence_data FanSpeed.cst -> S2SWrapperCst.uint8;
 strong_precedence_data S2SWrapperCst.d -> PID.w_i;
 strong_precedence_data Timer2Controller.fire -> TTSpeedControl.fire;
 strong_precedence_data PID.u_i -> S2SDataWrapper.d;
code_phases:  loopseq;


# Architectures

# Main Algorithm / Main Architecture
main algorithm FanSpeedControlMonoProcessor ;
main architecture eBoard/monoProcNode3;

# Extra durations

# Operation groups, previously called Software components

# Constraints
```

Listing B.8: Textual SynDEx algorithm