Abstract

This paper presents a model for estimating the total cost of a proposed electronic architecture for embedded, distributed, real-time systems. The main purpose of the model is to allow system engineers to compare different solution alternatives with respect to cost, in order to perform an early optimization. This is necessary to cope with the complexity of such systems, and make them profitable for the company producing them. The total cost includes product cost, i.e. the cost of hardware components, hardware development, and software development. The model is modular to allow modifications in order to deal with specific project circumstances. The modelling approach is also general, and could easily be extended to include other characteristics such as weight, reliability, or power consumption, or be used to model other systems than electronics.

1. Introduction

When a new product is designed, one of the main targets is usually to reduce the cost. Still, much of the engineering effort is put into increasing performance and functionality. One of the reasons for this is likely to be that engineers have good models for estimating the performance, which makes it easy to take those factors into account during early design, whereas models for cost estimation are scarce. This paper is an initial attempt to provide a cost estimation model for distributed electronic systems. The main contribution is a well-structured and general model that covers the total cost of the complete system, including both development and product cost. To our knowledge, no such models have been presented elsewhere. Although the model is based on experience from the automotive industry, it is sufficiently general to also be valid in other areas.

1.1. Motivation

In the early concept study phases of an electronic system development project, the engineering team has to decide on issues like:

- What electronic control units (ECUs) to include, and where to place them physically in the surrounding system.
- What functionality, i.e., software, to put in each ECU.
- What communication paths to include between ECUs.

In many projects, these decisions are made without a conscious and systematic examination of the alternative solutions, with respect to important characteristics such as cost. However, this lack of attempt to optimize the cost in the early phases might severely affect the final cost of the product. During the concept study, there is a large cost span between different alternatives, and the engineers thus have a large possibility to control the cost. Once the concept has been frozen and detailed development starts, only smaller modifications of the cost are possible. Figure 1 (adapted from [10]) illustrates this phenomenon. The lower curve in the figure shows the engineering effort during different stages of development, and it is clear that moving a small amount of effort to the concept study phase can have a large effect on the total cost, if it is spent on optimizing the concept. Cost estimation models play an important role in that optimization, since they allow the engineers to make trade-offs between different alternative designs. Once the design is frozen, the cost estimations become budgets on the parts of the design, which makes it easier to control the project as it proceeds, and the use of formal models increases the realism of the budgets, thus reducing the project risk.

1.2. Requirements on the model

When developing a model, it is important to state what requirements it should fulfill to be useful. In our experience, the main requirements for early cost estimation are:

- **Usability.** The model should be easy to use, and not require a large amount of extra work to gather the data needed. Instead, it should share data that is collected for...
other purposes, and the calculations should be possible to automate.

- **Flexibility.** The model should provide a default solution for typical situations, but also allow the user to modify parts of it to encompass issues that are particular to the system being developed. Therefore, the model has to be flexible.

- **Accuracy.** The model should give estimations that are as accurate as possible given the information available, and also allow the user to assert the uncertainty of the results.

- **Suitability.** The model should be adapted to the situation where it is intended to be used. A model for early estimations may thus not assume the availability of information that is usually not present until late in the development process.

### 1.3. Limitations of early estimation

As mentioned above, the current practice is to make very little cost estimations early in the projects. One reason for this is that the values will always be of limited precision, and it is important to be aware of the fact that the work done in the early design phases is not "rocket science", i.e. one needs to be able to cope with vagueness. Still, as pointed out in [8], by using systematic models it is possible to get both better estimates and a better understanding of the precision of the estimate, than is possible with the rules of thumb used today. Also, it is easier to assess the technical risks of the project using estimation models.

A fundamental difficulty when working with estimation models for early design is to determine what value to compare the estimates with in order to determine the precision. One might assume that it would be possible to make an estimate, then let the development proceed, and once the final product is ready compare the estimate to the actual value. However, the early phases only provide limits on what solutions are to be investigated during detailed design, which means that there are many degrees of freedom to be used for optimization during later phases. The outcome of the early phases is thus a set of potential solutions, and the estimation describes the characteristics of one member of that set. The designers are free to choose any other member if they, based on information becoming available in later phases, decide that the other solution is better. (For a detailed discussion of these issues, see [2].)

### 1.4. Object-oriented modelling

Since different alternative electronic architectures may have very different structure, it is important to have models that can easily be composed in various ways without having to remake them from scratch. Usually, the architectures are built from standard electronic components, and it would simplify the system analysis if models of such components were available in a library. Also, the models can become quite large, and it is therefore essential to be able to structure them in a clear way.

All these characteristics are typical of so-called object-oriented (OO) models, that are popular for describing software, and we will therefore adopt OO descriptions using the Unified Modelling Language (UML) [11]. Software and system development is often done using OO methods, and

![Figure 1. The allocation of product cost and engineering effort as development proceeds.](image-url)
the cost models could thus be included in the normal tools, allowing estimations to be done with little extra work.

1.5. Overview of paper

In the next section, the assumptions that are made about the contents of the system specification are presented. It is important to have a clear picture of this information, since it is the basis for selecting the solution. For electronic systems, one of the cost drivers is the software, and it is therefore necessary to have models for estimating characteristics of software based on the specification. Such models are discussed in Section 3. Then, in Section 4, the structure of the electronic system is presented together with the cost equations. The development cost is treated in more detail in Section 5, and in Section 6, it is discussed how to apply the models in trade studies. Finally, in Section 7, the conclusions are summarized and indications are given how to improve the models further.

2. Information model of system specification

In this section, it is described what information the system specification is assumed to contain. An overview of the classes and their relations are given in Figure 2. The structure is general and includes:

- **Specification**: describes the system to be built. It consists of an arbitrary number of functions, interfaces, performance requirements, and constraints.
- **Function**: describes something that the system is required to do. The functions are assumed to be described as a sequence of interactions with the environment and changes to the internal state of the system. They are related to an arbitrary number of interfaces and performance requirements.
- **Interface**: describes an interaction point between the system and the surrounding world. It includes information about its type (i.e., what kind of data it handles) and its physical position. It is related to an arbitrary number of functions.
- **Performance requirement**: describes how well the system is to perform its functions. This includes the rate of invocation and response time of functions involved in real-time control tasks. It is related to an arbitrary number of functions.
- **Constraint**: describes a restriction on the design. It is specialized by ECU position.
- **ECU position**: describes a place in the environment where an ECU may be placed. This is a constraint due to the overall layout of the surrounding system, and includes information about e.g., temperature, humidity, vibrations, and EMC. We capture all these issues in the attribute environmentFactor, which is a multiplicative factor summarizing how much the environment increases the cost of the ECU housing and connectors.

3. Estimating software characteristics

The purpose of the system is to perform the set of functions described in the specification, while fulfilling the performance requirements and the constraints. Since the functions in an electronic architecture are mostly implemented in software, it is critical to be able to estimate the following characteristics of the software modules:

- **Program size**: determines the amount of non-volatile memory needed, and also indicates the effort needed to develop the software.
- **Data size**: determines the amount of volatile memory needed.

![Figure 2. Information model of the system specification.](image-url)
• **Execution effort**: determines the processing power needed to meet the performance requirements.

During the design step, the functions in the specification are refined into a set of software modules, where each function is performed by a set of modules in collaboration. The software modules are small enough to execute on a single ECU. The problem is thus to assess the characteristics of each of the software module.

### 3.1. Function points

Based on the information available in the specification, there is no possibility to exactly measure the characteristics of the software. The requirements only describe the behaviour, and thus the most one can do is to quantify the complexity of that behaviour, and hope that there is a relation between the complexity and the actual characteristics of the software.

A common technique to measure the complexity is through so-called **function points** (FP). This technique was originally used for large transaction-based software with a lot of file handling, but variants that are better suited to handle real-time systems have been developed (see e.g. [5] and [7]). The basic idea is to count the number of interactions with the external environment of the ECU and with its data storage. These operations work on the data attributes of the system, which should be inherent in the specification. The operations are counted, and weighed with a factor depending on the kind of operation, and the grand total gives the number of FP.

### 3.2. Program size

FPs is an abstract measure of complexity, and to make use of it to estimate e.g. the memory size or development time, it is useful to transform it into something more concrete. Probably, the most common metric on program size is the number of lines of source codes. However, this metric must be used with care, since it depends on the language used. A high-level language needs fewer lines of code than assembly, and a graphical language does not even have lines. It also depends on the coding style of the programmer.

Another issue is the size of the program memory. Assuming that the software is written in a high-level language and compiled, the memory size depends on the language and compiler, and also on the processor, since the instructions of different processors vary in size. The model thus needs to be calibrated to the specific situation.

### 3.3. Data size

The amount of data memory stems from two sources: permanent data, which is repeatedly updated and referenced throughout the execution; and transient data which is stored on the stack during the execution of an algorithm or subroutine call. The permanent data can to a large extent be gathered from the specification, since it is mostly mentioned directly or indirectly in the functions. The transient data is somewhat more difficult, but probably there is a relation to the complexity of the function, i.e. the number of FPs. However, in most cases the permanent data is the bulk of the memory, since only a small portion of all the possible volatile data is stored at any given moment.

### 3.4. Execution effort

The execution effort of a segment of software depends on the number of instructions involved. Basically, the processor performs three types of operations:

- Read instructions from the program memory.
- Read or write data from the data memory.
- Perform internal calculations.

Usually, simple calculations such as additions or logical operations require no extra effort, apart from the time needed to fetch the instructions. The amount of instructions to read is proportional to the size of the software that make up that segment, which is given by the FPs as described above. Since the amount of data accesses is also given directly by the FPs, they are likely to be a good indicator of the performance required.

The size and execution effort are related in that they measure the same set of FPs, and they have identical values if the operation is a straight code sequence. However, if there are loops the effort increases since some parts of the code is executed several times, but the size remains the same. Also, if there are conditionals, parts of the code may not be executed at all in certain invocations, meaning that the size could be larger than the effort.

### 3.5. Processor load

Once the execution effort of the code segments has been established, this value can be used to obtain the performance needed from the processor. It is then necessary to also consider the invocation rate of the functions, and the processor load for a given processor is the sum of the execution time of each function multiplied by its invocation rate.

In most cases, several processes are executed in parallel, and this means that the scheduling has to be taken into account. If priority-based scheduling is used, a rule of thumb is that the processor load should not be above 70% to guarantee all timing constraints, assuming that it is suffi-
cient that each process finishes before it needs to be activated again [9]. If there should be deadlines that are substantially shorter than the period of the process, the processor performance needed has to be determined by this peak processing requirement. For other scheduling strategies, similar guidelines might be found.

4. Product cost estimation models

Based on the specification, synthesis is performed to find a suitable set of parts to implement the specification. The parts are organized hierarchically in a system breakdown structure (SBS) which forms the basis for the cost model. An overview of the SBS is given in Figure 3. Three levels are covered: the product, the parts that make up the product, and the components that the parts consist of. The system design process decides what parts the system should consist of, and we do therefore not decide upon the components at this stage. The reason for including the components in the model is that the cost estimations for the parts are based upon “intelligent guesses” about the components they might consist of. The cost estimations are expressed as invariants which relate the attributes of different elements.

4.1. Product

The top node in the SBS is the product being developed, which in this case is the electronic architecture. It is related to a specification, which it implements, and consists of an arbitrary number of parts. Attributes:
- totalCost: the total cost of each copy of the system up to delivery [$].
- productCost: the cost of building one additional copy of the system [$].
- devCost: the cost of developing the system [$].
- numberOfUnits: the expected number of units manufactured of the system [-].
- swDevEffort: the effort in man months (MM) required to develop the software in the system [MM].
- hwDevEffort: the effort required to develop the hardware in the system [MM].

Invariant: The total cost of one unit is the product cost, plus the development cost divided by the number of units:

\[
\text{totalCost} = \text{productCost} + \frac{\text{devCost}}{\text{numberOfUnits}}
\]

The discounting factor captures the fact that the cost of development must be seen as an investment which is paid back with an interest when the product is sold. It is beyond the scope of this paper to describe how it is calculated, but we note that it is a significant factor in many cases. For automotive systems, which have a development time over several years, it may very well be on the order of 1.5.

The product cost is the sum of the product cost of all its parts:

\[
\text{productCost} = \sum_{p \in \text{part}} p.\text{productCost}
\]

The development cost is calculated as follows:

\[
\text{devCost} = (\text{swDevEffort} + \text{hwDevEffort}) \cdot \text{manCost}
\]

where \(\text{manCost}\) is the cost of one MM.

The derivation of development effort for hardware and software is discussed further in Section 5. In this paper, we do not treat the investment cost to set up manufacturing, nor the cost of handling spare parts on the after market, since these issues vary considerably between different applications. Also, by saying that the product cost of the system equals the sum of the product cost of the parts, we neglect the processing cost of assembling the parts, but this can be justified by the high degree of automation in the electronic industry.

4.2. Parts

The product is decomposed into parts, and an overview of the SBS on the part level is given in Figure 4. The different parts are presented in the following subsections.
4.2.1. Part

A part is a piece of equipment being connected together during assembly. It consists of an arbitrary number of components. It is related to an arbitrary number of functions that are allocated to the part and an arbitrary number of interfaces that are used by it. **Attributes:**

- **productCost**, i.e. the product cost of the part [$].

**Invariant:** The product cost is the sum of the product cost of all its components:

\[ \text{productCost} = \sum_{c \in \text{component}} c.\text{productCost} \]

4.2.2. Wiring harness

A wiring harness is a part that consists of two connectors and an arbitrary number of wires. **Attributes:**

- **coatingCost**, i.e. the cost of adding wiring harness coating to a wire [$/m].

**Invariant:** The product cost of the wiring harness equals the sum of the cost of all the wires it contains, plus the cost of the coating that holds them together. Since the coating is a function of the length and the number of wires, we include this cost as a factor multiplied by the cost of the wires themselves:

\[ \text{productCost} = \text{coatingCost} \cdot \sum_{w \in \text{wire}} w.\text{productCost} \]

4.2.3. ECU

An ECU is a part which is capable of performing some of the system's functionality. It is placed in one ECU position and is associated with an arbitrary number of software modules, which it executes. The components of an ECU are: one housing, one processor, one program memory, one data memory, one PCB, an arbitrary number of peripheral circuits, and an arbitrary number of connectors.

4.2.4. Interface unit

An interface unit is a part which implements an interface to the surrounding world, i.e. a sensor or an actuator. It is related to one interface and consists of one connector. For specific interface units, the product cost is provided as a constant.

4.2.5. Software module

A software module is a part that participates in the implementation of functions. It consists of an arbitrary number of attributes and an arbitrary number of operations, and is related to an ECU which executes it.

**Invariant:** There is no product cost associated with software modules, \( \text{productCost} = 0 \).

4.2.6. Attribute

An attribute is a piece of data that is stored in a software module. **Attributes:**

- **size**, i.e. the size of the data held in the attribute [byte].

---

**Figure 4. An overview of the parts in the system breakdown structure for electronic architectures.**
4.2.7. Operation

An operation is a piece of executable code in a software module. Attributes:
- size: the size of the operation [FP].
- effort: the maximum number of FPs performed when this operation is invoked, i.e. the length of the longest execution path through the code of the operation [FP].
- invocationRate: how often the operation is invoked, as a maximum [-/s].

4.3. Components

The parts are decomposed into components. An overview of the components in the SBS is given in Figure 5, and in the following subsections, the components are presented in greater detail. It should be noted that certain real processor components contain some peripheral circuits or memory on chip, which may for other components have to be bought separately to provide a certain functionality. At the stage of system concept definition, it is unnecessary to determine exactly which of these alternatives should be used; that is a task of the ECU designers. However, by noting what circuits are necessary, a reasonable estimate of the cost can be given.

4.3.1. Component

A component is the smallest piece of equipment that is necessary to identify to perform the analysis. It is a part of a part. Attributes:
- productCost, i.e. the product cost of the component [$].

4.3.2. Electronic component

An electronic component is mounted on the circuit board of an ECU. Attributes:
- size, i.e. the size of the electronic component [m²].

4.3.3. Connector

A connector is a component used to interface two other parts electrically. It is related to one other connector, to which it is connected during system assembly. Attributes:
- numberOfPins: the number of pins of the connector [-].
- startCost: the cost of an imaginary 0-pin connector [$].
- pinCost: the cost increase for each additional pin [$].

The value of the startCost and pinCost attributes depends on the technology used, which depends on the environmental harshness, in particular its humidity, and the number of times the connector needs to be disconnected during the

![Figure 5. Components of the electronic architecture (showing only some attributes).]
4.3.4. Wire

A wire is a component used to carry electrical signals between two parts. It is part of a wiring harness. Attributes:
- length: the length of the wire [m].
- meterCost: the cost per meter of the wire [$/m].

Invariant: The cost is a function of the length, i.e.
\[
\text{productCost} = \text{length} \cdot \text{meterCost}.
\]

4.3.5. Housing

The housing is the box that contains all the other components of the ECU. It protects them from the environment, and vice versa. Attributes:
- sizeCost, i.e. the cost of the housing needed for the PCB [$/m^2$].

Invariant: The cost of the housing are functions of: the environmental harshness of the ECU position where it is placed; the need of cooling, which is determined by the placement of the ECU; and its size, which depends on the size of the PCB:
\[
\text{productCost} = \text{env} \cdot \text{ecu.pcb.size} \cdot \text{sizeCost}
\]

where \(\text{env} = \text{ecu.ecuPosition.environmentFactor}\). (In a more advanced model, one might also wish to include the size of the connector, which is a function of the number of pins, as a driving factor of the housing size.)

4.3.6. PCB

The PCB (printed circuit board) is the component where the electronic components of the ECU are mounted and connected together. Attributes:
- size: the size of the PCB [m²].
- wiringSize: the increase to the size of the board needed to hold the wiring [%].
- sizeCost: the cost of a PCB of a certain size [$/m^2$].

Invariant: The size depends on: the size of the individual components mounted on the board and the amount of interconnections:
\[
\text{size} = \frac{100 + \text{wiringSize}}{100} \sum_{c \in \text{cs}} \text{c.size}
\]

where \(\text{cs} = \text{ecu.electronicComponent}\). The product cost is a linear function of the size, \(\text{productCost} = \text{sizeCost} \cdot \text{size}\).

4.3.7. Memory

A memory is an electronic component in an ECU with the ability of storing information. It is specialized by program memory and data memory. Attributes:
- capacity: the amount of data the memory can hold [byte].
- byteCost: the cost per byte [$/byte$].
- spareCapacity: the percentage of the capacity which is unused [%].

Invariant: The cost and size are functions of the capacity of the memory:
\[
\text{productCost} = \text{capacity} \cdot \text{byteCost}
\]

4.3.8. Program memory

A program memory contains the instructions to be performed by the processor of the ECU. This is usually a FLASH type of memory.

Invariant: The capacity of the program memory is the sum of the program size of all software components that are part of the ECU:
\[
\text{capacity} = \frac{100 + \text{spareCapacity}}{100} \cdot \sum_{\text{ops} \in \text{fpsize}} \text{ops.size}
\]

where \(\text{ops} = \text{ecu.softwareModule.operation}\) and \(\text{fpsize}\) is the factor which approximates the number of bytes per FP.

4.3.9. Data memory

A data memory contains temporary data stored by the ECU during its computation. This is usually an SRAM or DRAM type of memory.

Invariant: The capacity of the data memory is the sum of the data size of all software components that are part of the ECU:
\[
\text{capacity} = \frac{100 + \text{spareCapacity}}{100} \cdot \sum_{\text{atts} \in \text{atts}} \text{atts.size}
\]

where \(\text{atts} = \text{ecu.softwareModule.attribute}\). (The transient data has been neglected here, but it could either be added directly or included in the spare capacity.)

4.3.10. Processor

A processor is an electronic component executing the software of an ECU. Attributes:
• capacity: the required execution speed of the processor [FP/s].
• spareCapacity: spare capacity that the processor should have [%].
• speedCost: the cost of execution speed [$/(FP/s)].

**Invariant:** The capacity of the processor depends on the functionality performed by it and the execution rate:

\[
\text{capacity} = \text{spare} \cdot \sum_{o \in \text{ops}} o \cdot \text{effort} \cdot o \cdot \text{invokationRate}
\]

where \( \text{ops} = \text{ecu}.\text{softwareComponent}.\text{operation} \) and

\[
\text{spare} = \frac{100 + \text{spareCapacity}}{100}
\]

The cost is modelled as a linear function of the capacity, \( \text{productCost} = \text{capacity} \cdot \text{speedCost} \).

### 4.3.11. Peripheral circuits

A peripheral circuit is typically an electronic component in the ECU needed to adapt inputs and outputs to make them suitable to the processor. In this paper, we do not analyze these further, but in a more detailed model, this class would be specialized for different components, including communication network interface circuits; interrupt controllers; DMA (direct memory access) controllers; DRAM controllers (if DRAM is used as data memory); timers; power supply circuits; input circuits (e.g. analogue, analogue high-frequency, digital); and output circuits (e.g. analogue effect, digital effect, analogue levels, digital). The need for input and output circuits depends on the kind of sensors and actuators used, which is in turn dependent on the interfaces in the system specification.

### 5. Development cost

In this section, we discuss how to estimate the development cost. We treat the cost for software and hardware separately, since they largely depend on different factors. It should be noted that in the previous section, when product cost was discussed, the number of produced units determined the cost, whereas when development cost is the issue, it is the complexity of the design that is of interest.

#### 5.1. Software development

Software engineering projects have been plagued with schedule and budget over-runs for decades, and therefore researchers have attempted to predict the effort of developing software. A widely accepted model is Barry Boehm’s *Constructive Cost Model (COCOMO)*, and the current COCOMO version from 1995 is described in [3]. It is beyond the scope of this paper to describe the details of COCOMO, but we recapitulate the main points, and show how it is connected to other parts of our model.

The basic formula for COCOMO gives the number of MM needed to conclude a project:

\[
\text{swDevEffort} = A \cdot \text{swSize}^B
\]

The software size (in thousands of lines of source code) is derived from the function points:

\[
\text{swSize} = \frac{\text{fpSourceCode}}{1000} \cdot \sum_{o \in \text{ops}} o \cdot \text{size}
\]

where \( \text{fpSourceCode} \) is the number of lines of source code of the selected programming language that result from one FP, and \( \text{ops} = \text{softwareModule}.\text{operation} \). The exponent \( B \) captures the overhead for communication etc. in large projects. Normally, \( B > 1 \), i.e. the effort grows faster than linear with the size of the project, and the exact value is determined based on the maturity of the development practices in the organization.

The multiplicative factor \( A \) is a product of about 20 factors describing characteristics of the project and organization. In summary, they can be divided into the following categories:

- **Product factors** are things like the complexity and required reliability of the product.
- **Platform factors** include e.g. available hardware resources such as memory and processor capacity.
- **Personnel factors** capture the capability and experience of the staff.
- **Project factors** are decided by e.g. the use of tools and the tightness of the schedule.

For a particular project, the exponent \( B \) as well as the product and personnel factors are usually difficult to influence. The project factors are typically the same for different implementation alternatives, which means that only the platform factors need to be considered during trade studies. The COCOMO model can thus, for trade studies, be summarized as:

\[
\text{swDevEffort} = A' \cdot \text{storage} \cdot \text{time} \cdot \text{swSize}^B
\]

where:

- **storage** is a factor depending on the amount of available storage (i.e. the values of the spareCapacity attribute of the memory components; see Section 4.3.7) that is actually used. Its value is 1 above 50% spare capacity, and increases to reach 1.46 as the free space drops to 5%.
- **time** is the corresponding factor for processor execution time (i.e. the values of the spareCapacity attribute of
the processor components; see Section 4.3.10). Its value is 1 above 50% spare capacity, and increases to reach 1.63 as the overhead drops to 5%.

(The numerical values are based on those given in [4].)

5.2. Hardware development

For the development of hardware, there does not appear to be any established effort estimation model. On the other hand, the variations in time is usually smaller than for software, and does for instance not depend very much on the choice of processor or memory. Just as for the COCOMO model, one would need to include factors capturing the experience of the designers, use of tools, etc., but no such model seems to exist, and it is thus up to each organization to create their own models based on previous projects.

In [1], some factors are discussed that contribute to the quality of the hardware design, but the paper is mainly relevant for the design of custom integrated circuits. A more holistic approach is taken in [6], where the relation between hardware and software and its implication on cost is studied. However, the paper assumes both a limited application area and serious limitations on the design of the hardware (using only standard boards), and does also not cover the hardware development cost.

6. Application

In the previous sections, we have presented the complete cost model. It has been structured to be as modular and flexible as possible, to make it widely usable. In this section, we discuss how the model is applied in the context of a trade study.

6.1. Performing trade studies

The purpose of the trade study is to compare the characteristics of different alternative solutions. Although this paper focuses on cost, all important characteristics have to be treated together in the trade study, to find the best compromise.

The designer first needs to make a description of the design, and in our model, this means creating instances of the objects in the SBS. Each time an instance is created, a new copy is obtained of all the attributes in the class, and this gives a set of variables describing the characteristics of the system. Also, a copy of the invariant equations is obtained, giving a system of equations relating the variables to each other. There are usually more variables than equations, so the designer must give input values on sufficiently many of them to make the system solvable. These values have to be obtained using the engineer’s expert knowledge, or be based on previous, similar products.

After the equations are solved, all the variables have values, and the engineer must again use his expert knowledge to determine the validity of the result. Sometimes, the system being developed has parts that break some of the assumptions in the model, meaning that the model must be adjusted, and sometimes the engineer might wish to adjust the margins to reduce or increase the risk in the project.

6.2. Discriminating factors

It is important to get a good picture of what the discriminating factors are between the alternatives, and concentrate on those issues during the trade study. Factors that discriminate are typically those where the specification gives the designer a considerable freedom. As an example, when the task is to find a suitable system architecture (i.e. what ECUs to have and how to connect them), design choices related to the interfaces of the system are mostly not discriminating. The choice of which sensor or actuator to use is not highly dependent on which node it is connected to. Further, the choice of input or output circuits needed in the ECU to interface the processor to the sensor or actuator mostly depends on what sensor or actuator has been chosen, and not in which ECU the circuit is placed. Thus the total contribution to the system cost from these choices remains the same, and they are thus not discriminating. If the trade study is performed manually, the effort can be substantially reduced by disregarding non-discriminating issues.

6.3. Tool support

Although the calculations in the models are not overly complex, it is desirable to have tool support that performs them automatically as far as possible. Preferably, the tools should use the information in an object-oriented design database, since this allows the designer to play “what-if” games with the design and immediately see the effect on the system cost.

It might also be helpful to have tool support for calculating some of the input data. In particular, the FP calculation for the operations of the software modules may require a substantial amount of work. It is probably worth the effort, since it can be used not only to estimate cost but also to derive realistic project plans. Still, the effort could be reduced if the functional requirements are stated in a formal or semi-formal language, which describes the system’s interactions in a precise way that is amenable to mechanical analysis.

The design solutions in the SBS must be traceable to the requirements in the specification. In practice, requirements always change during development, and by ensuring trac-
It becomes possible to quickly assess the economical implications. According to [8], changes in requirements is the most important factor contributing to cost estimation inaccuracy. Attaching a cost to all requirement changes becomes a restraining factor which will prevent modifications that are not absolutely necessary, thereby creating a more stable project.

7. Conclusions and future work

In this paper, we have presented a model for making early estimations of the cost of distributed electronic systems. The goal was to provide usable, flexible, accurate, and suitable models, which lead to the adoption of an OO approach. The models can be easily integrated into standard development tools, or used stand-alone, to provide a rapid calculation of the cost of a proposed design. This allows the engineers to objectively and systematically compare the cost of design alternatives, and optimize the solutions. This is in sharp contrast to the current practice, where estimations are either not made at all, or are performed in a very haphazard manner, with each individual designer reaching different results. We can thus expect that the application of models of this kind would quickly increase the control over the cost and also reduce the total project cost drastically, due to improved designs.

The model described in this work is a pioneer attempt to capture the overall cost of embedded electronic systems, and the literature on the subject is very scarce. Therefore, it should be seen as a first sketch of a model, rather than a final solution, but it serves as a good starting point for iterative refinement. By applying the model to real projects, calibration data can be obtained and it can be determined if the set of cost drivers should be modified.

Although this paper focuses on cost, all important factors must be considered simultaneously in the trade studies, since the goal is to find the best compromise between several, often conflicting, goals. Similar models need thus be developed for other factors, e.g. weight and reliability, and this could be done by simply adding more attributes and invariants to the equation system. In fact, the model presented is a general framework for system analysis, and could easily be transformed to other areas than electronics.

An interesting development would be to include uncertainties in the model, e.g. by giving all data as probability distributions. This makes it easier to assess the risk by calculating (e.g. through Monte Carlo simulations) the cost budget needed to have a certain level of confidence that the project will be able to meet its objectives.

8. References


