Cost Models with Explicit Uncertainties for Electronic Architecture Trade-off and Risk Analysis

Jakob Axelsson[1, 2]

[1]Volvo Car Corporation
Dept. 94100 PV32
SE-405 31 Göteborg, Sweden
jaxelss5@volvocars.com

[2]Mälardalen University
Dept. Computer Science and Electronics
SE-721 23 Västerås, Sweden
jakob.axelsson@mdh.se

Abstract. This paper discusses how the early phases of developing embedded electronic systems can be improved by enhanced modelling of cost and performance that includes explicit handling of uncertainties. The approach is to add cost information to existing UML models, capture uncertainties using probability distributions, and use Monte Carlo simulation to analyze the risk of not reaching the cost targets. It is demonstrated how the information obtained can be used when evaluating different architecture alternatives, while including both development and product cost as well as risk in the trade-off.

Introduction

When a new technical system is developed, the early design phases are always critical. In those phases, the basic concepts are selected, and they effectively limit what can be achieved in the subsequent stages in terms of, e.g., cost and performance. But the early phases are characterized by a lack of information, and of ad hoc decisions based on the intuition and experience of the engineers. It seems natural to try to improve the early phases by providing methods that give better indications of the important attributes of the system, and that quantify the risks. Far-reaching redesigns late in the project can then be avoided, thus reducing the cost, duration, and risk of development. In this paper, we describe methods to tackle this issue for a particular kind of product, namely embedded electronic systems and software and a certain attribute, namely cost. However, the approach should be useful also for other kinds of systems and attributes.

Overview of the problem

The early design phases are characterized by important decisions being made by a small group of engineers on the basis of uncertain information. The quality of those decisions could be improved by reducing the uncertainty of the information. However, that would require more people involved, which in many organizations is not possible. What can be done though is to quantify the uncertainty and use that to estimate the risk of different decision alternatives.

Due to the limited resources available, it is important to make the process as efficient as possible and make maximum use of the information available. In many cases, analysis models are being built as support for the decision making. If model libraries are available, that process can be made more efficient. Also, there is a subset of information that can be used for many types of analysis. For instance, the structure of the system is a basis for many models. If the common subset can be described once and for all, development of analysis models will become more efficient, and the risk of inconsistencies between different analyses is reduced if they share a common base.

Our solution

The approach of this work is to make use of a common model for many purposes. Object-oriented techniques are useful for structuring complex information and showing different aspects of the system of interest. We have therefore based the models on the Unified Modeling Language, UML (Rumbaugh 1999). In many organizations, UML is already used for system modeling, in particular for software, and with focus on functionality. Therefore, a formal description of certain aspects of the system is already available in a database, and we add the information needed for cost estimations in the early phases directly into that model, thus minimizing the time spent on modeling. The cost models can also be structured into libraries, allowing easy reuse and yet faster development. The same approach can be used to study other quantitative attributes than cost.

With the information present in the UML model of the system, we can extract the information necessary for cost estimations. This gives us a mathematical model with input and output parameters, where input parameters are associated with attributes at lower level of design (parts, components) and output parameters are key measures of performance at the system level. By describing the input parameters as probability distributions rather than point values, the uncertainty of parameter estimations can be captured. Using a Monte Carlo simulation, the resulting uncertainties in the output parameters can be derived. By comparing those probability distributions with the desired values (as described e.g. in the requirements specification) the risk of not meeting the targets can be explicitly determined and considered in the decision making.

Overview of the paper

In this paper, we extend and integrate ideas from two previous publications. In (Axelsson 2000), a UML based cost model for embedded electronic systems was presented. In (Axelsson 2005), the idea of using explicit modeling of uncertainties was investigated for a different analysis, namely that of performance of real-time systems, and Monte Carlo simulation was used for the evaluation. It turns out that the evaluation sometimes yields counter-intuitive results, which proves that it is valuable.

The contribution of this paper is to extend the previous cost model with the possibility to use uncertain parameters rather than point values, thereby making the framework for early architecture trade studies more complete and allowing a detailed risk analysis to be performed.

The remainder of the paper is structured as follows. In the next section, issues relating to estimations in early phases are discussed in more detail. Then, a product cost estimation model in UML is presented followed by a model for estimating development cost. To make the approach more concrete, an example is then discussed in some detail. Finally, the conclusions are summarized and indications are given in what direction this research could continue.

Estimations in early phases

In this section, we discuss in more detail some of the characteristics of the early conceptual phases of embedded electronic system design.

Early design decisions for embedded systems

An embedded system is by definition an integral part of a larger product, such as an automobile, an aircraft, or an industrial robot. The development of the embedded system is therefore intimately linked to that of the total product. In the early phases, it is usually necessary to make overall decisions about the parts, such as what electronic control units (ECUs) should be present in the network, what processors to use and what memory capacity is needed. These decisions can in practice hardly be delayed, since they are important prerequisites for the subsequent design steps, including software development. They also have important consequences on the overall product. For instance, the number of ECUs and their physical sizes must be known to ensure that they can be packaged properly in the product. The physical placement of the ECUs also provides environmental constraints such as temperature, vibrations, and EMC, which is necessary input to the detailed ECU design. Finally, the selection of hardware components is necessary to enable investments in tooling for mass production, which has a long lead time in many projects.

Current practice

In the early phases, there is typically a rough description of the functionality of the embedded system, but the details are not known. In some cases, it could be just a list of function names, and in other cases a first simulation model could be present. Often, UML is used to capture the models. It is common to work with rough sketches of the architecture, and based on that guess its attributes. The estimations are typically point values, that do not provide any indication of the uncertainty embedded in the estimation. This makes it difficult to reason about the risk associated with a particular design alternative, and to systematically compare different alternatives.

Often existing systems are being enhanced or a new product is developed based on some components from an old one. This means that there are large variations in the knowledge about different parts of the new system. For components that already exist, many attributes are well known, and can be measured, and the software code could even exist. For other parts of the same system, knowledge is as scarce as for a completely new design. This means that a realistic estimation approach for early phases must be able to handle such discrepancies in the available knowledge.

Estimations

The basic methodology followed when doing estimations of key attributes in the early phases is:

1. Identify the critical attributes needed for the design decisions.
2. Create a model that links those attributes to other attributes at a lower level of design, for which values can be directly estimated more easily.
3. Estimate the values for the lower level attributes including uncertainty.

Once the lower level attributes have been identified, the designers can estimate them using all their available knowledge and tools. By describing the attributes as probability distributions, they can also capture how large the uncertainty is. In many cases, this is in practice most easily done by letting the experts assess some key points, such as the 50th percentile (most likely value) and the 90th percentile (a value that the person believes the attribute will by 90% certainty not exceed). Then the actual probability distribution parameters can be determined by fitting the curve to these values.

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. 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 modules.

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.

One approach is to simply let an expert make intelligent guesses about the software characteristics directly. As an alternative, a more structured approach can be employed if the functions have been described in some detail, e.g. as UML models. A common technique to measure the complexity is through so-called function points (FP). 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. The following characteristics of the software modules can then be derived.

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 transformed into the number of lines of source code in the language used and the size of the compiled code in the program memory.

Data size. The size of the data can to a large extent be gathered from the specification, since it is mostly mentioned directly or indirectly in the functions.

Execution effort. The execution effort of a segment of software depends on the number of instructions involved. This is proportional to the size of the software that make up that segment, which is given by the FPs as described above. 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.

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.

Weibull distribution

In some cases it is reasonable to assume that estimations follow the normal distribution, where the mean value has the highest probability and the probability of other values decrease the further away they are from the mean. This works well if the range of the estimated parameter is the whole range of real numbers, but for practical applications the range is often limited in one end, as the parameter is always positive. Examples of this is time, cost, weight, temperature (in Kelvin), and physical size. For such parameters, a normal distribution would always assign a certain (albeit arbitrarily low) probability to impossible values.

An alternative is to use a distribution which gives zero probability to all negative values. In (Kujawski 2004), a number of such distributions are discussed for the purpose of cost estimation, and the authors conclude that the Weibull distribution is suitable for estimations. It has the following cumulative distribution function:

$$F(x;\alpha,\beta) = 1 - e^{-(x/\beta)^{\alpha}}$$

The parameter α controls the shape and β controls the spread. The distribution has desirable properties such as a peak around the mean (if an appropriate α is used), no probability of values below zero, and a tail towards positive infinity. In addition to the above, it is possible to add a location parameter, so that the whole curve can be shifted along the x-axis.

Monte Carlo-based evaluation

In this section, a brief overview of Monte Carlo simulations is given. The methodology is actually very simple, and is based on the steps described in the section on estimations above. Based on the evaluation model, a number of iterations is run, each consisting of the following steps:

1. Generate a random value for each of the low level attributes, according to the probability distribution provided by the estimator.
2. Put those values into the model that links the low level attributes to the high level ones.
3. Collect the outputs from the model in a histogram for each high level attribute.

The histograms will after the completed simulation show the probability distributions for the high level attributes, and can be used to determine key information such as mean values and the confidence that the critical parameters will have acceptable values.

The following important benefits should be noted:

- It is a very general methodology that does not assume anything about the model. For embedded systems, any assumptions about hardware, software, etc. is feasible.
- An existing model for linking high level and low level attributes can be used as is.
- It does not require any analytical application of probability theory, which can be very cumbersome or even impossible for complex systems.
- It can be used with varying levels of knowledge. Some parameters can have fixed values and others can be probability distributions of any kind, which is important when refining existing products, as described in the section on current practices above.

The main possible drawback is that if there are many parameters, a large number of iterations could be necessary. If the model is complicated, and requires a long computation time, this might be a slow process. On the other hand, each iteration is totally independent, and thus could be run in parallel. For cost models fairly little computation is required, so performance is not an issue.

Using UML to build a multi-purpose model

A benefit of our approach is that we use the UML standard modeling notation to capture the information needed for cost estimations. In that way, an already existing system model can be extended with small means. However, UML is mainly intended for describing (software) functionality, so therefore a short description of how it can be used to capture mathematical models of quantitative features is needed.

The basic idea is that the model is captured in a number of UML classes. The mathematical variables are attributes of those classes, and the mathematical equations of the model are invariants of the UML classes, that describe how the attributes of the class and attributes of related classes are connected to each other.

To use the model, the classes are instantiated for a particular system. Then, the output parameters of the model are identified, and all attributes on which they depend are extracted, together will all invariants that related them. In this way a number of mathematical input and output variables are obtained. The input variables are given values as part of the model instantiation, and the equation system is solved to obtain the output parameters.

The approach might seem unnecessarily elaborate, since the same mathematical model could be built from scratch without involving UML at all. However, much of the information needed, such as the structural relations between classes, would be present anyway, thus the modeling effort is reduced. Also, the system models are refined iteratively in most projects, and the cost models then also need to be updated. Those repetitive updates become much easier to handle, and inconsistencies are avoided, if the same modeling base is used for all purposes.

A product cost estimation model

In this section, we present a cost estimation model in UML for electronic architectures. It is based on (Axelsson 2000), but somewhat simplified, since the main topic of this paper is to show how the cost analysis can be enhanced by including uncertainties.

The model is structured as a set of classes, which is shown in Figure 1, using UML notation. The basic structure is a normal system breakdown structure, with the product consisting of a number of parts, which in turn contain a number of components. Part and component are abstract classes that are specialized in this model with a number of other classes that are specific to the application domain of embedded electronic systems.

The attributes of the classes are summarized in Table 1. For completeness, all attributes of each class are explicitly indicated, even if they are inherited (in which case their name is in italics). It is also shown how the attributes are used in the cost model, where "in" means that they should be given a value by the engineer, "out" means that the value is derived by the model, and "const" that it is a scale factor that is the same for all instances of the class. It should be noted that although the term constant is used for the scale factors, this does not necessarily mean that the values are known exactly in the early phases, but an estimation with uncertainty applies also to these attributes.

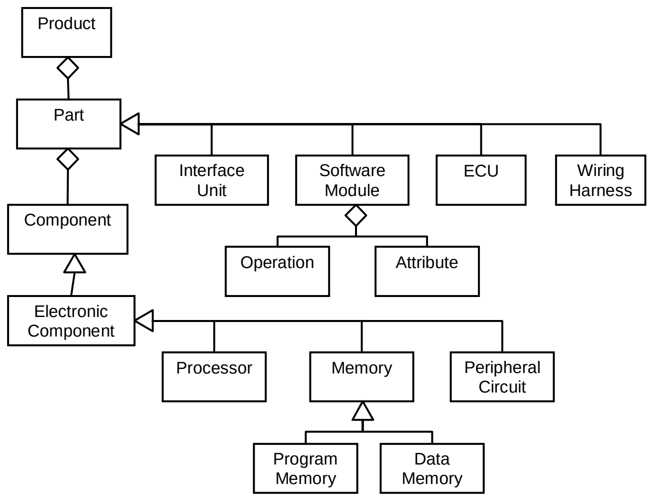

Figure 1. Class diagram for cost model

Class	Attributes			
	Use	**Name**	**Unit**	**Description**
Product	in	numberOfUnits	-	Expected number of units to be manufactured
	out	totalCost	$	Total cost of each copy of the system up to delivery
	out	productCost	$	Cost of producing one more copy of the system
	out	developmentCost	$	Cost of developing the system
	out	swDevEffort	MM	Effort in man months to develop the software
	out	hwDevEffort	MM	Effort in man months to develop the hardware
	const	manCost	$/MM	Cost of one man month
Part	in/out	productCost	$	Cost of producing one more copy of the part
Interface unit	in	*productCost*	$	Inherited from Part
Software module	out	*productCost*	$	Inherited from Part
Attribute	in	size	bytes	Size of the data held in the attribute
Operation	in	size	FP	Size of the software implementing the operation
	in	effort	FP	Maximum number of FPs performed when this operation is invoked, i.e. length of the longest path through the code of the operation
	in	invocationRate	-/s	How often the operation is invoked, as a maximum
Wiring harness	in	numberOf Conductors	-	Number of conductors in the wiring harness
	out	*productCost*	$	Inherited from Part
	const	conductorCost	$	Cost of adding one more conductor to the harness
ECU	out	*productCost*	$	Inherited from Part
	const	sizeCost	$\$/m^2$	Factor indicating how the total cost of the ECU increases with its content, measured as size of the circuits (captures cost of housing, PCB, etc.)
Component	in/out	productCost	$	Cost of producing one more copy of the component
Electronic component	in/out	*productCost*	$	Inherited from Component
	in	size	m^2	Size (footprint) of the electronic circuit
Memory	in	*size*	m^2	Inherited from Electronic Component
	in	capacity	bytes	Number of bytes of information the memory can hold
	out	*productCost*	$	Inherited from Electronic Component
	out	spareCapacity	%	Percentage of capacity unused
	const	byteCost	$/byte	Cost of one byte of memory
Program memory	in	*size*	m^2	Inherited from Electronic Component
	in	*capacity*	bytes	Inherited from Memory
	out	*productCost*	$	Inherited from Electronic Component
	out	*spareCapacity*	%	Inherited from Memory
	const	*byteCost*	$/byte	Inherited from Memory
Data memory	in	*size*	m^2	Inherited from Electronic Component
	in	*capacity*	bytes	Inherited from Memory
	out	*productCost*	$	Inherited from Electronic Component
	out	*spareCapacity*	%	Inherited from Memory
	const	*byteCost*	$/byte	Inherited from Memory
Peripheral circuit	in	*size*	m^2	Inherited from Electronic Component
	out	*productCost*	$	Inherited from Electronic Component
Processor	in	*size*	m^2	Inherited from Electronic Component
	in	capacity	FP/s	Number of FPs the processor can execute per second
	out	*productCost*	$	Inherited from Electronic Component
	out	spareCapacity	%	Percentage of time the processor is idle
	const	speedCost	$/(FP/s)	Cost of a processor that can execute one FP per second

Table 1. Attributes of classes in the cost model

Equations

The equations used to derive the cost estimations (the "out" attributes) are expressed as invariants for the classes, and these equations relate the attributes of different elements. In the remainder of this section, the invariants are listed for each class.

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

$$totalCost = productCost + \frac{devCost}{numberOfUnits}$$

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

$$productCost = \sum_{p \in part} p.productCost$$

The development cost is calculated as follows:

$$devCost = (swDevEffort + hwDevEffort) \cdot manCost$$

The derivation of development effort for hardware and software is discussed further in the next section.

Part. The product cost of a part is the sum of the product cost of all its components:

$$productCost = \sum_{c \in component} c.productCost$$

Software module. There is no product cost associated with software modules, $productCost = 0$.

Wiring harness. The product cost of the wiring harness equals the number of wires times the cost of one wire (including connectors).

$$productCost = numberOfConductors \cdot conductorCost$$

It is assumed that the wires are reasonably short, i.e. in the order of a few meters. For larger distances, the length should be included as well.

ECU. The ECU product cost is the sum of the cost of the components, plus a factor covering housing, PCB, etc, whose cost increases with the size of the package, which in turn depends on the sizes of the content, i.e., of the electronic components inside:

$$productCost = \sum_{c \in component} c.productCost + sizeCost \sum_{c \in component} c.size$$

Memory. The cost and size are functions of the capacity of the memory:

$$productCost = capacity \cdot byteCost$$

Program memory. The spare capacity of the program memory is calculated from the sum of the program size of all software components' operations (*ops*) that are part of the ECU:

$$spareCapacity = \left(1 - \frac{fpSize \cdot \sum_{o \in ops} o.size}{capacity}\right) \cdot 100$$

Data memory. The spare capacity of the data memory is calculated from the sum of the data size of all software components' attributes (*atts*) that are part of the ECU:

$$spareCapacity = \left(1 - \frac{\sum_{a \in atts} a.size}{capacity}\right) \cdot 100$$

Processor. The spare capacity of the processor is calculated from the operations performed by it and their invocation rates:

$$spareCapacity = \left(1 - \frac{\sum_{o \in ops} o.effort \cdot o.invocationRate}{capacity}\right) \cdot 100$$

The cost is modeled as a linear function of the capacity, $productCost = capacity \cdot speedCost$.

Models for 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.

Software development

A widely accepted model for software development cost estimation is Barry Boehm's Constructive Cost Model (COCOMO), which is described in (Boehm 1995). 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 man months (MM) needed to conclude a project:

$$swDevEffort = A \cdot swSize^{B}$$

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

$$swSize = \frac{fpSourceCode}{1000} \cdot \sum_{o \in ops} o.size$$

where *fpSourceCode* is the number of lines of source code of the selected programming language that result from one FP, and *ops* are the total set of operations of all software modules. 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:

$$swDevEffort = A' \cdot storage \cdot time \cdot swSize^{B}$$

where:

- *storage* is a factor depending on the amount of available storage (i.e. the values of the attribute *spareCapacity* 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 attribute *spareCapacity* 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%.

A' captures the remaining factors in *A*. The numerical values for *storage* and *time* are based on those given by (Chulani 1999).

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 (Aas 1995), 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 (Debardelaben 1997), 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.

Example

In this section, we present a small example to illustrate how the analysis can be used, and what conclusions can be drawn. In the first subsection, the example is introduced by means of a functional view. Then, different alternative solutions regarding hardware and the allocation of software to ECUs are presented. Finally, an evaluation using the Monte Carlo simulation is performed, and the results are discussed.

The product and its functionality

The system in the example is a simple embedded control system whose functional structure is illustrated in Figure 2. There are three sensors and two actuators that constitute interfaces to the rest of the product. Sensor 1 is monitored by a separate software module, and another software module monitors sensor 2 and 3 together. The system performs two main functions each of which is implemented in a separate software module. Finally, there is one module for each of the two actuators. It is assumed that the sensors and actuators are distributed in the physical product, but with Sensor 2 and 3 located close to each other.

The marketing department estimates the total production volume of the system to 100,000, and the budget for the project is \$6 million, with a cost for one man month of \$16,000. For the COCOMO model, the factor A' is known to be 2, and the exponent B is estimated using a Weibull distribution, with the 50th percentile at 1.3 and the 90th at 1.5.

The company has previously developed a similar product, and the staff is fairly familiar with the algorithms etc. through the flow from sensor 1 through the software for function 1 to actuator 1. These modules can therefore be estimated with higher accuracy than the other modules.

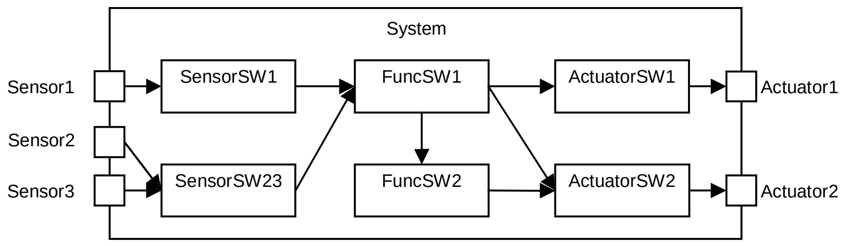

Figure 2. Functional view of the system

Alternative architectures

We will now exploit three alternative system architectures, illustrated in Figure 3. The first architecture is centralized, with only one ECU handling all the software tasks and all sensors and actuators. Since the input and output units are spread out physically, there will be relatively long wires between the ECU and those units.

Architecture 2 is a distributed solution with two ECUs. Each ECU handles one function, and also the sensors and actuators connected to it. Since the functions are interrelated, some additional wiring is necessary for communication between the ECUs.

In the last architecture alternative, a separate ECU is provided to handle one of the actuators. The point-to-point communication between the ECUs in the previous architecture is replaced by a data bus to reduce wiring.

In all cases, each ECU consists of a processor, a program memory (FLASH), a data memory (RAM), and one peripheral circuit for each connection. There is a fixed set of processors and memories to choose from, with different capacity and prices.

In the programming language used, it is estimated that the number of lines of code per FP is a Weibull distribution with median 50 and 90th percentile 90. The estimated distributions for software programming productivity (lines of code per hour) is 10 (median) and 30 (90th percentile). The size of the compiled code is 20 bytes per line of source (median) and 30 (90th).

Similar estimations are given for all other necessary input parameters, although space does not permit us to give the exact numbers. In general, it is assumed in this example that Weibull distributions are used, and that the 90th percentile is 1.5 times the median. However, for SensorSW1, FuncSW1, and ActuatorSW1, the previous knowledge about the algoritms reduces the uncertainty, so that the 90th percentile is 1.2 times the median. For all Weibull estimates, $\beta = 2$ is used.

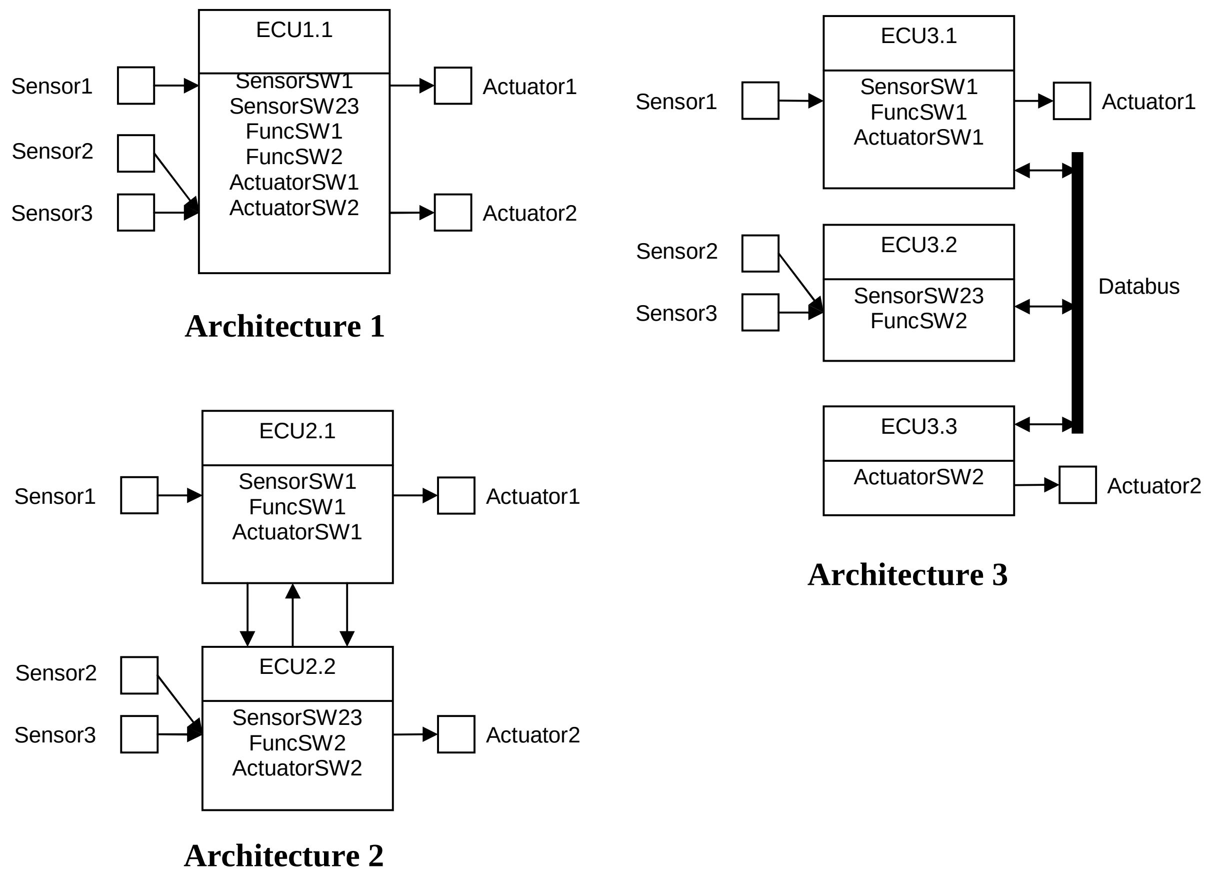

Figure 3. Three alternative architectures

Evaluation

Based on the data described above, a Monte Carlo simulation was run and output parameter estimates obtained. Some of the key attributes are listed in Table 2, with their mean values. Figure 4 shows the probability distributions of the total cost for the three architectures. An analysis of the data shows that the 90th percentile of the total cost of the architecture alternatives is \$350, \$285, and \$265, respectively. As can be seen from the table, Architecture 3 is the cheapest alternative in terms of product cost. However, it exceeds the project budget slightly, so if that is a critical factor, the alternative to choose would be Architecture 2.

Attributes	**Unit**	**Architecture 1**	**Architecture2**	**Architecture3**
Total cost	$	258	199	186
Product cost	$	169	135	118
Development cost	M$	8.1	5.9	6.3
- SW	MM	485	330	330
- HW	MM	20	41	61
ECU 1 spare capacity				
- Processing	%	90%	97%	97%
- Program memory	%	69%	92%	92%
- Data memory	%	26%	48%	48%
ECU 2 spare capacity				
- Processing	%		72%	73%
- Program memory	%		77%	79%
- Data memory	%		55%	57%
ECU 3 spare capacity				
- Processing	%			100%
- Program memory	%			98%
- Data memory	%			98%

Table 2. Mean values of parameters obtained from Monte Carlo simulation.

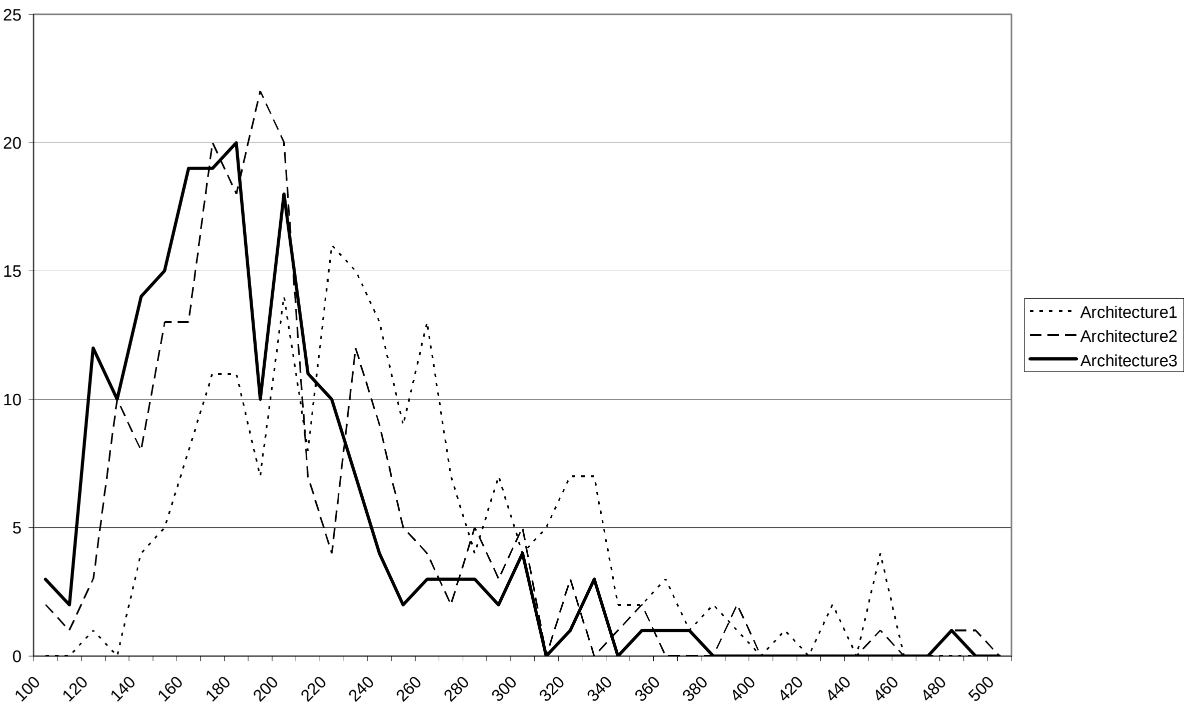

Figure 4. Probability distributions of the total cost for each architecture.

Discussion

Although the example presented is very simple, it illustrates some of the steps that need to be taken to use the analysis model, and what information can be drawn from it. One point to notice is that the model links product development cost and product cost in a rather subtle way. As can be seen in Table 2, the first architecture has substantially higher software development cost than the other two, although the functionality is the same. The reason for this is that COCOMO penalizes architectures with little spare capacity heavily (due to the added cost for optimizing the software). Indeed, it can be seen that ECU 1 has much less spare capacity in the first architecture than in the others. This gives the engineers a clue that by adding more memory to that ECU in the next design iteration could be worth investigating to improve that alternative.

Conclusions

In this paper, we have presented a model for making early estimations of cost of embedded electronic systems with explicit capturing of risks and uncertainties. The model is suitable for trade studies in the early development phases. The approach is to extend UML models of the system to include cost information in order to minimize the modeling effort and risk of inconsistencies. The cost model can then be extracted from the UML database. To capture uncertainties, input parameters are given as probability distribution, where in many cases the Weibull distribution is suitable. The output parameters are obtained as probability distributions through a Monte Carlo simulation, and the resulting distributions can be compared to requirements to see how large the risk of not fulfilling the demands is.

In our experience, efficiency of modeling is paramount in practice, since the models will otherwise not be used. Instead, engineers fall back on intuition in their decision making. We believe that our approach provide the needed efficiency. Combined with our previous model for performance analysis (Axelsson 2005), a cost-performance trade-off capturing risk becomes possible based to a large extent on UML models already present in the organisation.

Although the paper focuses on a particular system domain, we believe that the main contribution is the approach, which can be used for many types of systems. Also, other quantitative properties than cost can be studied in the same way, by adding attributes and equations to existing UML models and extracting a model to perform analysis. The cost model we presented is fairly simple, and in most cases a more advanced model would be needed, and domain specific considerations must be added. Using the modeling framework described in the paper, such modifications should be easy to accomplish in most cases.

So far, the modeling has been implemented in a minimum effort approach, requiring some manual work to extract the information. In the future, we plan to integrate it into commercial UML tools, thereby making it possible to conduct larger scale trials. Also, we plan to develop model libraries to make the approach even more useful in practice.

References

Aas, E. J. and Sundsbø, I. Harnessing the Human Factor for Quality Designs. *Circuits & Devices*, Vol. 11, No. 3, pp. 24-28, May 1995.

Axelsson, J. Cost Models for Electronic Architecture Trade Studies. In *Proc. 6th IEEE International Conference on Engineering of Complex Computer Systems*, pp. 229-239, Tokyo, September 2000.

Axelsson, J. A Method for Evaluating Uncertainties in the Early Development Phases of

Embedded Real-Time Systems. In *11th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, Hong Kong, August 2005.

Boehm, B *et al*. Cost Models for Future Software Life Cycle Processes: COCOMO 2.0. In *Annals of Software Engineering Special Volume on Software Process and Product Measurement.* J. C. Baltzer AG, Science Publishers, 1995.

Chulani, S. *et al*. Bayesian Analysis of Empirical Software Engineering Cost Models. *IEEE Transactions on Software Engineering*, vol. 25, no. 4, July 1999.

Debardelaben, J. A. *et al*. Incorporating Cost Modeling in Embedded-System Design. *IEEE Design & Test of Computers*, Vol. 14, No. 3, pp. 24-35, July 1997.

Kujawski, E. *et al*. Incorporating Psychological Influences in Probabilistic Cost Analysis. *Systems Engineering*, Vol. 7, No. 3, pp. 195-216, 2004.

Rumbaugh, J. *et al*. *The Unified Modeling Language Reference Manual*. Addison Wesley, 1999.

Biography

Jakob Axelsson received an M.Sc. from Linköping University in 1993, and a Ph.D. in 1997 for a thesis on hardware/software codesign of real-time systems. He has been working at ABB Corporate Research and ABB Power Generation (now Alstom) in Baden, Switzerland, Volvo Technological Development (now Volvo Technology) and Carlstedt Research & Technology in Göteborg, Sweden. He is currently with the Volvo Car Corporation in Göteborg, where he is program manager for research and advanced engineering for electrical and electronic systems. He is also an adjunct professor in software and systems engineering at Mälardalen University in Västerås, Sweden. Dr. Axelsson was a member of the board of the Swedish INCOSE Chapter 2002-05, serving as chapter president in 2003.