The Cost of Communication Protocols and Coordination Languages in Embedded Systems

K.G.W. Goossens and O.P. Gangwal

Philips Research Laboratories, Eindhoven, The Netherlands
{Kees.Goossens,O.P.Gangwal}@philips.com

Abstract. We investigate the use of communication protocols and coordination languages (henceforth interaction languages) for high-volume consumer electronics products in the multimedia application domain. Interaction languages are used to reduce the cost of designing a product by introducing structure and abstraction, by being application-domain specific, and by enabling re-use. They can also reduce the manufacturing cost which includes the cost of using the interaction language, the implementation cost of the interaction language, and its running cost.
We classify services that can be offered by an interaction language and their impact on the cost of designing. Choices that can be made in their implementations are also categorised, and their impact on the manufacturing cost is shown. This is illustrated by three existing interaction languages: C-HEAP, Arachne, and STM.
We conclude first that the type of services offered by an interaction language must match the application domain. Furthermore, implementation choices are constrained by the underlying system architecture as well as the services to be offered. Finally, an interaction language with fewer services minimises the manufacturing cost but increases the cost of designing, and vice versa. The cost of designing and the manufacturing cost both contribute to the cost of the final product, which is to be minimised. These costs must be balanced when designing an interaction language.

1 Introduction

In this paper we consider the role of communication protocols and coordination languages[1] in high-volume consumer products in the multimedia application domain. There are two salient points. The multimedia application domain deals with complex systems, which can be easier to design using appropriate interaction language [24]. Moreover, high-volume production of consumer products requires a focus on system cost, which interaction languages can help to reduce. We elaborate both in turn below.

First, the multimedia application domain has its own characteristics. Multimedia processing entails huge amounts of computation and communication due

[1] To avoid repetition of the long phrase "communication protocol and coordination language" we write interaction language. We leave open whether the communication protocol and the coordination language are strictly separated or not [3].

to the high data rates (throughput) of digital video, and the large number of operations per video sample. Furthermore, the processing must obey real-time constraints because data rates are externally imposed. Applications are increasingly complex and dynamic in their behaviour, and the design and implementation of multimedia systems is therefore challenging [28]. The use of interaction languages structures the design process (how do we come to a design) and the design itself (the software and hardware components and their interaction).

Second, our applications are implemented as embedded systems (or devices) for the high-volume consumer electronics market. This means that the cost of the device is very important. The cost of a consumer device can be split into two parts: the cost of an individual device (*the manufacturing cost*, including material costs and manufacturing tests), and the so-called non-recurring engineering costs (*the cost of designing* the system). Table 1 shows these costs and their subdivisions, which we explain below.

cost of	**with contributions from**	**amortised over**
designing	abstraction, structuring, decomposition	design method
	tailoring to application domain	application domain
	re-use	product families & design method
manufacturing	cost of use of interaction language	design
	interaction language implementation cost	design
	running cost	design

Table 1. Factors contributing to the device cost.

To reduce the cost of designing, the design process must be better structured [13], using abstraction and compositionality. The design of individual products can be eased through the use of interaction languages appropriate to that application domain [24]. At a higher level, product families (multiple products with common features) can benefit from re-use of existing systems to create derivatives, and standardisation of the design process, through promotion of platforms, through the use of interaction languages, and so on. It may be argued that in high-volume product runs non-recurring engineering costs are not important because they are amortised over many devices. However, improving the design process reduces the so-called time to market, which can have a large impact on the number of products sold. The amortisation column of Table 1 indicates the largest scope in which the contributions mentioned in the second column are effective. For example, the use of abstraction, structuring, and decomposition in a design method benefits all designs made with that method. Hence the cost of developing an abstract, structured, compositional interaction language is amortised over all these designs.

We relate the cost of a single device to the manufacturing cost, which is to be minimised by using an appropriate interaction language. This cost is composed of the cost of the implementation using the interaction language (the *cost of* *use of interaction language*), the cost of the implementation of the interaction language itself (the *interaction language implementation cost*), and the cost of using the interaction language in an implementation (the *running cost*). All these costs are paid per design.[2]

In the remainder of this paper we use the terminology of the ISO OSI reference model [8], where an interaction language offers *services* to its users, implemented by a peer-to-peer *protocol* that uses the services of lower layers. Further terminology: *designing* (or: to design) leads to a single *implementation* or *design* that when *manufactured* leads to a *device* or *product*. A design is therefore the blueprint for the tangible device. A *product family* contains designs that have common characteristics, and are perhaps *derived* from each other, through a re-use strategy or otherwise [29].

In Section 2 we classify the services provided by interaction languages and how these affect the cost of designing.

In Section 3 we observe that the same service can be implemented in different ways and we classify implementation choices and their impact on manufacturing costs.

We show how different interaction language choices, in terms of services and implementations, lead to different costs by means of three running examples. All three interaction languages address the limited amount of memory that is available to applications in (embedded) systems in the multimedia domain. Still their solutions, i.e. services and costs, are different: C-HEAP [15] implements local memory management at a low implementation cost using an explicit claim/release mechanism, Arachne [16] implements global memory management across communication channels at a higher implementation cost using an explicit claim/release mechanism, and space-time memory (STM) [25] implements global memory management with a shared data space and garbage collection.

In Section 4 we conclude that varying the balance between the cost of designing and the manufacturing cost results in different ways of designing, different interaction languages, and different designs. They can be judged only by the cost of the final products.

2 Services Offered by Interaction Languages

In this section we look at the influence of the application domain on the services that an interaction language must offer. In Section 2.1, these services are divided in coordination or configuration services, communication services, and inspection services. Some general properties of services are then defined and demonstrated in the running examples in Section 2.2 and 2.3 respectively. Finally, in Section 2.4 we examine the cost (or better: advantage) of using interaction languages in the design of embedded systems.

[2] The cost of getting to an implementation of the interaction language is amortised over the design method, which is included in the re-use part of the cost of designing.

2.1 Classification of services

The services that an interaction language offers must match the applications for which it is used. For example, automotive applications are control dominated and must be robust (cannot lose events). In video surveillance applications data collection and compression may be very lossy and computation and storage must be minimised, whereas in medical applications data integrity is essential and computation and storage costs are subservient. As stated in the introduction, we focus on multimedia applications that typically deal with high data rates (e.g. high-resolution digital video) and many computations per datum. The massive number of operations that must be performed per second can only be performed in parallel. Instruction-level parallelism (e.g. by using VLIW architectures), single-instruction multiple-data (SIMD), and task-level parallelism (TLP) can be used to obtain the desired performance. Multimedia applications often consist in streams of data processed by filters (scaling, decoding, encoding, transcoding, mixing, interleaving, etc.) that are composed according to the currently required functionality. This fits the TLP programming model of a number of concurrent tasks flexibly composed by an application manager depending on the user requirements [5]. Now, the resources that are available in the system are limited because we deal with embedded systems. The quality of service delivered to the user should be optimal using these limited resources. Tasks must be coordinated and parameterised dynamically because data rates vary in the system (e.g. variable bit rate MPEG), and also to obey the fickle user.

To address these different aspects we distinguish three kind of services. The composition and interaction of tasks is regulated by an application manager, which uses configuration or *coordination services*. The tasks operating on data use *communication services* to obtain access to input data and pass results to other tasks. Both tasks and the application manager can use *inspection services* to interrogate their environment, for example to check for the availability of data or the absence of deadlock in a task graph.

Configuration or coordination services

During execution, a system traverses a series of configurations (or steady states). A configuration is composed of different entities, such as tasks or components, channels or connections, ports, and tokens. Tokens are memory regions used for data storage and/or communication. Tokens are gathered in collections such as channels or pools. Components and channels can be connected in a topology using ports, and tokens are associated to channels or perhaps more generally to token pools. Systems can be generically described in many ways; we do not intend to do so here, but refer to e.g. [3, 2, 29, 27].

Configuration services enable the construction and modification of a configuration of the system. Entities must be created, destroyed, or modified (e.g. started, suspended, stopped, moved, flushed), and their interaction specified or modified (e.g. event triggers, changing the task graph). The computation and communication entities must be distributed over the limited resources of an embedded system. Entities are bound to (associated with) resources identified by their type, location, or name. For example, tasks are bound to processors, and tokens to memory regions. Similarly, mobile agents are bound more elaborately by a security profile that includes credits for computation and communication, an area to roam (LAN, WAN, virtual private network, anywhere) [6], and so on. The coordination of entities and resources, usually by an application manager, is essential for embedded systems that usually have hard real-time constraints, and always have limited resources.

Reconfiguration is the transition between two configurations. Defining the transition behaviour of entities is not easy [17]. We must define when, with whom, and how a reconfiguration takes place and what its effects on entities and their state are. *When* relates to time or location in the control or data flow; *whom* can vary from a single entity (local, autonomous reconfiguration) to everyone (global reconfiguration); *how* can mean a master-slave relationship between entities (e.g. a single quality-of-service manager plus rest of system) or cooperative via negotiation. The effect of reconfiguration can be visible (a disconnected or dangling channel) or not (suspended without warning or consent).

The difference between configuration and reconfiguration is not always sharply defined. For example, task scheduling on a programmable processor is seen as reconfiguration if it needs to be explicitly requested, otherwise it is part of the steady state. Pre-emptive scheduling of tasks is a service that is implicit, whereas explicit user-initiated suspension or thread switching could be classified as a reconfiguration service or a steady-state (communication) service. Another example is memory management. It can be implemented pervasively (like garbage collection in STM or Linda [22]), offered at the coordination level (global memory management in Arachne, local memory management in C-HEAP), or as explicitly managed shared memory [20] using communication services.

Steady-state or communication services

Interaction languages exist to allow entities to exchange data and synchronise in a given configuration. Data transfer services vary in the way data is accessed (message passing, shared data) and the structure of the transmission medium (does it lose or duplicate data, how is data identified and addressed, is there an underlying structuring of the data). Examples are shared memory with a load/store interface, Bonita [26] with values in a shared data space, JavaSpaces [14] with objects and associated methods or triggers in shared medium, point-to-point FIFO message-passing [19], broadcast events, and so forth.

As soon as multiple entities execute autonomously, their synchronisation becomes germane. Communication primitives often combine data transfer and synchronisation [10]. Their separation, however, allows the granularity of data communication (perhaps imposed by the communication medium) to be different from the granularity of synchronisation (which is natural to the component) [15, 16]. Synchronisation can be synchronous (performed at the same instant, synchronous hand-over, rendez-vous, two threads of control coincide) or asynchronous (at different points in time) [2, 11]. Asynchronous communication decouples the operation of components, and therefore requires storage of values or objects. The structure of the communication medium then becomes important. A component can fully commit to a synchronisation (blocking, i.e. for all time under all conditions) or commit conditionally. Conditions can include a deadline before which the synchronisation must take place, the amount or type of data that the other party must produce or consume [14], and so on. Multiple concurrent or dependent synchronisations can be useful, such as the two forks a hungry philosopher claims simultaneously (cf. the dependent transactions of [21]).

There may be a single coordinating task not computing on data (a boot task or quality-of-service manager) or there may be many tasks doing both (e.g. a collaborative agent-based interaction). However, the coordination and communication services are probably best separated so that a task can be designated either as a data-processing or coordinating task. Multiple coordinating tasks are best avoided as unforeseen or unwanted interactions easily arise.

Inspection services

In a given configuration a component may want to observe and analyse the state or performance of the system. This information can be used to manage resources and to regulate the quality of the service, by reconfiguring the system if needed. The inspection services must match the reconfiguration services; it is probably not useful to observe that which cannot be steered.

At the coordinating level, activity of components could be used to detect starvation, deadlock, etc. At the component level tasks may wish to observe the state of other components before they want to engage in communication or synchronisation, or the state of the communication medium to ensure the presence of sufficient data to commence processing.

2.2 General Properties

Interaction languages and coordination languages have some general properties that we mention briefly.

Global versus local

A global service, such as memory management or scheduling, takes into account all relevant information in the system. Globality is usually used to obtain optimal resource usage system wide. Global services are cost effective only when the use of resources varies over different parts of the system. A global software approach for buffer sharing over channels is presented in the context of embedded systems in [23].

Static versus dynamic

Decisions can be taken for the life time of the system (static), for the duration of a configuration (dynamic), or be continually under review (more dynamic). Examples of decisions are binding of memory to channels, and binding of processes to processors. Lowering the frequency of decision-taking will usually result in less overhead. Dynamic decision-taking is worthwhile for designs where the average and worst-case resource use differ much. An example of a memory manager for dynamic data structures (but not communication) for embedded systems is presented in [9].

Global dynamic services can move resources to parts of the system where they are needed most, which can result in optimal resource throughout all configurations (Table 2). On the other hand, local static services may not be optimal, but are simplest to implement. We return to their relative merits in Figure 2(C) of Section 3.4.

combination of service choices	consequences
global & dynamic	optimal throughout all configurations
global & static	optimal at start of each configuration
local & dynamic	not useful
local & static	not optimal, simple & fast

Table 2. Some examples of interactions of service choices.

2.3 Three Running Examples

We consider three interaction languages in the application domain of signal-processing. STM [25] is for use in the so-called Smart Kiosk, produced in low volumes with limited cost restriction, whereas Arachne [16], and C-HEAP [15] are meant for embedded systems in the high-volume consumer market with a cost focus. Although their basic services are more or less equal, they optimise different properties: the structure of the communication medium and the scope of the memory management.

STM allows reconfiguration of tasks and channels at any point by any task (creation and destruction, as well as changes in topology). The communication medium is based on message passing (synchronisation and data transfer are combined) to random-access channels. Specifying a channel at a virtual-time, data can be written only once, but read several times. Reading and writing take place at any time in a time window to enable pervasive memory management by garbage collection. There are no inspection services.

Arachne supports autonomous reconfiguration by any of its components at any time. Tasks are static, but channels can be created or emptied at any time by any task. Empty tokens can be added or removed from the system at any time. The communication medium is a shared memory organised in FIFO multicast, narrowcast, and merged channels for full tokens, and pools (sets) for empty tokens. Synchronisation for empty and full tokens is independent of data transfer to and from these tokens. Synchronous token hand-over is also supported. Token management is global. Inspection services include checking the amount of data (space) in a channel (pool), the activity on communication ports, and examining the interconnection of tasks.

C-HEAP supports reconfiguration initiated by a single master (the application manager) reacted to by multiple slaves (the tasks) at predetermined points in their control flows. Tasks and channels can be created, destroyed, (re)started, suspended, resumed, and stopped. The communication medium is a shared memory organised in FIFO channels for full and empty tokens. Synchronisation for empty and full tokens is independent of data transfer to and from these tokens. Token management is per channel, i.e. local. Inspection services are limited to the checking of the amount of full or empty tokens in a channel.

For more or less the same application domain, STM offers the highest level of services, in terms of reconfiguration and abstraction of data communication, followed by Arachne, and then C-HEAP.

2.4 Services and the Cost of Designing

Using appropriate interaction languages the *cost of designing* embedded systems can be lowered in three ways (recall Table 1).

First, large or complex systems like today's multimedia products benefit from a structured and compositional design method. Interaction languages can help by separating component design from communication design (computation versus communication and coordination [3]). This separation of concerns enables hierarchical and divide-and-conquer design methods. In general, more abstract interaction languages relieve designers of detailed working out of an implementation (e.g. a message-passing instead of shared-memory architecture), leading to a reduction of the design time (but perhaps at higher implementation costs, as we shall see later). Shortening the design time is an essential prerequisite for reducing the time to market.

Second, an interaction language tuned to an application domain enables a natural, and hence efficient, description of the design at hand. For example, it depends on the application domain whether sampling of incoming data may be lossy (the data rate is higher than the sampling rate, e.g. Splice [4]) or duplicative (the sampling rate is higher than the data rate), or a combination of both. Thus we see that C-HEAP and Arachne support only lossless FIFO channels, whereas STM offers random-access data channels for more sophisticated subsampling.

Third, interaction languages and (especially) coordination languages can facilitate re-use of (parts of) designs in different products [1, 29]. Design for re-use may lead to a higher design cost for lead products, but derivative products should be cheaper to design [13].

In Figure 1 we show a possible trade-off of the level of abstraction or services of an interaction language against the cost of designing a system using that interaction language. The conclusion that an interaction language lowers the cost of system design by being sufficiently abstract, by offering the appropriate services for the application domain, and enabling re-use is trite by itself, but must be balanced by the manufacturing costs of the design resulting from the use

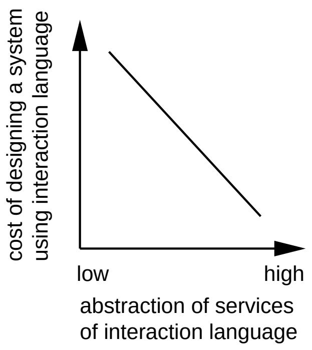

Fig. 1. Services versus cost of designing.

of the interaction language. Care must be taken that the interaction language offering perfect services is not too expensive for its intended use (also argued in [12]). We return to this trade off in Section 3.4.

3 Implementation of Interaction Language Services

The design of high-volume consumer products is driven by cost in two ways. The first, as we saw in Section 2, is the cost of designing a system (with as major component the time to market). In this section we focus on the second, the cost of a single device. We commence by classifying the ways in which an interaction language can be implemented, in Section 3.1. Naturally, the services that an interaction language offers influence the implementation choices, as will constraints on the implementation architectures (such as re-use of existing hardware and software components, platforms, etc.). These ideas are described in Section 3.2 and illustrated using the running examples in Section 3.3. In Section 3.4 the manufacturing cost is then broken down and examined in detail. The next section (Section 4) then combines services and the cost of designing of Section 2, and implementations and the manufacturing cost of this section.

3.1 A Classification of Interaction Language Implementations

Implementations of an interaction language can be classified at a high level along a number of axes. An implementation has varying cost depending on the choices taken but the cost is also related to the match with the services that are to be offered by the interaction language.

Centralised versus distributed

A centralised implementation of an interaction language concentrates all functionality in one place. It may therefore be easier to make than a distributed implementation because all coordination, communication, and inspection services converge on a single server. A single server avoids interference, synchronisation and consistency problems for shared data and control, and for services such as reconfiguration. The main disadvantage is its fundamentally non-scalable nature, resulting in lower reconfiguration, data, and synchronisation rates.

Hardware versus software

Multimedia systems require both efficiency and flexibility. Embedded systems therefore contain heterogeneous components of varying flexibility and performance (CPUs, VLIWs, application-specific instruction-set processors (ASIP), dedicated and programmable hardware blocks, etc.). Interaction languages must be implemented efficiently on all these components (i.e. in hardware and in software) to provide transparent communication. C-HEAP, one of our running examples, has been implemented in software on programmable components such as MIPS, ARM, and ASIP, and in a dedicated hardware block [24].

Emulated versus native

An interaction language can provide all its services as an integral part of the language. Usually such *native* implementations of the services are efficient because optimisations across services are possible. Alternatively, a basic set of services can be implemented natively, while the remainder is *emulated*, i.e. provided as a library or a layered protocol [12]. An example is the coherent shared-memory service [7], which can be implemented natively (in hardware) or emulated (in software on a non-coherent shared-memory service). Naturally, emulated services tend to be implemented in software, and native services in hardware or software, depending on the component they are used by (Table 3).

combination of implementation choices	consequences
emulated & software	flexible, slower
emulated & hardware	difficult to achieve
native & software	rigid, but upgradable, slower
native & hardware	rigid, faster

Table 3. Some examples of interactions of implementation choices.

3.2 Interaction Language Implementations: A Compromise of Services and Architecture

An interaction language implementation must obviously match the services that are supported. Table 4 shows some combinations of services and implementations. Some choices reinforce each other, while others cannot be usefully combined. Global services tend to have a centralised implementation; they are harder

combination of service and implementation choices	consequences
global & centralised	not scalable, slower & optimal (possibly)
global & distributed	not scalable, fast & optimal
local & centralised	not useful
local & distributed	scalable & fast, but not optimal
dynamic & software	best resource utilisation, but slow
dynamic & hardware	same, but hard to achieve
static & software	less useful
static & hardware	fastest, but worst resource utilisation

Table 4. Some examples of interactions of service and implementation choices.

to implement when information is distributed in the system. Local services can be distributed more easily, and can result in scalable implementations. Global services that have a distributed implementation are possible (e.g. global synchronisation using a token ring) but still lack scalability. Services are often more easily implemented in software, but at a lower performance than hardware. Moreover, the more dynamic a service, the higher the rate at which decisions are taken. The ideal situation consists in a dynamic hardware implementation. This is often hard to achieve, and dynamic software and static hardware are compromises (the requirements of the service are opposed by the implementation).

The architecture of the system is also important in the implementation of the services. For example, if the underlying architecture is based on message passing then a shared-memory service is expensive to implement. The reverse may be easier, but is still not necessarily very efficient.

An interaction language must balance services offered, to suit an application, and the implementation cost of those services, to fit an underlying architecture. Rightly or wrongly, in embedded systems the latter often is given more consideration. This may lead to a lower manufacturing cost, but almost certainly to a higher cost of designing.

3.3 Examples Continued

As shown in Section 3.1, different implementations of the same service are possible. Using the examples of Section 2.3, we show that implementations and services of interaction language are related, and lead to different costs. We now describe why STM, C-HEAP, and Arachne are implemented in the way they are, and the consequences regarding the manufacturing cost.

STM is implemented in software to support distributed garbage collection. It is based on a multiple-address-space distributed message-passing architecture to support clusters of symmetric multiprocessors. It assumes the provision of atomic operations (e.g. read-modify-write) by the underlying architecture. Latency to services is higher (approximately 20000 cycles). Data is often copied in message-passing architectures; this can be avoided using shared memory, like in C-HEAP.

C-HEAP uses a distributed implementation to provide local memory management services. Services are fairly static: reallocation of memory, for example, is possible only by reconfiguring explicitly. A single-address-space (distributed) shared-memory architecture reduces data copying to minimise resources (buffer sizes and bus load). No atomic operations (such as read-modify-write) are required, to ease implementation. All the services offered by the interaction language are native, and are transparent across hardware and software. The combined result of these choices is that the latency to execute services is low (approximately 30-100 cycles). There are several software and hardware implementations of the interaction language. Local memory management may be pessimistic when buffers are needed for different channels at different times. Global memory management, like that performed in Arachne, aims to address this.

Arachne uses a centralised implementation to facilitate global memory management. It uses single-address-space (distributed) shared-memory architecture for the reasons mentioned before. All the services are native, and are transparent across hardware and software. Services are executed with medium latency (approximately 100-3000 cycles). There are hardware and software implementations of the interaction language.

We see that the implementation of each interaction language matches the type of services: C-HEAP's local static services & distributed implementation, Arachne's global dynamic services & centralised implementation. STM's global services are offered on a symmetric multiprocessor platform and are therefore emulated in software. The implementation choices of each interaction language are reflected in its manufacturing cost. For example, the manufacturing cost is affected by the latency to access services: higher latency means that data and command buffers must be larger to avoid stalling tasks. Reducing the latency reduces buffer sizes, which translates to a smaller chip area, and hence lower cost. The next section further elaborates these cost aspects.

3.4 Implementations and the Manufacturing Cost

A design (the blueprint) is manufactured to a device. The manufacturing costs include the bill of materials (e.g. related to the silicon area), cost of testing, and so on. To avoid the explicit distinction between the cost of the device and the cost of the design that led to it, we define the term *manufacturing cost*. The resources a design uses are reflected directly in the cost of the device. Examples are the code size of a program (resulting in embedded or external memory), and memory management (leading to more or less memory). Other examples are power-efficient computation and communication, electromagnetic interference, and number of IO pins of a device. All affect the price of the chip package, and must be taken into account during the design process.

Recall from Table 1 that the manufacturing cost is composed of the cost of the implementation while using the interaction language (the *cost of use of interaction language*), the cost of the implementation of the interaction language itself (the *interaction language implementation cost*), and the cost of using the interaction language in an implementation (the *running cost*).

The *cost of use of interaction language* is basically the cost of the implementation of the required application using the given interaction language. Whereas the the cost of designing measures the effort spent in getting to the design, the cost of use of the interaction language quantifies the end result, ignoring the cost of implementing the interaction language itself. Usually, for an interaction language at a high level of abstraction both the cost of designing (Figure 1) and the cost of the use of the interaction language are low (Figure 2(A)) because, to some extent, the interaction language absorbs application functionality.

The *interaction language implementation cost* depends on the abstraction level of the services it has to offer as well as on the architecture on which it has to be implemented. Implementing an interaction language with few services is always cheaper than implementing an interaction language offering more. However, Figure 1 shows that offering few services raises the cost of designing (i.e. getting to a design takes longer). Offering too many services, and thus raising the interaction language implementation cost, is also to be avoided. An option only open to emulated services is that services that are not used in a design can be omitted from the implementation of the interaction language. The services are tailored to the design, as it were. Examples are Horus [30], x-kernel [18], and Koala [29]. It will not escape attention that offering the service of protocol stripping will not be free. See Figure 2(B) for some possible relationships between the services and their cost of implementation. An emulated implementation (curve 2(B)I-2(B)III) may be cheaper than a native implementation for few services (curve 2(B)II-2(B)IV) but lose when cross-service optimisations kick in (2(B)V).

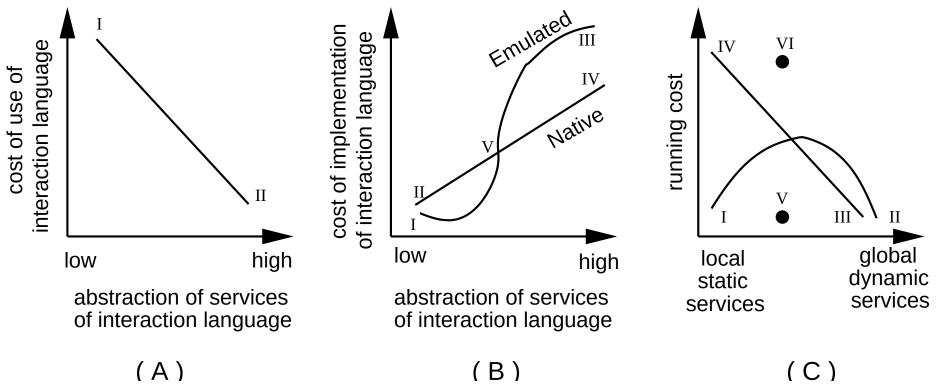

Fig. 2. Services, implementations, and costs.

Another influence on the manufacturing cost is the *running cost* of a interaction language. A first example is memory management. When the average-case and worst-case memory requirements vary in time and throughout the system, global dynamic memory management will use fewer resources than local static memory management. Arachne and STM fall in the first category. C-HEAP, with its local static memory management, has pessimistically sized buffers. In the end, however, its buffer sizes may be comparable to those of Arachne and STM (2(C)II) because its efficient distributed implementation offers services with lower latency [15] (2(C)I), reducing the sizes of its buffers.

A second example is scheduling. Dynamic global (multi-processor) pre-emptive task scheduling may have minimal running costs (2(C)II) but expensive to achieve (2(B)IV). Dynamic local (per-processor) task scheduling can be pre-emptive, or by explicit hand-over (using synchronisation primitives like setjmp/longjmp, coroutines [19], etc.) at medium implementation cost (2(B)V). Static local task scheduling avoids run-time scheduling altogether and is cheap to implement (2(B)II). It will perform badly when processor loads vary much over time and place (2(C)IV), when dynamic global task scheduling will excel (2(C)III).

A third example is the coherent shared-memory service [7]. A native hardware implementation perhaps costs more than an emulated software implementation (2(B)IV versus 2(B)III) but is more efficient at run-time (2(C)V versus 2(C)VI).

We conclude that for identical services different implementation choices result in different manufacturing costs (i.e. cost of use of interaction language, cost of interaction language implementation, and running cost).

4 Closing Remarks

In this paper we focus on high-volume consumer products for multimedia applications. Multimedia applications are increasingly dynamic and complex. Interaction languages help to contain this complexity by structuring the design process. This includes separating communication and coordination from computation. Furthermore abstraction, composition, and application-specific services (e.g. FIFO-based streaming) are offered.

High-volume consumer products are implemented as embedded systems that are cost driven. Interaction languages impact the cost in two ways: the cost of designing and the manufacturing cost (Table 1). We classify services that an interaction language can offer in coordination services, communication services, and inspection services. The ensemble of services must match the application domain of the interaction language. Offering more services will reduce the cost of designing an embedded system (see Figure 1). Services to enhance re-usability of (parts of) systems may lead to a higher design cost for lead products, but derivative products should be cheaper to design.

Next, we categorise the ways in which services can be implemented. Examples are distributed versus centralised, and native versus embedded implementations. The implementation choices must match the services:- a global service is better implemented centrally than in distributed fashion, for example.

We divide the manufacturing cost into the cost of using the interaction language, the implementation cost of interaction language, and its running cost. An interaction language with more services reduces the cost of use (Figure 2(A)), may reduce the running costs (Figure 2(C)), but is more expensive to implement (Figure 2(B)).

The final cost of a product is made up of a combination of the cost of designing (especially influenced by the time to market) and the manufacturing cost. Defining and using the right interaction language for the application domain amounts to finding the right trade-off between the various costs. These trade-offs can be judged only by the cost of the final products.

References

1. F. Arbab, C. L. Blom, F. J. Burger, and C. T. H. Everaars. Reusable coordinator modules for massively concurrent applications. *Software Practice and Experience*, 28(7):703–735, 1998.
2. F. Arbab, F. S. de Boer, and M. M. Bonsangue. A coordination language for mobile components. In *Proceedings of SAC, ACM press*, pages 166–173, 2000.
3. Farhad Arbab. The IWIM model for coordination of concurrent activities. In *Coordination languages and models*, volume 1061 of *Lecture notes in computer science*, pages 34–56, 1996.
4. M. Boasson. Control systems software. In *IEEE Transactions on Automatic Control*, volume 38, pages 1094–1106, July 1993.
5. R.J. Bril, C. Hentschel, E.F.M. Steffens, M. Gabrani, G. van Loo, and J.H.A. Gelissen. Multimedia QoS in consumer terminals. In *IEEE Workshop on Signal Processing Systems*, pages 332–343, 2001.
6. Paolo Ciancarini and Davide Rossi. Jada - Coordination and Communication for Java Agents. In Jan Vitek and Christian Tschudin, editors, *Mobile Object Systems: Towards the Programmable Internet*, volume 1222, pages 213–228. Springer-Verlag: Heidelberg, Germany, 1997.
7. A. L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, and W. Zwaenepoel. Software versus hardware shared-memory implementation: A case study. In *Proc. of the 21th Annual International Symposium on Computer Architecture (ISCA'94)*, pages 106–117, 1994.
8. J. D. Day and H. Zimmerman. The OSI reference model. In *Proceedings of the IEEE*, volume 71, pages 1334–1340, 1983.
9. Gjalt G. de Jong, Bill Lin, Carl Verdonck, Sven Wuytack, and Francky Catthoor. Background memory management for dynamic data structure intensive processing systems. In *Proceedings of the international conference on computer-aided design*, pages 515–520, 1995.
10. E.A. de Kock, G. Essink, W.J.M Smits, and P. van der Wolf. YAPI: Application modeling for signal processing systems. In *Proceedings 37th Design Automation Conference*, 2000.
11. S. Edwards, L. Lavagno, E.A. Lee, and A. Sangiovanni-Vincentelli. Design of embedded systems: Formal models, validation, and synthesis. In *Proceedings of the IEEE*, volume 85, pages 366–390, 1997.
12. D. Engler, M. Kaashoek, and J. O'Toole Jr. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, 1995.
13. A. Ferrari and A. Sangiovanni-Vincentelli. System design: traditional concepts and new paradigms. In *International Conference on Computer Design*, pages 2–12, 1999.
14. Eric Freeman, Susanne Hupfer, and Ken Arnold. *JavaSpaces: Principles, Patterns and Practice.* Addison-Wesley, 1999.

15. O. P. Gangwal, A. Nieuwland, and P. E. R. Lippens. A scalable and flexible data synchronization scheme for embedded HW-SW shared-memory systems. In *International Symposium on System Synthesis*, pages 1–6, Montréal, October 2001.
16. K. G. W. Goossens. A protocol and memory manager for on-chip communication. In *International Symposium on Circuits and Systems*, volume II, pages 225–228, Sydney, May 2001. IEEE Circuits and Systems Society.
17. Christine R. Hofmeister. *Dynamic Reconfiguration of Distributed Applications*. PhD thesis, Computer Science Department, University of Maryland, College Park, 1993.
18. N. C. Hutchinson and L. L. Peterson. The x-kernel: An architecture for implementing network protocols. *IEEE Transactions on Software Engineering*, 17(1):64–76, 1991.
19. G. Kahn and David M. MacQueen. Coroutines and networks of parallel processes. In B. Gilchrist, editor, *Proceedings of IFIP*, pages 993–998, 1977.
20. Jeffrey Kang, Albert van der Werf, and Paul Lippens. Mapping array communication onto FIFO communication – towards an implementation. In *Proceedings of international symposium on system synthesis*, pages 207–213, 2000.
21. J. Kramer and J. Magee. The evolving philosophers problem: Dynamic change management. *IEEE Transactions on Software Engineering*, 16(11):1293–1306, 1990.
22. Ronaldo Menezes. Experiences with memory management in open Linda systems. In *ACM Symposium on Applied Computing*, pages 187–196, March 2001.
23. Hyunok Oh and Soonhoi Ha. Data memory minimization by sharing large size buffers. In *ASP-DAC*, Tokyo, January 2000.
24. Rafael Peset Llopis, Marcel Oosterhuis, Sethuraman Ramanathan, Paul Lippens, Albert van der Werf, Steffen Maul, and Jim Lin. HW-SW codesign and verification of a multi-standard video and image codec. In *IEEE International Symposium on Quality Electronic Design*, pages 393–398, 2001.
25. Umakishore Ramachandran, Rishiyur S. Nikhil, Nissim Harel, James M. Rehg, and Kathleen Knobe. Space-time memory: a parallel programming abstraction for interactive multimedia applications. In *10th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, May 1999.
26. A. Rowstron and A. Wood. Bonita: a set of tuple space primitives for distributed coordination. In *Proceeding of HICSS30, SW Track*, pages 379–388, Hawaii, 1997. IEEE Computer Society Press.
27. D.B. Stewart, R. A. Volpe, and P. K. Khosla. Design of dynamically reconfigurable real-time software using port-based objects. *IEEE transactions on Software Engineering*, 23(12):759–776, 1997.
28. Marino T.J. Strik, Adwin H. Timmer, Jef L. van Meerbergen, and Gert-Jan Rootselaar. Heterogeneous multi-processor for the management of real-time video & graphics streams. *IEEE Journal of Solid-Sate Circuits*, 35(11):1722–1731, November 2000.
29. Rob van Ommering, Frank van der Linden, Jeff Kramer, and Jeff Magee. The Koala component model for consumer electronics software. In *IEEE Computer*, volume 33, pages 78–85, 2000.
30. Robbert van Renesse, Kenneth P. Birman, and Silvano Maffeis. Horus: A flexible group communication system. *Communications of the ACM*, 39(4):76–83, April 1996.