Extending component-based design with hardware components [⋆]

Péter Arató, Zoltán Ádám Mann, András Orbán

Budapest University of Technology and Economics
Department of Control Engineering and Information Technology
H-1117 Budapest, Magyar tudósok körútja 2, Hungary
Phone: +36 14632487, Fax: +36 14632204

Abstract

In order to cope with the increasing complexity of system design, component-based software engineering advocates the reuse and adaptation of existing software components. However, many applications – particularly embedded systems – consist of not only software, but also hardware components. Thus, component-based design should be extended to systems with both hardware and software components.

Such an extension is not without challenges though. The extended methodology has to consider hard constraints on performance as well as different cost factors. Also, the dissimilarities between hardware and software (such as level of abstraction, communication primitives etc.) have to be resolved.

In this paper, the authors propose such an extended component-based design methodology to include hardware components as well. This methodology allows the designer to work at a very high level of abstraction, where the focus is on functionality only. Non-functional constraints are specified in a declarative manner, and the mapping of components to hardware or software is determined automatically based on those constraints in the so-called hardware/software partitioning step.

Moreover, a tool is presented supporting the new design methodology. Beside automating the partitioning process, this tool also checks the consistency between hardware and software implementations of a component.

The authors also present a case study to demonstrate the applicability of the outlined concepts.

Key words: component-based design, hardware/software co-design, hardware/software partitioning

[⋆] This work has been supported by the European Union as part of the EASYCOMP project (IST-1999-14151) and by OTKA T 043329
Email address: `arato@iit.bme.hu, {Zoltan.Mann,Andras.Orban}@cs.bme.hu`

1 Introduction

The requirements towards today's computer systems are tougher than ever. Parallel to the growth in complexity of the systems to be designed, the time-to-market pressure is also increasing. In most applications, it is not enough for the product to be functionally correct, but it has to be cheap, fast, and reliable as well.

Component-based software engineering holds the promise of overcoming the design productivity gap by the systematic reuse of software components [1,2].

In this paper, we address the problem of using a component-based methodology in the design of *embedded systems*. According to a recent study, embedded software is an over $1.4 billion business and is growing steadily [3]. Embedded systems have become a part of our lives in the form of consumer electronics, cell phones, smart cards, car electronics etc. These computer systems consist of both hardware and software. The design of the hardware and software parts cannot be done separately because they depend heavily on each other. Therefore, the design of such systems involves *hardware/software co-design* (HSCD [4]). It should also be noted that the differences between hardware and software and their interaction also contribute significantly to the complexity of the systems. Therefore, the design of such systems would also benefit from a component-based methodology.

Of course, the reuse of previously designed components is not unfamiliar in the hardware world either. Actually, because of the high costs of hardware production, the idea of reusing existing units and creating the new applications out of the existing building blocks is even more adopted in the hardware world. This process has led from transistors to logic gates, then to simple circuits like flip-flops and registers, and then to more and more complex building blocks like microprocessors. Today's building blocks perform complex tasks and are highly adaptable. These building blocks are called IP (intellectual property) blocks [5–7].

Despite the striking similarity between IP blocks and software components, there are also some important differences:

- Since modifiability is not a key issue in hardware design, there is no strict decoupling between the interface and the implementation of an IP block.
- Similarly, there are no standardized high-level component models (such as *e.g.* CORBA or EJB in the software world), nor supporting middleware platforms.
- The 'interface description' of an IP block (which is typically just a textual

(Péter Arató, Zoltán Ádám Mann, András Orbán).

description and a data sheet) is very low-level, focusing on voltages and clock signals.

This paper introduces a component-based design methodology that handles hardware and software components in a uniform way by using a generic component notion focusing on functionality, and by using software-like interface adapters for hardware. The methodology is described in Section 2. The authors have also developed a tool supporting the new concepts, which is demonstrated in Section 3. Moreover, a case study was conducted to evaluate the practical applicability of the presented concepts (Section 4). Section 5 presents related work, and finally, Section 6 concludes the paper.

2 Extended component-based methodology

Based on the growing needs towards system design, as well as both the software and hardware industry's commitment to emphasize reuse as the remedy for growing design complexity, we propose a novel HSCD methodology we call component-based hardware-software co-design (CBHSCD). CBHSCD is an important contribution in the Easycomp (Easy Composition in Future Generation Component Systems [1]) project of the European Union. The main goal of CBHSCD is to assemble the system from existing pre-verified building blocks allowing the designer rapid prototyping [8,9] at a very high level of abstraction. At this abstraction level components do not know any implementation details of each other, not even whether the other is implemented as hardware or as software. The behavior of this prototype system can be simulated and validated at an early stage of the design process. CBHSCD also supports hierarchical design: the generalized notion of components makes it possible to reuse complex hardware-software systems as components in later designs.

2.1 Basic concepts

Our overall aim is to handle hardware components similar to software ones and use the existing software composition methodologies to assemble complex heterogeneous systems. To achieve this, hardware components should be provided with a software–like interface hiding all hardware–specific details. The need for a high–level abstract interface has led us to the following component notion.

We define a component as a *functional unit.* The composition of components

[1] www.easycomp.org

Table 1
Mapping between the hardware and software notations

Software notation		Hardware notation
property	$\longleftrightarrow$	status/state signal
method call	$\longleftrightarrow$	start/enable signal, command
event	$\longleftrightarrow$	interrupt

is based on their functionality. This functionality is captured by the interface of the component. It is described in a very generic way, via *methods*, *properties* and *events*. Although these terms originate from the software–world, they are general enough to capture the *functional behavior* of hardware elements as well. The status and state signals of the hardware component can be mapped to properties, the various start/enable signals and commands to method calls, and the interrupts to events. This mapping (see Table 1) is realized by a *wrapper* surrounding the hardware component. To be exact, the wrapper is designed around the device driver communicating directly with the hardware (see Fig. 1). The device driver and the wrapper together hide all hardware-specific details including port reads/writes, direct memory access (DMA) etc.: these are all done inside the wrapper and the device driver, transparently for other components. As a consequence hardware components behave exactly the same way as software ones for the rest of the system, they can be composed as software components, they can also participate in remote method calls both as initiator or as acceptor.

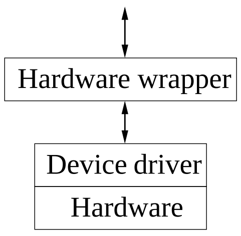

Fig. 1. Wrapper around a hardware component to achieve software–like interface

The functionality captured by the interface is completely decoupled from its implementation. It is also possible to have *more than one implementation* for the same interface. What is more, it is possible that there is a hardware implementation and a software implementation for the same interface. One of our main motivation was to achieve *hardware/software transparency*, which means that a change between the two implementations is transparent to the rest of the system, hence the decision which implementation to use can be made as late as possible. The designer works with abstract behavioral units in the majority of the design process.

We can identify three different kinds of components. There can be components for which there is only a software implementation. For instance, GUI elements or a database component are typically implemented in software. Similarly, there can be components for which there is only a hardware implementation. For example, it does not make sense to implement a video card in software. And finally, there can be components for which there is both a hardware and a software implementation. For instance, a cryptographic algorithm can be realized either by a program or by a special-purpose hardware unit. Such components will be called *partitionable components* (see Fig. 2).

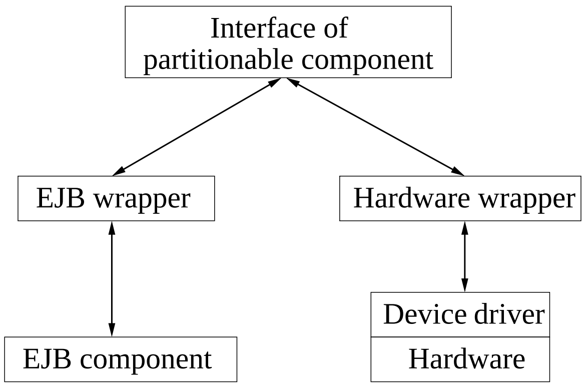

Fig. 2. Example partitionable component with two implementations

After transforming the components to a generic abstract component model, the composition and communication between components can be realized with existing software methodologies [10]. The communication between the components is facilitated through a middleware layer, which consists of the wrappers for the respective component types, as well as support for the naming of components, the conversion of data types and the delivery of events and method calls. (See Section 4 for an example.) This way we can achieve hardware-software transparency much in the same way as middleware systems for distributed software systems achieve location and implementation transparency. The resulting architecture is shown in Fig. 3. Note that the adapters facilitating the event–to–method mapping can be generated automatically [10].

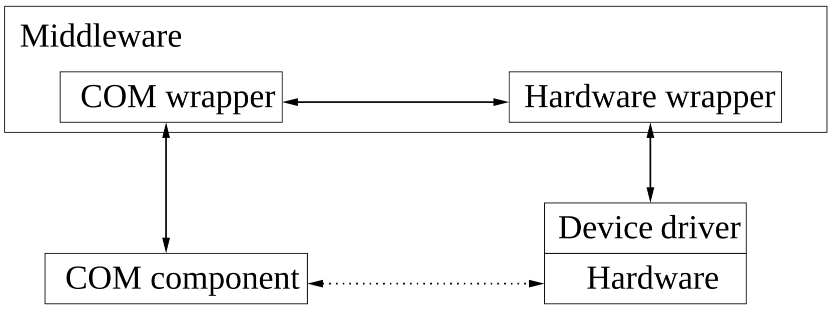

Fig. 3. Communication between a COTS software component (COM component in this example) and a hardware unit. The dotted line indicates the virtual communication, the full line the real communication.

The drawback of this approach is the large communication overhead introduced by the wrappers and the middleware layer in general. Furthermore, this is only problematic if the communication between hardware and software involves many calls, which is not typical. Most often, a hardware unit is given an amount of data on which it performs computation-intensive calculations and then it returns the results. In such cases, if the amount of computation is sufficiently large, the communication overhead is less important. However, the flexible but complicated wrapper structure is only used in the design phase, and it is replaced by a simpler, faster, but less flexible communication infrastructure in the synthesis phase. There are standard methodologies for that task, see *e.g.* [11,6].

2.2 CBHSCD process

The main steps of CBHSCD are shown in Fig. 4. In the following each subtask is detailed except the issues related to *synthesis* which are beyond the scope of CBHSCD.

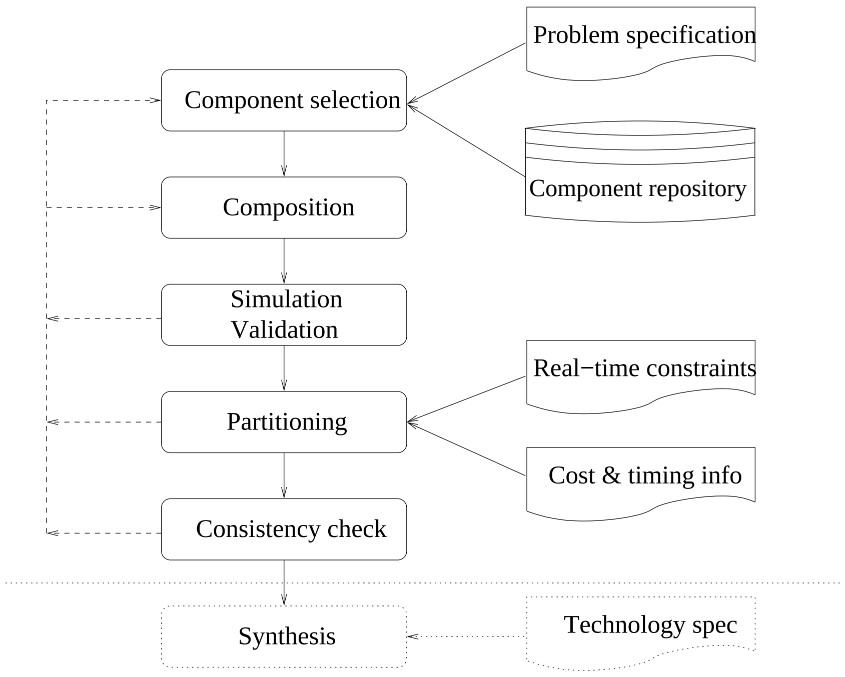

Fig. 4. The process of CBHSCD

Component selection. The process starts by selecting the appropriate components from a component repository based on the problem specification. (A related field of research addresses the problem of automating the component selection process, see *e.g.* [5,12], but this is orthogonal to CBHSCD.) From the aspect of CBHSCD it does not matter how the components are implemented: CBHSCD does not aim at replacing or reinventing specific hardware design and synthesis methods or software development methods. Instead, it relies on existing methodologies and best practices, and only complements them with co-design aspects. The used components might include pure software and pure hardware components, but mixed components are also allowed, as well as components which exist in both hardware and software. In the latter case the designer does not have to decide in advance which version to use (only the functionality is considered), but this will be subject to optimization in the partitioning phase.

Composition. After the components are selected, they are composed to form a prototype system. This composition mechanism deals with abstract functional units with interfaces; the implementation issues, the hardware/software boundary are irrelevant at this stage.

Each component provides an interface for the outside world. The specification of this interface is either delivered with the component or if the component model provides a sufficient level of *reflection*, it can be generated automatically. One of the important contributions of CBHSCD is that the composition of all components is based on remote method calls between components supported by the underlying middleware. Since hardware components are also transformed to the generic component model, they can be composed using the technique in [10].

Composition is supported by a visual tool that provides an intuitive graphical user interface as well as an easy-to-use interconnection wizard. This ease-of-use helps to overcome problems related to the learning-curve, since traditionally system designers have had to possess professional knowledge on hardware, software and architectural issues; thus, the lack of qualified system designers has been a critical problem.

Simulation and validation. Since the application has been composed of tested and verified components, only the *correctness of the composition* has to be validated by simulation. The individual units are handled as black-box components in this phase and only functional simulation is carried out. For instance, if a calculation is required from a hardware component, one would only monitor the final result passed back to the initiator component and not the individual steps taken inside the hardware. If problems are detected, the component selection and/or composition steps can be reviewed.

It is important to note that components are fully operable at composition time (*e.g.* a button can be pressed and it generates events), hence the application can be tried out by simply triggering an event or sending a start signal to a component. This helps validate the system enormously.

Since the design is only in a premature prototyping phase, it is possible that the (expensive) hardware components are not available at this stage[2]. If the

[2] Before partitioning it is not even known of each component whether to be realized

hardware component is already available and the component is decided to be in the hardware context, it can be used already in the simulation phase. However, it is possible that we want to synthesize or buy the hardware component only if it is surely needed. In this case, we can use software simulation instead.

Note that simulation concentrates only on the functionality and not on the timing characteristics of the system. The latter is unfortunately distorted by the several indirections caused by the wrappers around the components. Of course these wrappers are eliminated during synthesis.

Partitioning. After the designer is convinced that the system is functionally correct, the system has to be partitioned, *i.e.* the partitionable components have to be mapped to either their software or hardware implementation. (The 'mapping' of components which only exist in hardware or only in software is trivial.) This is an important optimization problem, in which the optimal trade-off between cost and performance has to be found, since hardware is typically faster but more costly than software. (See Section 4 for an example.) Traditionally, this has been the task of the system designer, but manual partitioning is very time-consuming and often yields sub-optimal solutions.

CBHSCD on the other hand makes it possible to design the system at a very high level, only concentrating on functionality. This frees the designer from dealing with low-level implementation issues. It is important to note that up to this point, the design process is *completely implementation-independent.* Partitioning is automated based on a declarative requirements specification. We defined a graph-theoretic model for the partitioning problem and developed appropriate algorithms for it [13,14]. The partitioning algorithm takes into account the software running times, hardware costs (price, area, heat dissipation, energy consumption etc.), communication costs between components as well as possible constraints defined by the user (including soft and hard real-time constraints, area constraints etc.). This is very helpful for the design of embedded systems, especially real-time systems. When limiting the running time, partitioning aims at minimizing costs. Similarly, when costs are limited, the running time is minimized. It is also possible to constrain both running time and costs, in which case it has to be decided whether there is a system that fulfills all these constraints, and in the case of a positive answer, such a partition has to be found.

To generate all the input data for the partitioning algorithm is rather challenging. In case of *hardware costs*, it is assumed that the characteristic values of the components are provided with the component itself by the vendor. *Communication costs* are estimated based on the amount of exchanged data and the communication protocol. Concerning the *running times*, a worst case (if hard real-time constraints are specified) or average case running time is either

in software or hardware.

provided with the component or extracted by profiling techniques. An independent research field deals with the measurement or estimation of these values, see *e.g.* [15,16]. The time and cost *constraints* must be specified explicitly by the designer via use cases (see Section 3 for more details).

Consistency check. The requirement of hardware/software transparency implies two consistency problems specific to CBHSCD. Note that we are not dealing here with the—otherwise very important—consistency of the composition which occurs in pure software composition as well [17], only with additional consistency problems due to hybrid hardware/software systems.

The first is the *interface consistency problem.* The question to answer here is whether or not two implementations can form a partitionable component.

To ensure interface consistency, the compulsory features of a given component, that is: those features that both implementations have to implement, are specified in an interface description file. Then, it is automatically checked if the two implementations do implement the required features.

The second is the *state consistency problem.* The prototype system is likely to be partitioned and repartitioned several times during the design process. Each time to realize a transparent swap between implementations, the new implementation should be set to exactly the same state as the current one, because otherwise it might behave differently in the future than expected by the rest of the system. The designer might not want to reset the whole system to its initial state (and restart the simulation process) every time the system is repartitioned, so the state consistency must be handled in a more sophisticated way. It is not straightforward to achieve this, because the components are regarded as black-box, and it is generally not possible to access all the state-variables from the outside. (A number of component models explicitly forbid stateful components to avoid these problems.)

To address the state consistency problem we should define the notion of *consistent states*:

> *Two implementations are in consistent states if the same sequence of method calls can be executed on them and they produce the same output.*

The consequence of this definition is, that starting the two implementations from consistent states and executing the same sequence of method calls on them, they will be again in consistent states.

Assuming that the initial states of the two implementations are consistent, we can bring the newly selected implementation to a state consistent with the old one by automatically repeating the same sequence of method calls and property changes that had been performed on the old one by the outside world since the last consistent states. (Remember that the old and new implementations offer the same interface, hence providing exactly the same methods, properties etc.) This way, we can guarantee that consistent states are reached.

A special attribute is associated with every method in the description of the component which specifies the effect of this method on the state of the component. The appropriate methods to repeat have to be selected intelligently according to these attributes. (See Section 3 for more details.)

3 CWB-X: a tool for CBHSCD

Our tool to support CBHSCD is an extension of a component-based software engineering tool called Component Workbench (CWB), which has been developed at the Vienna Technical University in the Easycomp project [10].

CWB is a graphical design tool for the easy composition of applications from COTS software components. The main contribution of CWB is the support for multiple component models, like COM, CORBA, EJB etc. To achieve this, CWB uses a generic component model called Vienna Composition Framework (VCF). This generic model offers a flexible way to represent components, hence all existing software component models can be transformed to this one by means of wrappers.

In the philosophy of CWB, each component is associated with a set of *features*. A feature is anything a component can provide. A component can declare the features it supports and new features can also be added to the CWB. The most typical features are the following.

Property The properties (attributes) provided by the component.
Method The methods of the component.
Eventset The sets of events the component can emit.
Lifecycle If a component has this feature, then it can be created and destroyed, activated or deactivated.
GUI The graphical interface of the component.

Each component model is implemented as a *plug-in* in the CWB (see Fig 5). The plug-in class only provides information about the features the component can offer, the real functionality is hidden in the classes implementing the features. New component models can be implemented by creating a new plug-in class and a class for each required feature.

For the communication between the components, CWB offers event-to-method communication, *i.e.* a component triggers an event which induces a method

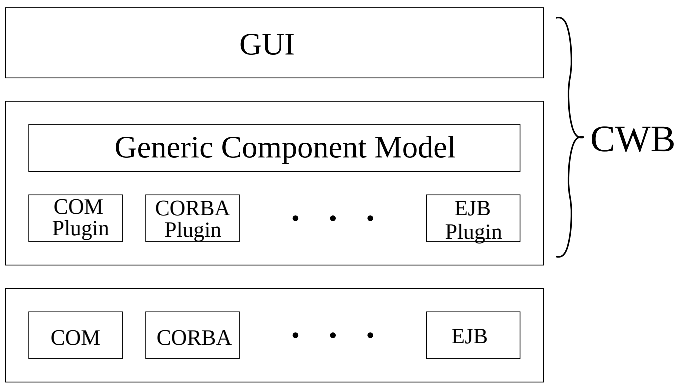

Fig. 5. The architecture of the CWB.

call in all registered components. The registration mechanism and the remote method call is supported by Java. A wizard helps the user to set up a proper connection. Adapters are automatically created to facilitate the communication between the components.

The used components are already operable at composition-time. This is very advantageous because this way the simulation and evaluation of the system is possible already in the early phases of the design process. Also, the user can invoke methods of the components, thus use cases or call sequences can be tested without any programming efforts.

3.1 Extension of CWB to support CBHSCD

CWB offers a good starting point for a hardware-software co-design tool because of its flexibility and extensibility. We extended CWB to support CBHSCD principles. In CWB-X (CWB eXtended), the designer of a hardware-software application may select software, hardware, or partitionable components from a repository. These components can originate from different vendors and different component models. The selected component is put on the working canvas. In case of pure software components, the operable component itself—with possible GUI—can appear, but in case of hardware components the component itself might not be available and simulation is used.

To enable the integration of hardware components in CWB-X, new component models are added to the CWB as plug-ins. Similarly to the software side, there is a need for several hardware component models according to the different ways the actual hardware might be connected to the computer. This goal is complicated by the lack of widely accepted industry standards for IP interface and communication specification.

Since the implementation details of a component should be transparent for the other components, the hardware components should provide similar features as the software ones. Therefore we define the Method, Property and Eventset features for hardware components as well, and map methods to operations of the underlying hardware, properties to status information and initial parameters, and events to hardware interrupts. Our mapping is actually between the device driver and the generic component model of the CWB, hence the plug-in does not have to address hardware–specific low–level issues.

To identify the features a hardware component can provide, reflection is necessary, *i.e.* information about the interface of the component. Today's IP vendors do not offer a standardized way to do that, often a simple text description is attached to the IP. In our model we require a hardware component to provide a description about its features (Properties, Methods, Events).

The composition of components is supported by wizards. Due to the wrappers, hardware components act the same way as software ones, thus the wizards of the CWB can be used.

When the architecture of the designed application is ready, partitioning is performed. We have integrated a partitioning algorithm [13] based on integer linear programming (ILP). This is not an approximation algorithm: it finds the *exact optimum*. This approach can handle systems with several hundreds of components in acceptable time. For the automatic partitioning process, the various cost parameters and the time constraints must be specified.

Time constraints are defined on the basis of use cases. A use case involves some components of the system in a given order. A component can also participate multiple times in a use case. The designer defines a use case by specifying the sequence of components affected in it and gives a time constraint for the sum of the execution times of the concerned components including communication. The constraints for all use cases are simultaneously taken into account during partitioning.

The partitioning algorithm also needs the estimated running times and communication cost parameters. As mentioned previously, the measurement or estimation of these cost values is a large independent research field (see *e.g.* [15,16]). We are aware of the importance of these measurements, but this is currently at an initial stage in our tool: we expect that this data is explicitly given.

CWB-X is able to check both interface and state consistency. To each partitionable component a Java-like interface is attached which describes the required features of the implementations. The tool checks whether the associated implementations are appropriate. Furthermore, to each method in this interface description file an attribute is ordered, which describes the behavior of this method in the state consistency check. The value and the meaning of the attribute are the following:

`NO_EFFECT`: the corresponding method has no effect on the state of the component, thus it should not be repeated after repartition.
`REPEAT_AT_REPARTITION`: the corresponding method affects the state but has no side effect, thus it should be repeated after repartition.
`REPEAT_AT_REPARTITION_ONCE`: the same as the previous one, but in a sequence of calls to this method only the last one should be repeated. An example is setting a property to a value.
`SIDE_EFFECT`: the method does affect the state and also has some side effect (*e.g.* sends 100 pages to the printer) or takes too long to repeat.

CWB-X logs every method call and property change since the last implementation swap. If all these belong to the first three categories, the correct state will be set automatically after the change of the implementations by repeating the appropriate function calls. If there is at least one call with `SIDE_EFFECT`, the system shows a warning and asks the designer to decide which methods to repeat. The designer is supported by a detailed log in this decision.

4 Case study

In this section, the CBHSCD methodology is demonstrated step by step on an example application. In this example a frequency modulated signal should be decoded. A signal generator (which is not part of the system to be designed) generates a signal with frequency modulation. The frequency of the incoming signal should be measured and the signal is decoded to 0 or 1 according to the measured frequency value. This task appears in several real-world applications. The architecture of the system can be seen in Fig. 6. The frequency measurer (FM) measures the frequency of the incoming signal and sends the measured value periodically to the demodulator unit (DU). The DU decodes the signal and sends the result to the displayer. The displayer consists of two components: a textual display shows the current value of the decoded signal, and a chart shows the graph of the alteration of the value. Furthermore, there are two buttons that control the measurer through start and stop signals. Both the FM and the DU are partitionable components.

There are two implementations available for the FM: the first one is a program on a PIC 16F876 microcontroller (software implementation) and a field-programmable gate array (FPGA) on a XILINX VIRTEX II XC2V1000 card (hardware implementation). The two implementations behave exactly the same way, but their performance (and cost) is different. The microcontroller is able to precisely measure the frequency up to 25kHz (*i.e.* taking a sample lasts 40μs). The FPGA on the other hand can take a sample in 50ns, thus it can

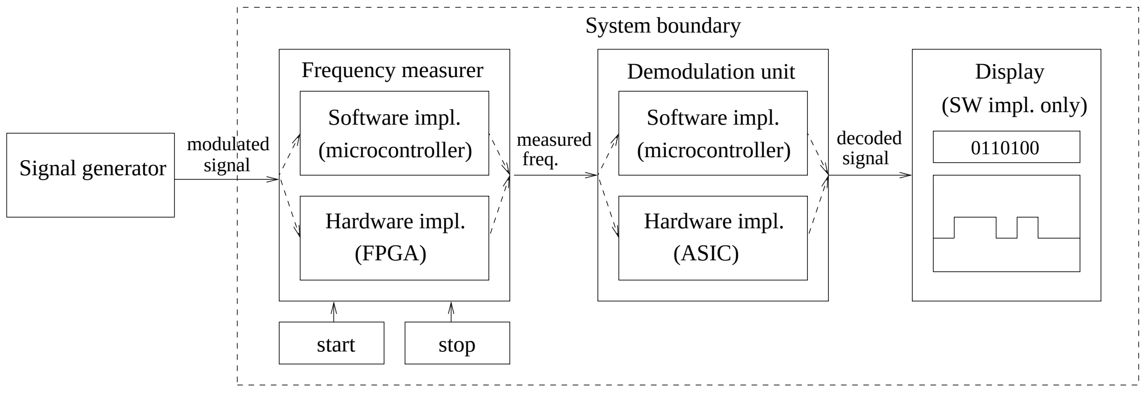

Fig. 6. The architecture of the example application

measure up to 20MHz without any problem. However, the FPGA is more costly. The software implementation of the DU runs on the same microcontroller, while the hardware implementation is an ASIC (application specific integrated circuit). Again the two implementations differ in price and performance.

The designer might want to impose several constraints on the system to be designed. These constraints are bound to use–cases of the system. In this example the following constraints are defined.

Frequency constraint (C1). The constraint defines an upper bound on taking one sample of the signal by the FM. This constraint implicitly declares the maximum frequency that should be handled correctly. In our example let us define the maximum frequency to be 50kHz, hence one sample should be taken in 20μs.

Response time constraint (C2). Prescribes the time needed for a sent bit to appear in the displayer, that is the response time of the system for an input. We define this limit as 1200μs.

The task of the partitioning algorithm is to decide which implementations to use for the partitionable components to satisfy the constraints. For the partitioning algorithm the system is converted to a graph representation with hardware and software costs on the vertices and communication costs on the edges. Fig. 7 depicts the graph corresponding to the demo application. (For simplicity, only the components affected by the constraints are shown.) For each partitionable node there are two values specified: the hardware cost (*e.g.* in \$) and the worst case software running time (in μs). For the edges the communication overhead is given (also in μs) [3] . Note that this cost arises only if the edge crosses the hardware/software boundary.

Table 2 shows the optimal hardware/software partitions found by our algo-

[3] The values are for demonstrative purposes only.

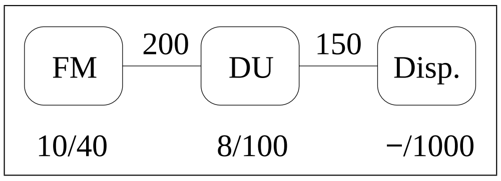

Fig. 7. The graph representation of the system for the partitioning algorithm

rithm. (See [13,14] for more details on the partitioning model.) If only constraint C1 is imposed, then FM should be implemented in hardware since in software it would require 40μs to take one sample, thus violating C1. To minimize cost, all the other components should be in software, and there are no other constraints forbidding this. If only C2 is required, then all components can be put into software, since the total running time is $40+100+1000 < 1200$. If both C1 and C2 are imposed, both FM and DU should be put into hardware. FM because of C1, and DU because of C2. (If we put only FM into hardware, the (FM,DU) edge would cross the hardware/software boundary, hence the total running time would be 200+100+1000, violating C2.) The example clearly demonstrates how the tool automatically achieves an optimal trade–off between price and performance.

Table 2
Optimal hardware/software partitions found by our tool according to the required constraints.

Constraints	Software	Hardware	Hardware cost
C1	DU, Disp	FM	10
C2	FM, DU, Disp	–	0
C1+C2	Disp	FM, DU	18

The described demo can be realized in CWB-X as follows. There are six components: two JavaBeans buttons (start and stop), a textfield and a chart component for display, the FM and the DU declared as a partitionable component with the two implementations detailed above[4]. The device driver of the hardware components is wrapped by a CWB wrapper providing a software-like interface. Special adapter classes are then generated automatically by the CWB for facilitating communication. Note that different hardware components with the same device driver interface (which is the case *e.g.* for the two implementations of the FM) require only one wrapper.

The structure of the system can be demonstrated on an example communication process. Fig. 8 shows the reaction of the system on pressing the start button. Each component on the figure is separated by a dashed line, the lower part contains the type of the component, while the upper part shows the name of the specific component. The arrows indicate the communication between

[4] The signal generator is regarded as an outside source, hence not part of the system

components, the labels on the arrows mean either a method call or other kinds of communication. One can see how the device driver hides all the hardware– and communication–specific details.

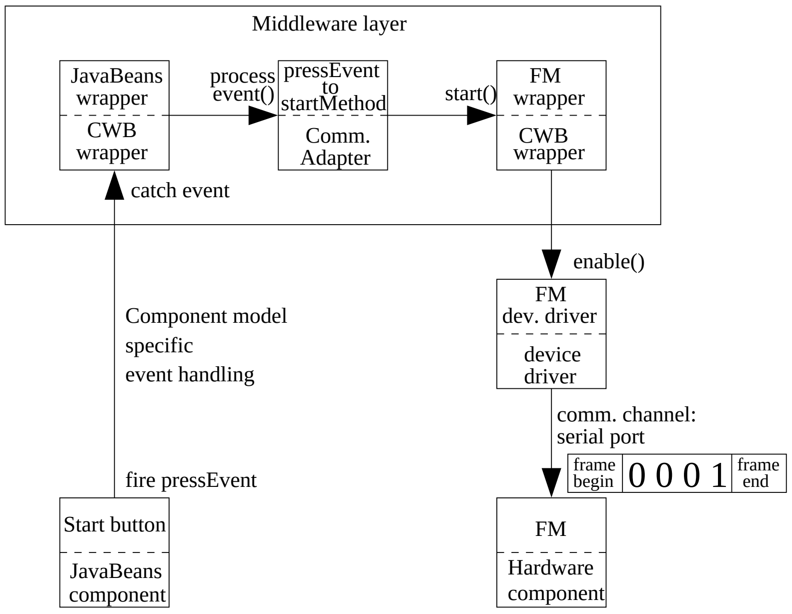

Fig. 8. Communication induced by pressing the start button.

For the purposes of the consistency check, an interface description of the required features is also provided with the component (Fig. 9). The tool checks whether the interfaces of the wrappers match the requirements. The interface description also contains the attributes necessary for the state consistency mechanism.

```
package frequency;

public interface FrequencyMeasurerInterface {

  SIDE_EFFECT public void start();
  SIDE_EFFECT public void stop();
  NO_EFFECT public void takeOneSample();
  NO_EFFECT public String getMeasuredFrequencyString();
  NO_EFFECT public Integer getMeasuredFrequency();
  REPEAT_AT_REPARTITION_ONCE public void setCountEveryEdge(boolean b);
  NO_EFFECT public boolean getCountEveryEdge();
  NO_EFFECT public void addFreqMeasuredEventListener(Listener l);
  NO_EFFECT public void removeFreqMeasuredEventListener(Listener l);
}
```

Fig. 9. Part of the required interface (with state consistency attributes) of the partitionable frequency measurer (FM) component

In the composition phase the start and stop buttons are mapped with the aid of the mentioned wizard to the start and stop methods of the FM, respectively. The FM sends an interrupt whenever a new measured value is ready. This interrupt appears as an event in CWB-X, which triggers the DU to ask for the measured value. When the DU is ready with the decoding, it again sends an interrupt. This triggers the `setText` function of the `TextField` and the `addValue` function of the chart. The system can be immediately simulated without any further effort: after pressing the start button the current implementation of the FM starts measuring the signal of the generator and the displayer displays the measured values. The system can be partitioned and repartitioned an arbitrary number of times based on the defined constraints; the state consistency mechanism makes sure that the change in implementation remains transparent to the rest of the system.

5 Related work

In recent years, there has been a substantial amount of work in the system design community targeting the design of SoC-s (System on Chip), and making use of existing components (IP blocks, also called Virtual Components) [18,6,19,20,7]. Unfortunately, most of these approaches handle low-level interconnection issues in system design, and provide very little tool support. There are hardly any standards for the interoperability of IP blocks. Although the VSI Alliance has published some standards and specifications in this field [21], they only handle the lowest levels of interconnection, *i.e.* the physical details.

A higher level of abstraction characterizes the approaches for hardware/software co-simulation [6,22,20,23]. They aim at enabling the simulation of the system in the early stages of the design, either functionally, or concerning both functionality and performance (real-time simulation). Our work also uses co-simulation; however, we support the designer with several automatisms (partitioning, consistency checks) as well.

Another interesting thread of related work is concerned with the automatic synthesis of hardware/software interfaces [11,24,6]. This approach is rather orthogonal to our work, because it aims at constructing run-time interfaces, whereas we focus only on the design phase, and hence use composition-time wrappers.

Component selection and trading [18,5,12] is also a related research field that is orthogonal to our approach. Here, the aim is to define description formats for components which enable the automatic retrieval of suitable components for a given task.

In the component-based software engineering community, the most strongly related efforts are those concerned with the *adaptation* of components [25–27,10]. Our way of using wrappers for this task is similar to these approaches; however, we also consider hardware components, which also leads to such issues as partitioning, which are not present in pure software systems.

6 Conclusion

This paper presented an extension to component-based software engineering to also include hardware components. The new methodology, called component-based hardware/software co-design (CBHSCD) provides a uniformly high level of abstraction for software, hardware, and partitionable components.

The concepts of CBHSCD, as well as partitioning, enable advanced tool support for the system-level design process. Our tool CWB-X is based on the Component Workbench (CWB), a visual tool for the composition of software components of different component models. CWB-X extends the CWB with new component models for hardware components as well as partitioning and consistency checking functionality. We presented a case study to demonstrate the applicability of our concepts and the usefulness of our tool.

We believe that the notion of CBHSCD unifies the advantages of hardware and software design to a synergetic system-level design methodology, which can help in designing complex, reliable and cheap computer systems rapidly.

References

[1] G. T. Heineman, W. T. Councill, Component Based Software Engineering: Putting the Pieces Together, Addison-Wesley, 2001.

[2] U. Assmann, Invasive Software Composition, Springer, 2003.

[3] Venture Development Corporation, The embedded software strategic market intelligence program 2002/2003, volume II, `http://www.vdc-corp.com/embedded/reports/03/br03-10.html` (2003).

[4] R. Niemann, Hardware/Software Co-Design for Data Flow Dominated Embedded Systems, Kluwer Academic Publishers, 1998.

[5] G. Martin, R. Seepold, T. Zhang, L. Benini, G. D. Micheli, Component selection and matching for IP-based design, in: Proceedings of Design, automation and test in Europe, IEEE Press, 2001.

[6] P. Chou, R. Ortega, K. Hines, K. Partridge, G. Borriello, IPCHINOOK: an integrated IP-based design framework for distributed embedded systems, in: Design Automation Conference, 1999, pp. 44–49.

[7] F. Pogodalla, R. Hersemeule, P. Coulomb, Fast protoyping: a system design flow for fast design, prototyping and efficient IP reuse, in: CODES, 1999.

[8] G. Spivey, S. S. Bhattacharyya, K. Nakajima, Logic Foundry: A rapid prototyping tool for FPGA-based DSP systems, Tech. rep., Department of Computer Science, University of Maryland (2002).

[9] K. Buchenrieder, Embedded system prototyping, in: Tenth IEEE International Workshop on Rapid System Prototyping, 1999.

[10] J. Oberleitner, T. Gschwind, Composing distributed components with the component workbench, in: Proceedings of the Software Engineering and Middleware Workshop (SEM2002), Springer Verlag, 2002.

[11] A. Basu, R. Mitra, P. Marwedel, Interface synthesis for embedded applications in a co-design environment, in: 11th IEEE International conference on VLSI design, 1998, pp. 85–90.

[12] P. Roop, A. Sowmya, Automatic component matching using forced simulation, in: 13th International Conference on VLSI Design, IEEE Press, 2000.

[13] Z. A. Mann, A. Orbán, Optimization problems in system-level synthesis, in: Proceedings of the 3rd Hungarian-Japanese Symposium on Discrete Mathematics and Its Applications, 2003.

[14] P. Arató, S. Juhász, Z. A. Mann, A. Orbán, D. Papp, Hardware/software partitioning in embedded system design, in: Proceedings of the IEEE International Symposium on Intelligent Signal Processing, 2003.

[15] X. Hu, T. Zhou, E. Sha, Estimating probabilistic timing performance for real-time embedded systems, IEEE Transactions on VLSI Systems 9 (6).

[16] S. L. Graham, P. B. Kessler, M. K. McKusick, An execution profiler for modular programs, Software Practice & Experience 13 (1983) 671–685.

[17] A. Speck, E. Pulvermüller, M. Jerger, B. Franczyk, Component composition validation, International Journal of Applied Mathematics and Computer Science (2002) 581–589.

[18] E. Casseau, Soc design using behavioral level virtual components, in: IEEE International Conference on Electronics, Circuits, and Systems, 2002.

[19] P. Coussy, A. Baganne, E. Martin, A design methodology for integrating IP into SOC systems, in: Conférence Internationale IEEE CICC, 2002.

[20] S. J. Krolikoski, F. Schirrmeister, B. Salefski, J. Rowson, G. Martin, Methodology and technology for virtual component-driven hardware/software co-design on the system level, in: ISCAS, 1999.

[21] VSI Alliance, VSIA architecture document, `http://www.vsi.org/resources/techdocs/vsi-or.pdf` (1997).

[22] J. Davis, C. Hylands, J. Janneck, E. A. Lee, J. Liu, X. Liu, S. Neuendorffer, S. Sachs, M. Stewart, K. Vissers, P. Whitaker, Y. Xiong, Overview of the Ptolemy project, Tech. rep., Department of Electrical Engineering and Computer Science, University of California, Berkeley (2001).

[23] P. L. Marrec, C. A. Valderrama, F. Hessel, A. A. Jerraya, M. Attia, O. Cayrol, Hardware, software and mechanical cosimulation for automotive applications, in: IEEE International Workshop on Rapid Systems Prototyping, 1998.

[24] W. Cesario, A. Baghdadi, L. Gauthier, D. Lyonnard, G. Nicolescu, Y. Paviot, S. Yoo, A. A. Jerraya, M. Diaz-nava, Component-based design approach for multicore SoCs, in: DAC'02, 2002.

[25] S. R. Thatté, Automated synthesis of interface adapters for reusable classes, in: Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages, 1994.

[26] D. M. Yellin, R. E. Strom, Protocol specifications and component adaptors, ACM Transactions on Programming Languages and Systems 19 (2) (1997) 292–333.

[27] R. H. Reussner, Automatic component protocol adaptation with the CoConut/J tool suite, Future Generation Computer Systems 19 (5) (2003) 627–639.