Component-based hardware-software co-design*

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
arato@iit.bme.hu, {Zoltan.Mann,Andras.Orban}@cs.bme.hu

Abstract. The unbelievable growth in the complexity of computer systems poses a difficult challenge on system design. To cope with these problems, new methodologies are needed that allow the reuse of existing designs in a hierarchical manner, and at the same time let the designer work on the highest possible abstraction level.
Such reusable building blocks are called components in the software world and IP (intellectual property) blocks in the hardware world. Based on the similarity between these two notions the authors propose a new system-level design methodology, called component-based hardware-software co-design, which allows rapid prototyping and functional simulation of complex hardware-software systems. Moreover, a tool is presented supporting the new design methodology and a case study is shown to demonstrate the applicability of the concepts.

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. With the wide spread of mobile systems and the advent of ubiquitous computing, size, heat dissipation and energy consumption [1] are also becoming crucial aspects for a wide range of computer systems, especially embedded systems.

To take into account all of these aspects in the design process is becoming next to impossible. According to the International Technology Roadmap for Semiconductors [2], the most crucially challenged branch of the computer industry is system design. The Roadmap clearly declares that Moore's law can hold on for the next decades only if innovative new ways of system design will be proposed to handle the growing complexity.

* This work have been supported by the European Union as part of the EASYCOMP project (IST-1999-14151) and by OTKA T 043329. This paper has been accepted for publication at the International Conference on Architecture of Computing Systems (ARCS) 2004.

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; they together determine the operation of the system. The differences between hardware and software and their interaction contribute significantly to the above-mentioned huge complexity of systems. On the other hand, the similarities between hardware and software design open many possibilities for their optimized, synergetic co-design. This is the motivation for hardware-software co-design (HSCD) [3].

To address the above problems, different, but in many ways similar solutions have been developed in the software and hardware world.

1.1 Solutions in the software world

Traditionally, the focus of software engineering has been on flexibility, code readability and modifiability, maintainability etc. This has led to the notions of separation of concerns, information hiding, decoupling, and object-orientation. In recent years though, as a result of the growing needs, the reuse of existing pieces of design or even code has received substantial attention. Examples of such efforts include design and analysis patterns, aspect-oriented programming, software product lines, and component-based software engineering [4].

Unfortunately, the definition of a *component* is not perfectly clear. There are several different *component models*, such as for instance the CORBA component model or the COM component model. Each of these component models define the notion of a component slightly differently. However, these definitions have much in common: a component is a piece of adaptable and reusable code, that has a well-defined functionality and a well-defined interface, and can be composed with other components to form an application. Components are often sold by third-party vendors, in which case we talk about COTS (commercial off-the-shelf) components.

Each component model defines a way for the components—which might be very different in programming language, platform or vendor—to interact with each other. The component models are also often supported by middleware, which provides services that are often needed—such as support for distribution, naming and trading service, transactions, persistence etc.—to the components. As a result, the middleware can provide *transparency* (location transparency, programming language transparency, platform transparency etc.), which facilitates the development of distributed component-based software systems enormously.

1.2 Solutions in the hardware world

Since the construction of hardware is much more costly and time-consuming than that of software, the idea of reusing existing units and creating the new applications out of the existing building blocks is definitely more adopted in the hardware world. This process has led from simple transistors to 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 largely adaptable. These building blocks are called IP (intellectual property) units [5–8]. They clearly resemble software components; however, IPs are even less standardized than software components. We do not know widely accepted component models such as CORBA or EJB in the hardware world.

Another consequence of the high cost of hardware production is that hardware must be carefully tested before it is actually synthesized. Therefore, testing solutions are more mature in the hardware world: *e.g.* design for testability (DFT) and built-in self test (BIST) are common features of hardware design. Moreover, it is common to use simulation of the real hardware for design and test purposes.

1.3 Convergence

The production costs of hardware units depend very much on the volume of the production. It is by orders of magnitude cheaper to use general-purpose, adaptable hardware elements which are produced in large volumes than special-purpose units. The general-purpose units (*e.g.* Field Programmable Gate Arrays or microprocessors) have to be *programmed* to perform the given task. Therefore, when using general-purpose hardware units to solve a given problem, one actually uses software. Conversely, when creating a software solution, one actually uses general-purpose hardware. Consequently, the boundary between adaptable hardware units and software is not very sharp.

As already mentioned, hardware is usually simulated from the early phases of the design process. This means that its functionality is first implemented by software. Moreover, there are now tools, for instance the PICO (Program In, Chip Out [9]) system, that can transform software to hardware.

Motivated by the above facts this paper introduces a new system-level design methodology which handles both software and hardware units at a high abstraction level and propagates the concept of reuse by assembling the system from hardware and software building blocks. Note that it is not the intention of this paper to address each system-level synthesis problem emerging during HSCD, our goal is only to highlight the concept of a new system-design approach and to deal with problems special to the new methodology.

The paper is organized as follows. Section 2 introduces the proposed new methodology and some related problems. The tool supporting the new concepts is demonstrated in Section 3 and a case study is presented in Section 4. Finally, Section 5 concludes the paper.

2 A new HSCD 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

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 [10, 11] 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 verified at an early stage of the design process. CBHSCD supports hierarchical design: the generalized notion of components makes it possible to reuse complex hardware-software systems as components in later designs. (See also Section 2.6.)

The main steps of CBHSCD are shown in Fig 1. In the following each subtask is detailed except the issues related to the *synthesis* which are beyond the scope of CBHSCD.

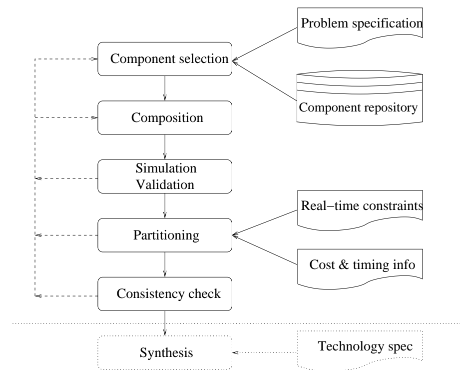

Fig. 1. The process of CBHSCD

2.1 Component selection

The process starts by selecting the appropriate components[2] from a component repository based on the problem specification (Of course the selection of an appropriate component is an individual challenge [5, 12], but it is beyond the scope of this paper to address this problem). 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

[1] www.easycomp.org
[2] We use the term component to refer to a reusable building block, which might be hardware, software, or the combination of both in hierarchical HSCD.

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 (see Section 2.4).

2.2 Composition

After the components are selected, they are composed to form a prototype system. 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 components is based on remote method calls between components supported by the underlying middleware.

To handle all components—including the hardware components—uniformly, a wrapper should be designed around the device driver communicating directly with the hardware. This wrapper has the task to produce a software-like interface for the hardware component, to delegate the calls and the parameters to the device driver and to trigger an event when a hardware interrupt occurs. 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 can also participate in remote method calls both as initiator or as acceptor.

Composition is supported by a visual tool that provides an intuitive graphical user interface (GUI) 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. The simple composition also allows for easy rapid prototyping of complex hardware-software systems.

2.3 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 even possible to simulate parts of the system, so that problems can be detected before the whole system is composed.

Important to note that components are living and 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[3]. If the 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 need software simulation. If a software equivalent of the hardware component is available—*e.g.* if the hardware is synthesized from a software description, which is often the case, or if the hardware performs a well-known algorithm, which is also implemented in software—then this software equivalent can be used for simulation. Even if a complete software equivalent is not available, there might be an at least interface-equivalent software, *e.g.* if the IP vendor provides a C code to specify the interface of its product. Also, if the description of the hardware is available in a hardware description language such as VHDL, a commercial hardware simulator can be used. However, we can assume that sooner or later all IP vendors will provide some kind of formal description of their products which is suitable for functional simulation [5].

Related work includes the embedded code deployment and simulation possibilities of Matlab (`http://www.mathworks.com`) and the Ptolemy project (`http://ptolemy.eecs.berkeley.edu/`).

2.4 Partitioning

After the designer is convinced that the system is functionally correct, the system has to be partitioned, *i.e.* the components have to be mapped to software and hardware. (There can be components which only exist in hardware or in software, so that their mapping is trivial.) This is an important optimization problem, in which the optimal trade-off between cost and performance has to be found. 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. Partitioning is automated based on a declarative requirements specification. We defined a graph-theoretic model for the partitioning problem [13, 14] and there are other partitioning algorithms in the literature, see *e.g.* [15–17] and references therein. The partitioning algorithm has to take 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

[3] Before partitioning it is not even known of each component whether to be realized in software or hardware.

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, which are largely the hardware costs. Similarly, when costs are limited, the running time is minimized, which is essentially the running time of the software plus the communication overhead. 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, for which there might be several possibilities with different cost and performance. Concerning the *running times*, a worst case (if hard real-time constraints are specified) or average case running time is either provided with the component or extracted by some profiling technique. An independent research field deals with the measurement or estimation of these values, see *e.g.* [18, 19]. The time and cost *constraints* must be specified explicitly by the designer via use-cases (see Section 3 for more details).

2.5 Consistency

One of the main motivations of CBHSCD is to raise the abstraction level high enough where the boundary of hardware and software vanishes. Since components interacting with each other are not aware of the context of the other (only the interface is known), hence the change of implementation should be transparent to others. It implies two consistency problems special to partitioning in CBHSCD.

Interface consistency. The components subject to partitioning are available also in software and hardware. There is an interface associated to all these pairs, which describes the necessary methods and attributes the implementations should provide in order to allow transparent change between them. It must be checked whether both implementations realize this interface. (For related work see *e.g.* [20].)

State consistency. The prototype system can be 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. (In the case of a long-lasting simulation it may not be feasible to restart the simulation after each swap.) This is not straightforward, because the components are handled as black-box, and it is not possible to access all the state-variables from outside. A number of component models explicitly forbid stateful components to avoid these problems. Our proposed solution to achieve the desired state is to repeat all the method calls on the new implementation that has affected the state of the current implementation since the last swap. (See Section 3 for more details.)

At the end, the system is synthesized, which involves generation of glue code and adapters for the real interconnection of the system, and also the generation of a test environment and test vectors for real-world testing. However, our main objective was to improve the design process, which is in our opinion the real bottleneck, so that the last phase, which involves implementation steps, is beyond the scope of this paper.

2.6 Hierarchical design

Hierarchical design [21, 22] is an integral part of CBHSCD. It helps in coping with design complexity using a divide-and-conquer approach, and also in enhancing testability. Namely, the system can be composed of well-tested components, and only the composition itself has to be tested, which compresses the test space enormously.

Hierarchical design in CBHSCD can be interpreted either in a bottom-up or a top-down fashion. Bottom-up hierarchical design means that a system that has been composed of hardware, software and mixed components[4] using CBHSCD methodology can later be used as a component on its own for building even more complex systems. Top-down hierarchical design means that a complex problem is divided into sub-problems, and this decomposition is refined until we get manageable pieces. The identified components can then be realized either based on existing components using CBHSCD methodology or using a traditional methodology if the component has to be implemented from scratch.

As a simple example of such a hierarchical design, consider a computation-intensive image-processing application, which consists of a set of algorithms. In order to guarantee some time constraints, one of the algorithms has to be performed by a very fast component. So the resulting system might consist of a general-purpose computer and an attached acceleration board. However, the acceleration board itself might include both non-programmable accelerator (NPA) logic and a very long instruction word processor (VLIW) processor [9], which performs the less performance-critical operations of the algorithm in software, as the result of a similar design step.

2.7 Communication

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. 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. Consequently,

[4] Clearly, pure hardware and pure software components are just the two extremes of the general component notion. Generally, components can realize different cost/performance trade-offs ranging from the cheapest but slowest solution (pure software) to the most expensive but fastest solution (pure hardware).

the communication between hardware and software becomes very much like remote procedure calls (RPC) in distributed systems. The resulting architecture is shown in Fig 2.

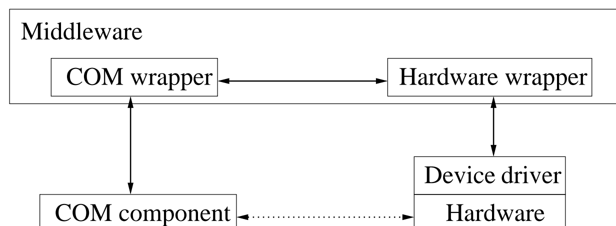

Fig. 2. Communication between a COTS software component (COM component in this example) and a hardware unit. The dotted line indicates the virtual communication, the full line the real communication.

The drawback of this approach is the large communication overhead introduced by the wrappers and the middleware layer in general. However, 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 can be reduced. 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.* [23, 7].

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 [24].

CWB is a graphical design tool implemented in Java for the easy composition of applications from COTS software components. The main contribution of CWB is the support for composition of components from different component models, like COM, CORBA, EJB etc. To achieve this, CWB uses a generic component model called Vienna Composition Framework (VCF) which handles all existing component models similarly. This generic model offers a very 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 set of events the component can emit.
Lifecycle If a component has this feature, then it can be created and destroyed, activate or deactivate.
GUI The graphical interface of the component.

Each component model is implemented as a *plug-in* in the CWB (see Fig 3). The plug-in class only provides information about the features the component can provide, the real functionality is hidden in the classes implementing the features. As the name suggests, new plug-ins can be added to the CWB, that is, new component models can be implemented. To do that, a new plug-in class and a class representing the required features have to be implemented. These classes realize the wrapper between the general component model of VCF and the specific component model.

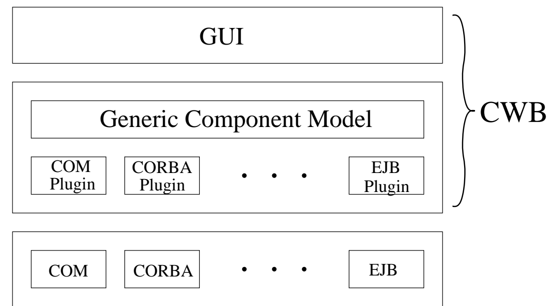

Fig. 3. The architecture of the CWB.

For the communication between the components, CWB offers multiple communication styles. One of the most important communication styles supported by CWB is the event-to-method communication, *i.e.* a component triggers an event which induces a method 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. New communication styles can also be added to the CWB.

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 both software and hardware and so called *partitionable* components from a repository. The latter identifies two implementations for the same behavior. These components can originate from different vendors and different component models including hardware and software. 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 some kind of simulation is used. The designer can choose between different simulation levels, as already discussed.

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.

To identify the features a hardware component can provide [5], 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; the wizard parses the component's features and allows the connection according to the selected communication style. Due to the wrappers, hardware components act the same way as software ones, 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. Each use-case corresponds to a specific usage of the system, typically initiated by an entity outside the system. 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 measurement of running time and communication cost parameters is at an initial stage in our tool; currently we expect that this data is explicitly given by the designer.

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_SIDE_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 this method call the last one should be repeated only. An example is setting a property to a value.

`SIDE_EFFECT`: the corresponding 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.

The system logs every method call and property change since the last implementation swap. If all these belong to the first three category, 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 method calls to repeat. The designer is supported by a detailed log in this decision.

4 Case study

In this section the CBHSCD methodology will be step-by-step demonstrated on a small example application. In this example the frequency of an unknown source signal has to be measured. This task might appear in several real-world applications like mobile phone technology, hence this system can be used as a building block in later designs. The architecture of the example can be seen in Fig 4. The frequency measurer (FM) measures the signal of the generator and sends the measured value periodically to the PC through the serial port. The PC on the one hand displays the current frequency value and plots a graph on the alteration of the value, and, on the other hand, controls the measurer through start and stop signals. There are two implementations available for the FM: the first one is a programmable PIC 16F876 microcontroller regarded as software implementation and an FPGA on a XILINX VIRTEX II XC2V1000 card as the 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 (to take a sample lasts 40μs). The FPGA on the other hand can take a sample in 50ns, thus it can measure up to 20MHz without any problem.

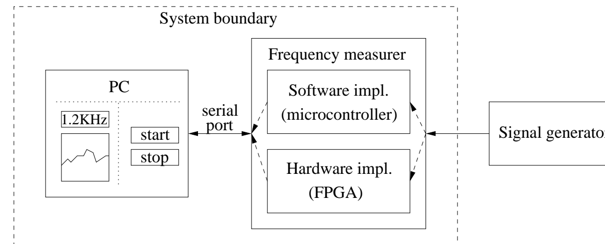

Fig. 4. The architecture of the example application

There are five components in this example: two JavaBeans buttons (start and stop), a `TextField` and a chart component for display and the FM declared as a partitionable component with the two implementations detailed above[5]. Both implementations belong to the component model whose device driver is able to communicate with the devices through serial port. For consistency purposes the interface on Fig 5 is provided with the component. The device driver is wrapped by a CWB wrapper providing a software-like interface. The tool checks whether the interfaces of the wrappers match the requirements.

```
package frequency;

public interface FrequencyEstimatorInterface {

  SIDE_EFFECT public void start();
  SIDE_EFFECT public void stop();
  NO_SIDE_EFFECT  public void takeOneSample();
  NO_SIDE_EFFECT public String getMeasuredFrequencyString();
  NO_SIDE_EFFECT public Integer getMeasuredFrequency();
  REPEAT_AT_REPARTITION_ONCE public void setCountEveryEdge(boolean b);
  NO_SIDE_EFFECT public boolean getCountEveryEdge();
}
```

Fig. 5. Part of the required interface with state consistency attributes of the partitionable frequency measurer (FM) component

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

In the composition phase the start and the stop button should be mapped with the aid of the mentioned wizard to the start and stop method of the FM, respectively. The FM sends an interrupt whenever a new measured value is arrived. This interrupt appears as an event in the CWB-X; this event 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 PC will display the measured values.

The task of partitioning will be to decide which implementation to use according to the time requirements of the system. The designer defines a use-case which declares a time-limit for the `takeOneSample` function of the FM. In this simple case the optimal partition is trivial[6]: if the time-limit is under $40\mu s$, the FPGA should be used, otherwise the microcontroller (here we assume, that to program the microcontroller is cheaper than to produce the FPGA). The partitioner finds this solution and changes the implementation if necessarily. The new implementation will be transformed to the same state as the current one according to the steps detailed in Section 3.

5 Conclusion

In this paper, we have described a new methodology for hardware-software co-design, which emphasizes reuse, a high abstraction level, design automation, and hierarchical design. The new methodology, called component-based hardware-software co-design (CBHSCD), unifies component-based software engineering and IP-based hardware engineering practices. It supports rapid prototyping of complex systems consisting of both hardware and software, and helps in the design of embedded and real-time systems.

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 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. H. Lekatsas, W. Wolf, and J. Henkel. Arithmetic coding for low power embedded system design. In *Data Compression Conference*, pages 430–439, 2000.
2. A. Allan, D. Edenfeld, W. H. Joyner Jr., A. B. Kahng, M. Rodgers, and Y. Zorian. 2001 Technology Roadmap for Semiconductors. *IEEE Computer*, 35(1), 2002.

[6] Generally the partitioning problem is $\mathcal{NP}$-hard.

3. R. Niemann. *Hardware/Software Co-Design for Data Flow Dominated Embedded Systems*. Kluwer Academic Publishers, 1998.
4. George T. Heineman and William T. Councill. *Component Based Software Engineering: Putting the Pieces Together*. Addison-Wesley, 2001.
5. G. Martin, R. Seepold, T. Zhang, L. Benini, and G. De Micheli. Component selection and matching for IP-based design. In *Proceedings of the DATE 2001 on Design, automation and test in Europe*. IEEE Press, 2001.
6. Ph. Coussy, A. Baganne, and E. Martin. A design methodology for integrating ip into soc systems. In *Conférence Internationale IEEE CICC*, 2002.
7. P. Chou, R. Ortega, K. Hines, K. Partridge, and G. Borriello. Ipchinook: an integrated ip-based design framework for distributed embedded systems. In *Design Automation Conference*, pages 44–49, 1999.
8. F. Pogodalla, R. Hersemeule, and P. Coulomb. Fast protoyping: a system design flow for fast design, prototyping and efficient IP reuse. In *CODES*, 1999.
9. V. Kathail, S. Aditya, R. Schreiber, B. R. Rau, D. C. Cronquist, and M. Sivaraman. PICO: automatically designing custom computers. *IEEE Computer*, 2002.
10. G. Spivey, S. S. Bhattacharyya, and Kazuo Nakajima. Logic Foundry: A rapid prototyping tool for FPGA-based DSP systems. Technical report, Department of Computer Science, University of Maryland, 2002.
11. Klaus Buchenrieder. Embedded system prototyping. In *Tenth IEEE International Workshop on Rapid System Prototyping*, 1999.
12. P. Roop and A. Sowmya. Automatic component matching using forced simulation. In *13th International Conference on VLSI Design*. IEEE Press, 2000.
13. Z. Á. Mann and A. Orbán. Optimization problems in system-level synthesis. 3rd Hungarian-Japanese Symp. on Discrete Mathematics and Its Applications, 2003.
14. P. Arató, S. Juhász, Z. Á. Mann, A. Orbán, and D. Papp. Hardware/software partitioning in embedded system design. In *Proceedings of the IEEE International Symposium on Intelligent Signal Processing*, 2003.
15. N. N. Binh, M. Imai, A. Shiomi, and N. Hikichi. A hardware/software partitioning algorithm for designing pipelined ASIPs with least gate counts. In *Proceedings of the 33rd Design Automation Conference*, 1996.
16. B. Mei, P. Schaumont, and S. Vernalde. A hardware/software partitioning and scheduling algorithm for dynamically reconfigurable embedded systems. In *Proceedings of ProRISC*, 2000.
17. T. F. Abdelzaher and K. G. Shin. Period-based load partitioning and assignment for large real-time applications. *IEEE Transactions on Computers*, 49(1), 2000.
18. X. Hu, T. Zhou, and E. Sha. Estimating probabilistic timing performance for real-time embedded systems. *IEEE Transactions on VLSI Systems*, 9(6), 2001.
19. S. L. Graham, P. B. Kessler, and M. K. McKusick. An execution profiler for modular programs. *Software Practice & Experience*, 13:671–685, 1983.
20. A. Speck, E. Pulvermüller, M. Jerger, and B. Franczyk. Component composition validation. *International Journal of Applied Mathematics and Computer Science*, pages 581–589, 2002.
21. G. Quan, X. Hu, and G. Greenwood. Preference-driven hierarchical hardware/software partitioning. In *Proceedings of the IEEE/ACM International Conference on Computer Design*, 1999.
22. R. P. Dick and N. K. Jha. MOGAC: A multiobjective genetic algorithm for hardware-software co-synthesis of hierarchical heterogeneous distributed embedded systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 17(10):920–935, 1998.

23. A. Basu, R. Mitra, and P. Marwedel. Interface synthesis for embedded applications in a co-design environment. In *11th IEEE International conference on VLSI design*, pages 85–90, 1998.
24. Johann Oberleitner and Thomas Gschwind. Composing distributed components with the component workbench. In *Proceedings of the Software Engineering and Middleware Workshop (SEM2002)*. Springer Verlag, 2002.