

A Framework for System-Level Partitioning of Object-Oriented Specifications

Carsten Schulz-Key Tommy Kuhn Wolfgang Rosenstiel

Wilhelm-Schickard-Institute
for Computer Science
University of Tuebingen, Germany
{schulzke,kuhn,rosen}@informatik.uni-tuebingen.de

Abstract — Object-oriented descriptions are gaining more and more importance in the high-level specification of hardware/software systems. Hiding the complexity from the developer is one of the key tasks in order to master the complexity of todays systems. With the high grade of abstraction necessary on the system level the automatic partitioning of a system is a difficult problem. In this paper we therefore present a strategy that allows the designer to guide the partitioning with object-oriented techniques. We also developed an object-oriented interface for the partition overlapping communication of objects. The usability of our approach is shown with the example of a JPEG codec.

I. INTRODUCTION

The ever increasing complexity of modern hardware/software systems imposes a great challenge on the developer. It is almost impossible to develop a system while adhering to the close time lines that are dictated by short product cycles.

This problem can be solved by making the specification on higher and higher levels of abstraction. State of the art are behavioral descriptions on the algorithmic level that are very well suited for monolithic designs that fit on a single ASIC or FPGA. The next higher level of abstraction –the system-level– is subject to ongoing research in the academic world. It is used for the specification of heterogeneous systems that consist of multiple components that can be instantiated on different ASICs, FPGAs, DSPs, or microcontrollers. It offers a high degree of flexibility in the way that each component can be located in the domain that it fits in best: Critical parts being bound to real-time conditions or requiring low power consumption can be realized in a custom IC or a dedicated FPGA; computation and data intensive parts can be handled by digital signal processors. The rest of the system, including those parts that use dynamic data structures, or that are likely to be changed in the next revision of the firmware, can be executed on a cheap microcontroller. Because of the high abstraction level there is virtually no difference between the description of the software and the hardware part, and it is possible to describe the whole system within a single specification.

In order to master the complexity object-oriented paradigms can be applied which have shown their usefulness in the development of large software packages. In the last years synthesis from object-oriented specifications that are made in languages

well known in the software domain like C++ [15] and Java [18, 9] has become possible. They give the benefit that they produce executable specifications that can be easily and efficiently simulated. These languages are supplemented by new languages like 'e' [8] that add constructs for verification.

If a whole system is described within a single specification, the problem arises how the different parts or partitions are determined. This problem is not well addressed on the system-level yet. On this high level of abstraction it is important to hide the underlying complexity from the developer and to provide a consistent object-oriented view. In this paper we therefore present a strategy how the partitioning can be integrated in an object-oriented framework for the synthesis of system specifications. The communication between objects lying in different partitions is modeled with an interface concept on object basis.

The rest of the paper is structured as follows: First we discuss the advantages and weaknesses of existing partitioning approaches in section II. In section III we will give an overview of our object-oriented partitioning strategy and describe some of the restrictions that have to be made. Afterwards we will show the usability of our approach with the example of a JPEG codec. We conclude the paper with a summary in section V.

II. PREVIOUS WORK

The partitioning of circuits is an old and well researched topic. Some of the older publications in this area date back to 1984 [7]. The granularity at that time was very fine and the objects that were partitioned were net-lists — hence this approach was appropriate for the *logic level*.

Later the abstraction level was successively raised to the *algorithmic level* and the designs were split up into separate hardware and software parts. Today, in the age of hardware/software co-design, the granularity of the partitions ranges from instructions [5] over loops and basic blocks [6, 10] to entire functions [4].

Recently the granularity was raised further; in [11, 12] Noguera and Badia consider whole objects for partitioning. This is a requirement for the partitioning of object-oriented specified systems. However, their automatic approach works on class basis and does not allow changes by the designer. Therefore it is not possible to have several instances of the

same class in different partitions. Furthermore they neglect the fact that certain objects that should be instantiated in hardware might be too large. Vahid [17] states that many applications consist of few large processes that benefit much if they are split up into several smaller processes. We think that this statement also holds in the object-oriented case. Another point that prohibits the instantiation of an *entire* object in hardware is the presence of dynamic structures, which could be avoided with object decomposition.

In our approach we are avoiding these weaknesses:

III. OBJECT-ORIENTED PARTITIONING

Traditional partitioning schemes are not well suited for the high level of abstraction inherent in *system-level* specifications. It is very difficult to obtain *accurate* estimations for the constraints that would influence partition decisions; they only become available much later in the synthesis process. The partitioning of a system specification can therefore only be an interactive, iterative process, where the designer validates the suggestions produced by algorithms. It is vital that the designer can influence the partitioning early in the specification phase and explicitly mark certain parts of the system for a later instantiation in a certain partition.

Unfortunately this adds complexity to the system, which we want to avoid. Therefore we have to present the partitioning to the designer with object-oriented methods. We must take care that the object-oriented view is maintained at all times while the designer is working on the specification.

In the next paragraphs we show how the partitioning can be integrated in an object-oriented synthesis framework. We start with the system modeling and specification, describe the methods a designer can use for marking objects for certain partitions and present the interface for partition overlapping communication. We also show a strategy for optimizing the partitions by avoiding some of the drawbacks inherent in the object-oriented synthesis.

A. System Design Flow

In our framework we use CASE¹ tools like *Together* [16] or *RationalRose* [13] for the entry of the system model. The class hierarchy, collaboration diagrams and sequence diagrams, etc. are entered in UML notation [1] and the destination source code for the specification is automatically generated in the selected language.

The partitioning strategy itself is pretty much language independent, as long as the language is object-oriented. The synthesis part of our framework can currently process specifications in Java and 'e'; support for C++ with SystemC libraries is under development. Since the CASE tools can not export 'e' source code yet, we use Java for the examples throughout this paper.

Once the source code is generated, our interface generation tool which we describe in paragraph III.C generates the

necessary classes for the partition overlapping communication. Then the designer can complete the specification by adding the code that is not generated by the CASE tools. The design flow in the framework continues with the object-oriented analysis that generates the input for commercial high-level synthesis tools for the hardware part and the source code for the software part.

In the next paragraph we present the partitioning specific classes a designer can use for the system specification within our framework.

B. OASE Specification and Partition Classes

The determination whether an object of a class should be instantiated in the hardware or in the software partition is expressed by inheritance. We provide a set of interface classes² as shown in the hierarchy in Fig. 1.

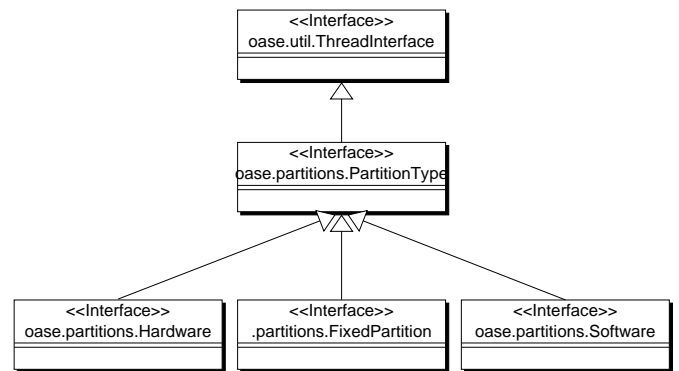


Fig. 1. UML class diagram of the interface hierarchy used for determining a certain partition type.

The following code fragment specifies a class whose objects will be created in hardware:

```

import oase.partition.*;

class Exmpl implements Hardware {
    ...
}
  
```

With this concept, interactively changing the membership of a class in a partition is just a matter of changing an inheritance arrow in the UML class diagram. This also works for the opposite direction: The changes to the partitions made by an automated scheme are reported back to the designer as changes in the inheritance hierarchy.

The *Software* interface is used analogously to the *Hardware* interface for the specification of objects that are going to be instantiated inside the software partition. The *FixedPartition* interface is used in conjunction with a *Hardware* or *Software* interface to express that an object

²The interface concept is Java specific, but the same functionality can also be expressed by multiple inheritance in C++ and 'e'. For clarity reasons we only show the Java part.

¹Computer Aided Software Engineering

has to be definitively instantiated inside a specific partition, even if the analysis process would later come to a different conclusion.

It is not necessary to mark every single object in a partition by deriving it from a partition class. All objects that are instantiated by a hardware object will be recognized during the analysis phase and automatically be associated to the appropriate partition — the same holds for the software objects. Together these objects form a component that is modeled in analogy to the JavaBeans component model [3]. Every component is running in its own thread — which is denoted by the `ThreadInterface` on top of the hierarchy in Fig. 1— and is mapped to a VHDL or Verilog process during synthesis. For details we refer to [9, 8].

It has some advantages and is good programming practice to insert another level of indirection: Instead of directly implementing the `Hardware` interface, it is better to implement a different interface which in turn extends the `Hardware` interface:

```
interface ExmplInterface extends Hardware
{
    ...
}

class Exmpl implements ExmplInterface {
    ...
}
```

With the extra level of abstraction the manipulation of the partition membership is very easy, because the number of dependencies in other objects that have aggregated the object is kept at a minimum. Furthermore with this it is possible to have instances of a given class in the hardware and the software partition at the same time, as shown in Fig. 2:

Both classes `A_HW` and `A_SW` are derived from the same class `A`, but implement different interfaces. Therefore it is possible to specify different partitions for them, while no extra code has to be written or copied. Class `A` remains the single location of all relevant code that is written by the designer.

C. Communication Model

The implementation of the interfaces for the communication between partitions is often a source of subtle errors. Therefore it is necessary to take this task from the designer and provide tools for the automatic generation of the interfaces. In our case we also have to encapsulate the interfaces in such a way that the object-oriented view can be maintained.

The interfaces have to fulfill two mayor tasks:

1. They have to provide methods for the exchange of data between the partitions, and
2. they have to provide ways for the synchronization of the threads in the different partitions.

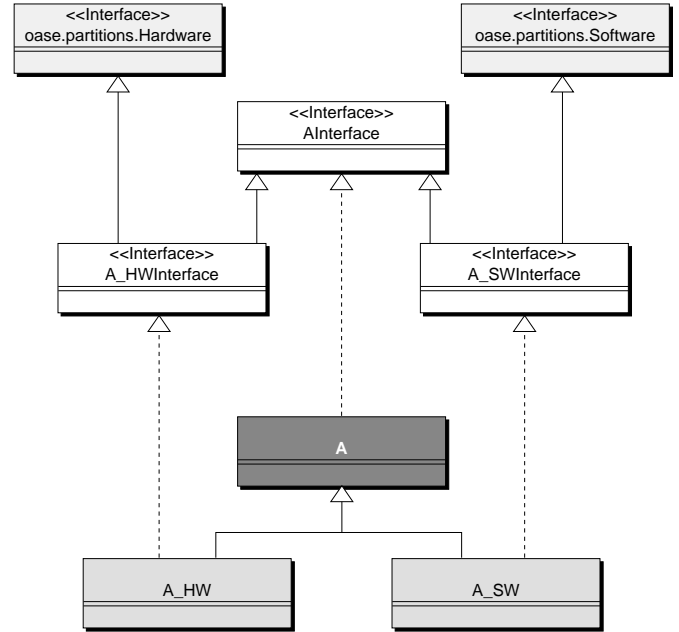


Fig. 2. Instantiating objects of the same class in different partitions.

For the realization of our communication interfaces we chose a stub/skeleton technique which is a standard in other object-oriented communication packages like Java RMI [14]. This way the designer is not confronted with unknown concepts.

We illustrate this technique with an example in Fig 3: Object `A` resides in the hardware partition and object `B` remains on the software side. Previous to the partitioning object `A` was directly associated in object `B` by the following code:

```
AInterface a = new A();
```

During the partitioning step object `A` becomes unavailable in the software partition and the communication has to be done via a stub/skeleton pair. Therefore all requests to it are now directed to it via a skeleton object `AHardwareSkeleton`:

```
AInterface a = new AHardwareSkeleton();
```

The skeleton object holds a reference to the stub object `AHardwareStub` and delegates all method calls to it after serializing them. This is done by pushing the op-code and the arguments for each method call on a special stack³ of the stub object.

The stub object mainly consists of a `while` loop that gathers and re-assembles the queued requests from the stack and performs the actual method calls on the hardware implementation of object `A`. The method calls on the stub object are implemented using a blocking semantic. Therefore they can serve as synchronization points between the two threads/processes in the different partitions.

³The stack is only necessary for simulation purposes — in the synthesized hardware the arguments are directly sent to the corresponding ports.

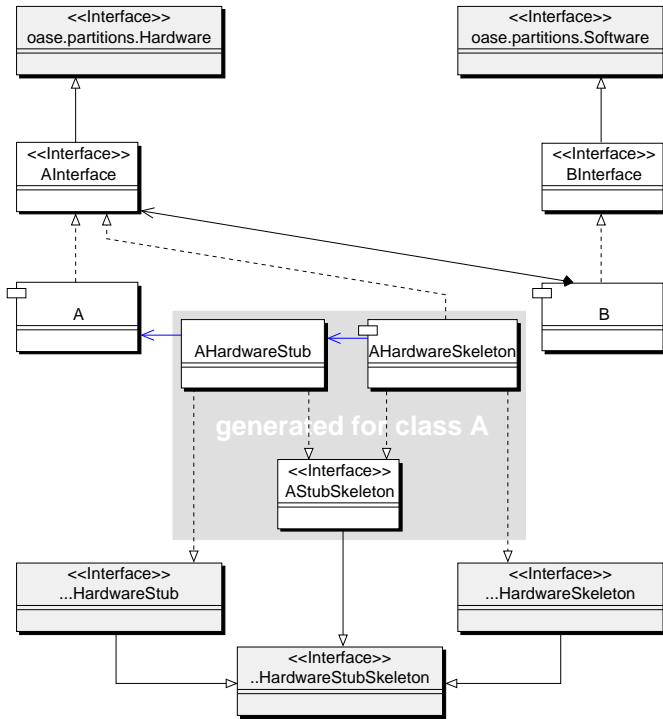


Fig. 3. Class diagram of the generated stub/skeleton pair and its integration in the specification context.

Currently we only support primitive data types like integers, floats and special objects like bit-vectors—that are defined within our synthesis package—as type of the arguments and return value of method calls. For the future we are planning to relax this restriction by also accepting a `this` reference to the calling object.

Direct access to public attributes of an object residing in a different partition is not possible with our interface concept. Read or write access to those variables has to be done via *setter/getter* methods, instead. Since this is considered good object-oriented programming practice the CASE tool *Together* automatically generates the *setter/getter* methods for each attribute that is entered.

For the automatic generation of the stub/skeleton pair we developed the interface generator *IFGen*. Our generator reads in the source code that was written by the designer and extracts all necessary information. Then the source code for the stub and skeleton of the interface is generated and all references to the partitioned object are updated accordingly. After this the new classes are imported into the CASE tool and the designer can continue with the specification. The specification can be compiled at any time and its functional correctness can be established by executing and thereby simulating it.

D. Object Decomposition

In some cases it is not possible to instantiate an object inside a hardware partition, even if the designer explicitly marks it with the `FixedPartition` tag:

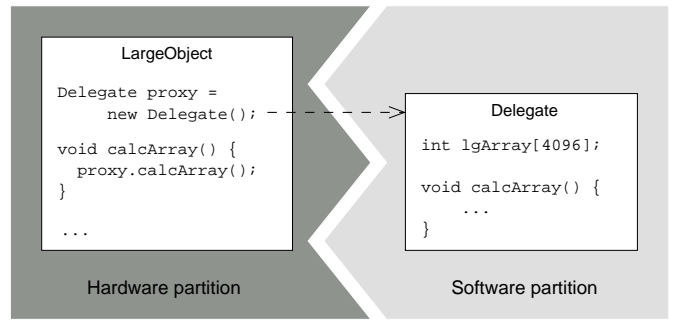


Fig. 4. Delegation of parts of an object into a different object that is located in the software partition.

- The net-list that is generated during synthesis is too large to satisfy given area constraints.
- The object contains dynamic structures. The synthesis can only be made if all instantiations of objects can be determined at compile-time. Therefore it is not possible to synthesize objects that instantiate other objects in loops with an unknown number of iterations.
- Some other constraints that are required for the analysis and synthesis of the specification are violated. For more information on this subject we refer to [9].

Our solution to this set of problems is object decomposition based on the application of the *delegation* pattern [2]. We create a new object and move all conflicting parts into it:

Suppose that the object `LargeObject` in the example in Fig. 4 can not be synthesized because its net-list does not fit onto any FPGA. We then create a new object `Delegate` and move some part of the object into it, in this case the attribute `lgArray[]` and the implementation of the method `calcArray()`. The original `calcArray()` method is substituted by a call to the method of the delegate object. Similarly, all references to the `lgArray[]` attribute are substituted by calls to *setter/getter* methods on the delegate object (not shown). If the delegate object implements the `Software-`

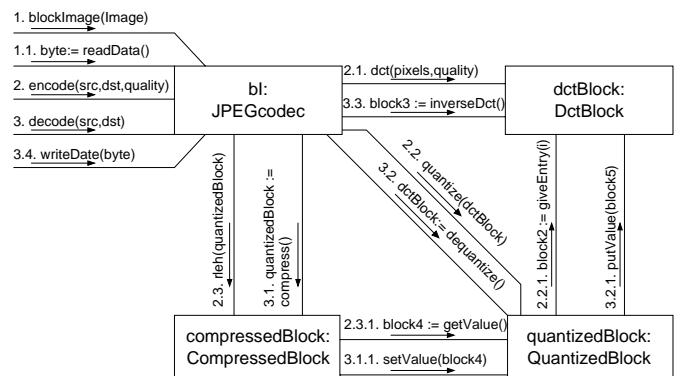


Fig. 5. JPEG collaboration diagram.

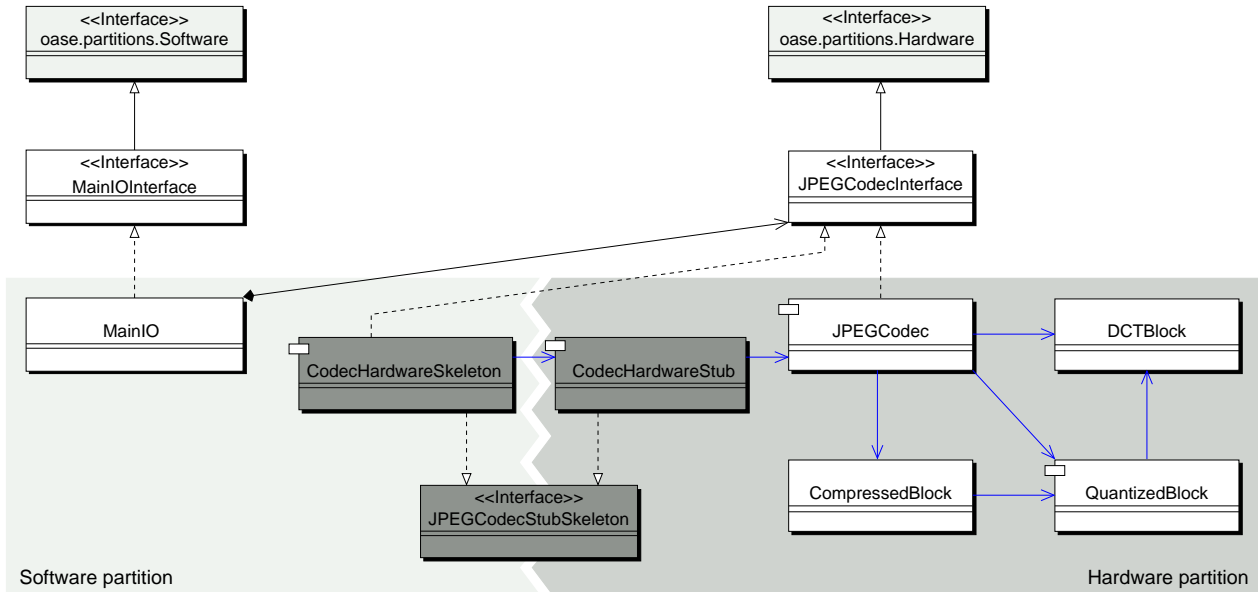


Fig. 6. Partitioning of the JPEG codec.

Interface it will be located in the software-partition. The HW/SW-interface generator will recognize this and automatically generate the necessary stub/skeleton pair for the partition overlapping communication between the objects.

With this strategy we increase the flexibility of the object segmentation and the designer can possibly generate optimized partitions containing smaller objects.

IV. EXAMPLE

In this paragraph we illustrate the application of our partitioning scheme with the example of a JPEG codec. The collaboration diagram for the classes of the codec are shown in Fig. 5.

All classes are strongly connected with each other, with the class `JPEGCodec` being the initiator of most actions. The amount of data that is going to be exchanged for each block is quite high. Furthermore the calculations performed in the `DCTBlock` and `QuantizedBlock` objects are relatively time consuming. Therefore the ideal solution would be to realize all the classes in hardware. This is, however, not possible, since in class `JPEGCodec` File-I/O is performed.

This problem can be solved by object decomposition: We move all I/O-related parts into a new object `MainIO`; as a result we get a different `JPEGCodec` implementation that can be instantiated in the hardware partition. Therefore we derive it from the `Hardware` class and derive the `MainIO` class from `Software`. In the next step the hardware stub/skeleton pair is automatically generated and the partitioned system is ready for simulation and synthesis. The result of the partitioning is shown in Fig. 6.

V. CONCLUSIONS

The synthesis of object-oriented specifications on the *algorithmic level* is almost a standard today. But in order to be able to raise the level of abstraction further suitable concepts for the object-oriented encapsulation of the partitioning are missing. We therefore introduce a partitioning scheme for *system-level* descriptions that does not show the drawbacks of existing approaches. It fulfills the difficult task of giving the designer the ability to control the partitioning process while not unnecessarily increasing the complexity. The partitioning is presented with object-oriented methods to the designer who can make partitioning decisions by changing the inheritance of classes. This is important because partitioning on this high level of abstraction has to be an iterative process. An automated approach can hardly produce satisfying results because the data needed for the partitioning decisions are not available on this level. With our object decomposition scheme the designer is given the ability to reduce the granularity of objects, thereby facilitating the generation of optimized partitions.

The partition overlapping communication is modeled with a stub/skeleton concept for which all necessary classes are generated automatically.

ACKNOWLEDGMENTS

This work is funded by DFG project No. RO 1030/8-1.

REFERENCES

- [1] B. P. Douglass. *Doing Hard Time: Developing Real-Time Systems with UML, Objects, Frameworks, and Patterns*. Addison-Wesley, Reading, MA, 1999.

- [2] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of reusable Object-Oriented Software*, chapter 1.6. Addison-Wesley, Reading, MA, 1994.
- [3] G. Hamilton (Ed). JavaBeans API specification. Technical report, Sun Microsystems, Inc., 1997.
- [4] W. Hardt and W. Rosenstiel. Prototyping of tightly coupled hardware/software-systems. In *Design Automation For Embedded Systems*, volume 2, pages 283–318. Kluwer Academic Publishers, 1997.
- [5] J. Henkel. A low power hardware/software partitioning approach for core-based embedded systems. In *Proceedings of the Design Automation Conference (DAC'1999)*, pages 122–127, 1999.
- [6] J. Henkel and R. Ernst. A hardware/software partitioner using a dynamically determined granularity. In *Proceedings of the Design Automation Conference (DAC'1997)*, pages 691–696, 1997.
- [7] B. Krishnamurthy. An improved min-cut algorithm for partitioning VLSI networks. *IEEE Transactions on Computers*, C-33:438–446, 1984.
- [8] T. Kuhn, T. Oppold, M. Winterholer, W. Rosenstiel, M. Edwards, and Y. Kashai. A framework for object oriented hardware specification, verification, and synthesis. In *Proceedings of the Design Automation Conference (DAC'2001)*, 2001.
- [9] T. Kuhn, C. Schulz-Key, and W. Rosenstiel. Object oriented hardware specification with Java. In *Proceedings of the ninth workshop on Synthesis and System Integration of Mixed Technologies (SASIMI'2000)*, Kyoto, Japan, 2000.
- [10] Y. Li, T. Callahan, E. Darnell, R. E. Harr, U. Kurkure, and J. Stockwood. Hardware-software co-design of embedded reconfigurable architectures. In *Proceedings of the Design Automation Conference (DAC'2000)*, 2000.
- [11] J. Noguera and R. Badia. An object-oriented HW/SW partitioning algorithm for dynamically reconfigurable architectures. Technical report, Universitat Politècnica de Catalunya, 2000.
- [12] J. Noguera and R. Badia. A HW/SW partitioning algorithm for dynamically reconfigurable architectures. In *Proceedings of the Design, Automation and Test in Europe Conference (DATE'2001)*, 2001.
- [13] Rational Software Corporation. <http://www.rational.com/products/rose/index.jsp>.
- [14] Sun. Java remote method invocation specification. Technical report, Sun Microsystems, Inc., 1999.
- [15] Synopsys, Inc. *SystemC Reference Manual Release 1.2*, 2001.
- [16] TogetherSoft. <http://www.togethersoft.com/us/products/index.html>.
- [17] F. Vahid. A three-step approach to the functional partitioning of large behavioral processes. In *Proceedings of the International Symposium on System Synthesis (ISSS'1998)*, 1998.
- [18] J. S. Young, J. MacDonald, M. Shilman, P. H. Tabbara, and A. R. Newton. Design and specification of embedded systems in Java using successive formal refinement. In *Proceedings of the Design Automation Conference (DAC'1998)*, pages 70–75, 1998.