VLSI DESIGN
1995, Vol. 3, Nos. 3-4, pp. 333-345
Reprints available directly from the publisher
Photocopying permitted by license only

PARTIF: Interactive System-level Partitioning

TAREK BEN ISMAIL, KEVIN O'BRIEN and AHMED JERRAYA
System-level Synthesis Group, TIMA/INPG, 46 Av. Félix Viallet, 38031 Grenoble Cédex, France

This paper presents a methodology and a tool box for system-level partitioning in the behavioral domain. The methodology is based on an extended finite state machine model. Partitioning is achieved interactively through the application of five system-level transformation primitives: MOVE, MERGE, SPLIT, CUT and MAP. This scheme allows an interactive exploration of the solution space. The result of the partitioning is a set of interconnected and heterogeneous sub-systems. The partitioning tool box which has been developed is named PARTIF. PARTIF includes an evaluation feedback loop that helps the designer estimate the quality of the design.

Key Words: *System-level synthesis, Partitioning, Communication synthesis, Extended Finite State Machine*

1. INTRODUCTION

In this paper a novel method of interactive system-level partitioning is presented. Partitioning at a high level is essential in order to obtain efficient designs. There are many advantages in performing partitioning at the system-level. Firstly and most obviously, partitioning is useful for a better mastering of the complexity of today's designs by distributing the initial specification among independent partitions. Another advantage is that it helps to bridge the gap between system design and existing hardware and software sub-system design tools, thereby allowing the design and synthesis of complex, mixed hardware/software systems.

The research presented in this paper is part of COSMOS [1], a hardware/software co-design environment. COSMOS is intended to fill the gap between system-level tools and existing synthesis tools. The COSMOS environment makes use of SOLAR [2] a unified synthesis intermediate form for supporting several description levels. The goal is to use this representation for the design of complex, control-flow dominated hardware systems. COSMOS is based on SDL [3] (standard CCITT) for the system side and VHDL [4] (standard IEEE) for the hardware side. COSMOS is thus connected to two kinds of environment: a software system design environment and a hardware design environment. It includes system-level synthesis tools for partitioning, communication synthesis and code generation (See Figure 1), as well as behavioral synthesis tools for architectural synthesis.

Partitioning starts with a set of hierarchical and communicating processes described in SOLAR. This partitioning involves transforming system-level constructs such as parallelism, hierarchy and global transitions into constructs that are more oriented towards existing synthesis methodologies. The partitioning is achieved through the use of five system-level transformation primitives: *MOVE*, *MERGE*, *SPLIT*, *CUT* and *MAP*. The four first primitives are related to partitioning whereas the last primitive (MAP) is targetted towards communication synthesis already generated by the partitioning process. The result of system-level partitioning is a set of interconnected and heterogeneous processes. Each of these processes is independent of all others and may be treated individually by the appropriate tool.

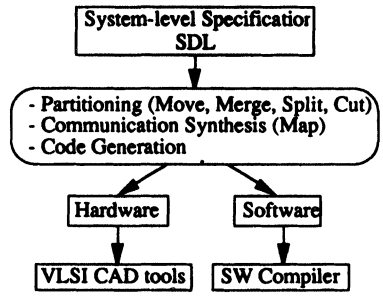

FIGURE 1 System-level Synthesis in COSMOS.

After introducing previous work in partitioning and an extended finite state machine model, the proposed methodology of using the system named PARTIF is described, and the system-level transformation primitives used to achieve partitioning are detailed. Next, an application example treated by PARTIF is described. The remaining sections present future work and conclude the paper.

2. RELATED RESEARCH

Several projects currently in progress (StateCharts [5, 6], SpecCharts and BIF at Irvine [7, 8, 9], Work at Eindhoven [10], SDW at Italtel [11], SIERA [12] at Berkeley, ADL [13] and CODES [14, 15] at Siemens) are all trying to build higher level design tools. Two kinds of solutions are explored today in the field of system-level design automation. The first tries to bring VLSI design and synthesis tools to a higher-level by developing high-level and behavioral synthesis tools. The second uses existing CASE tools for VLSI system design.

Until recently most of the research into partitioning has concentrated on lower levels of abstraction where the system has already been specified at the RT level [16, 17]. Most approaches to partitioning are based on clustering algorithms. This clustering is made according to some *closeness* criteria. These algorithms start with merging objects into clusters until some termination criterion, often related to user-imposed constraints, is met. The credit for developing the initial clustering algorithm is normally attributed to Johnson [18]. Important contributions to partitioning and decomposition methods were made in [19, 20, 21, 22, 23]. A detailed survey of these methods may be found in [24].

Most previous approaches to automatic behavioral partitioning, with the exception of those of Vahid [25] and Barros [26], use an internal representation based on a control/data-flow graph. However, as behavioral description sizes continue to increase, it becomes more and more difficult to realize an efficient partitioning. This is due to the relatively large size of the control/data-flow graph (may contain thousands of nodes) mapped from the initial description. The partitioning approach found in [25] is at the algorithmic-level. At this level, a description is viewed as a set of behaviors, such as processes, procedures, and substates, and a set of storage elements. These behaviors and storage elements are then partitioned such that each partition represents a chip. The goal of this approach is to satisfy chip-capacity constraints. However, the behavioral specification, written in SpecCharts [9], does not allow the declaration of global variables over concurrent processes. This implies a reduced possibility of sharing storage elements. Indeed, the communication model provided by SpecCharts consists of distributing all communication control among the communicating processes making them slightly more complex and also making it more difficult to extend the scheme by adding a new process for example.

Our solution has been to use a general system-level intermediate form called SOLAR. This makes the methodology independent of the description language, which may be software or hardware-oriented. SOLAR supports high-level communication concepts including channels and global variables shared over concurrent processes. Shared variables and channels are then taken into account during the partitioning task.

One of the main problems in partitioning at the system-level is to find a good cost function as it is difficult to estimate the size and/or the performance of the target design at this level of abstraction. Several approaches are trying to solve this problem [25, 26]. In order to get around this problem, we decided to implement a partitioning tool box approach leaving the main decisions to the designer. The partitioning involves a set of partitioning primitives and an evaluation feedback aimed to help the designer in making decisions.

3. SYSTEM-LEVEL MODELING AND REPRESENTATION

Several system-level description languages exist. Regardless of the syntax of these languages which may be graphical or textual, most of these languages are based on a few basic concepts. The four main concepts are hierarchy, concurrency, communication and synchronization. Most of the paradigms listed above are handled by system-level description languages oriented towards protocols (SDL [3], ESTELLE [27], LOTOS [28, 29] etc.), real-time systems (StateCharts [5], ESTEREL [30]) and parallel programming (CSP [31], Occam [32], CSML [33]).

Several models may be used to analyze and verify such languages. These include the trace theory (based on Turing machines), petri nets, the single FSM model, and extended FSM models [34]. The most popular model is the extended FSM. All of these models provide a formal analysis of descriptions. They may be classified according to their modeling power, analytical power and clarity power [35]. The extended FSM model seems to be a good trade-off. Besides, the manipulation of this model is the easiest to automate.

3.1. Modeling complex behaviors

The SOLAR intermediate form allows to describe system-level concepts using hardware semantics. An extended FSM model is used to represent behavioral

descriptions in SOLAR. The extensions include support of hierarchy, concurrency and communication between individual FSMs. In SOLAR, we can represent two types of hierarchy; one at the system-level and one at the process-level. The system-level hierarchy is modeled by the DesignUnit (DU). A DesignUnit can either contain a set of other DesignUnits and communication operators known as ChannelUnits (CU), or a set of transition tables modeled by the StateTable (ST) operator. This operator is used to model process-level hierarchy. StateTables can execute in parallel or sequentially. They can contain other StateTables, simple leaf states, state transitions and global actions which represent exceptions. As shown in Figure 2, a StateTable consists of a (possibly empty) set of declarations and a combination of States and StateTables. Transitions between states are not level restricted. In other words, transitions may traverse hierarchical boundaries. However, in the case a transition between two states is not explicit, the default next state is used. An entry states list indicates all the reachable states by other StateTables. In addition, the default entry state is the first state in this entry states list.

Figure 3 shows a hierarchical system modeled by a set of communicating FSMs. In Figure 3(a) we use the StateCharts notation [5]. Each box represents an FSM. Unbroken arrows imply transitions between FSMs. A dashed line between two FSMs denotes that these two execute concurrently. In Figure 3(b) we use a tree-like representation of the same hierarchical system. It is assumed that a process exists at the start of each fork. The topmost joint is the highest level of the hierarchy. Single lines between two branches of a fork indicate that the two branches are exclusive. Double lines imply parallelism.

These representations are an extension of the classical state diagram with the addition of two concepts, hierarchy and parallelism [5, 9]. In this diagram a state may be one of the following:

- A leaf state.
- A set of sequenced states.
- A set of parallel states.

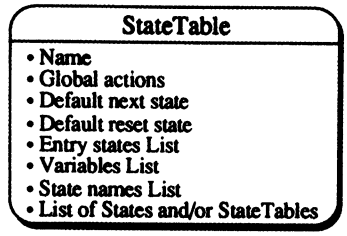

FIGURE 2 StateTable attributes.

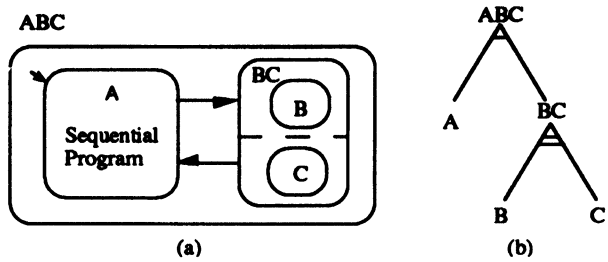

FIGURE 3 The Extended FSM Model: (a) StateCharts Representation, (b) Hierarchical Tree Representation.

When designing heterogeneous systems, one of the major bottlenecks in terms of overhead and performance is the fact that the different processes will inevitably need to communicate. Futhermore, the number and relative execution times of these processes is not fixed. In other words, communication might be necessary between serial or parallel processes, there may be many different configurations (1-1, 1-n, n-1) and the communication may be synchronous or asynchronous. In order to model so many different schemes we need a model that is semantically clean and separable from the rest of the design representation.

3.2. Modeling communication

With SOLAR, a system is structured in terms of communicating DesignUnits (See Figure 4(a)). SOLAR DesignUnits and ChannelUnits may be defined by the attributes shown in Figure 4(b) and Figure 4(c).

Each DesignUnit's contents can be behavioral or structural. The contents of a behavioral DesignUnit consists of a list of Variables (declarations) and

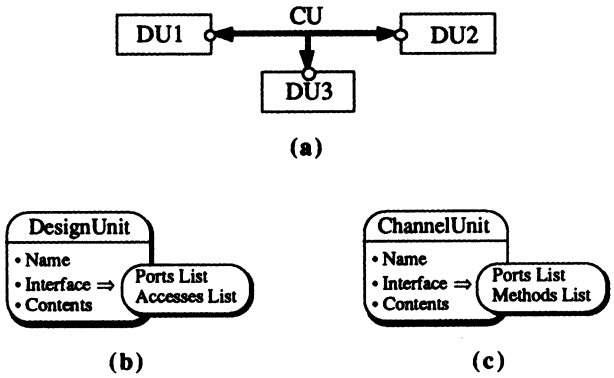

FIGURE 4 (a) Communicating DesignUnits, (b) DesignUnit attributes, (c) ChannelUnit attributes.

StateTables, whereas the contents of a structural DesignUnit consists of a Netlist and a list of Instances.

Communication between sub-systems (or DesignUnits) is performed using SOLAR's ChannelUnit [36]. The ChannelUnit combines the principles of monitors and message passing. It is possible to model most system-level communication properties such as message passing, shared resources and complex protocols. The model is known as the Remote Procedure Call or RPC. The RPC model is a mechanism that allows processes to communicate across message-carrying networks. The networking services are transparent to the user and communication is invoked using the semantics of a standard procedure call.

The ChannelUnit allows communication between any number of DesignUnits (a DesignUnit may be viewed as a system-level process). Access to the ChannelUnit is controlled by a fixed set of procedures known as Methods (services). These Methods correspond to the *visible* part of the channel. In order to communicate, a Design Unit needs to access at least one Method. It achieves this through the use of a special procedure call statement known as CUCall. In other words, the channel acts as a co-processor for the processes using it. Figure 5 shows a conceptual view of a simple channel. In this example, the channel contains two methods named Read and Write. Not only does this model enable the user to describe a wide range of communication schemes, it also separates the communication from the rest of the design, thereby allowing communication synthesis algorithms to be applied directly. The rest of the ChannelUnit (See Figure 4(c)) is completely transparent to the user and consists of a set of I/O ports linking the Methods' parameters to the channel's controller (contents). The controller stores the current state of the channel as well as conflict-resolution functions. The Methods interact with the controller which in turn modifies the channel's global state and synchronizes the channel.

Note that, in the case where the starting point is a system-level language (e.g. SDL), the description of the SOLAR ChannelUnit is obtained by translation from this language to SOLAR. This ChannelUnit contains the definition of all communication signals, Methods, and the protocol relevant to the corresponding communication scheme [36].

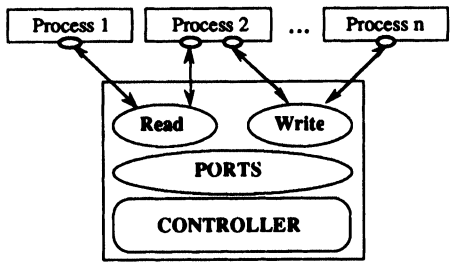

Figure 5 Structure of a ChannelUnit with two Methods.

4. PARTITIONING APPROACH

4.1. Principles

In this paper we focus on partitioning operating at the system-level. The main problem in partitioning at the system-level is the cost function as it is difficult to estimate the size and/or the performance of the target design at this level of abstraction. The goal is to provide a feedback to the designer at each step of the process, thereby allowing a close interaction with the designer and an exploration of different partitioning alternatives. The designer can then choose the most efficient solution that meets his constraints. Figure 6 shows three possible partitions of the machine of Figure 3. In Figure 6(a) the control for each hierarchical level is kept in the immediately preceding level. In other words, the process ABC contains all of the control necessary for sequencing A and BC. These processes are fired by ABC as required and when they have finished they report this fact and ABC decides the next state. In a similar way, process BC controls the sub-hierarchy containing B and C. This implementation requires a lot of overhead due to hierarchical communication. Such a technique is used in [37]. Figure 6(b) shows the opposite solution, that of flattening the hierarchy and having one process (FSM). This flat FSM approach suffers from the state explosion problem that renders it inefficient for complex circuits. Figure 6(c) shows a compromised solution. By merging certain processes into single FSMs, we have the benefit of reduced control overhead without the state explosion problem.

Let U be a symbol that denotes union. Let + and * be symbols to denote sum and product. Figure 7 presents a table that shows the effect of merging sequential machines and parallel ones.

Merging parallel FSMs may cause state explosion. For example, when we merge two processes containing n states each, the resulting machine will contain n^2 states. However, methods for reducing unreachable states exist [38].

Let us consider the case where we want to find the best combination of processes from Figure 3(a) in order to reduce both the number of states and the total number of functional units. Table I gives the estimated cost for each of these processes. For example, state B contains eight states and needs two adders. Let us assume that the cost

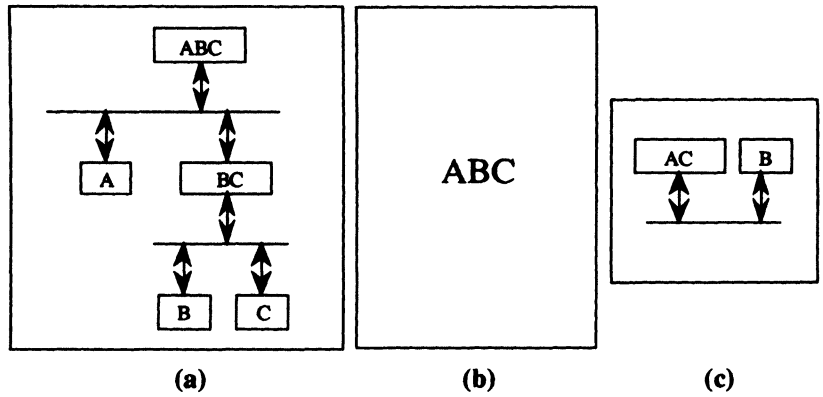

FIGURE 6 Potential partitions: (a) Distributed control, (b) Centralised control, (c) Trade-off.

function includes only the number of states and the number of functional units.

Table II shows five possible merges of the three leaf states in Figure 3(b) and the resulting hardware required. According to this simplified cost function, we see that, from row three, we obtain the best solution by merging processes A and C. This is because we are able to share hardware resources as these processes are exclusive (See Figure 3(b)).

4.2. Input

The starting point of PARTIF is a set of communicating processes organized in a hierarchical manner and described in SOLAR, as shown in Figure 8. Each process represents an extended FSM. These processes may operate concurrently or sequentially relative to other processes that constitute the design. Associated with each process will be various evaluations that are dependent on different features of the processes such as data-path operators, control operations, I/O operations, shared variables, states and so on. Once each of these evaluations has been established, interactive partitioning can start to find the best clustering combination for the design. The main operations performed are splitting and merging extended FSMs. As shown in Figure 8, another input to PARTIF is a library of communication models. A communication model defines the description of the protocol to be used. A similar approach of using a library protocol to be used. A similar approach of using a library of protocol descriptions was given in [39]. The remaining input to PARTIF consists of user-imposed constraints such as the number of partitions, the maximum number of shared variables between partitions and the number of pins allowed. Each violation of these constraints is signaled by PARTIF.

4.3. Output

The output of PARTIF is a set of interconnected processes where all system-level constructs such as hierarchy and communication has been removed and with the overall control distributed among the processes. This is shown in the bottom block of Figure 8. Each of these processes has a well-defined interface and may be treated individually using different design and synthesis tools. Another output of PARTIF is a feedback which includes some statistics about:

1. Resulting interconnections.
2. Shared variables.
3. Hardware area.
4. Data-path operators.
5. Local variables.
6. Hardware computation time.

This feedback allows the designer to evaluate the quality of the design and to compare different partitioning alternatives. According to Wolf [40], analyzing and

	Operators	States	I/O
Seq FSMs	U	+	U
// FSMs	+	*	U

FIGURE 7 Result of merging FSMs.

TABLE I
Cost Estimation for Each Process in Figure 3

FSM	States	FUs
A	4	+−
B	8	++
C	10	+−

TABLE II
Alternatives of Partitioning the System of Figure 3

		States	#FUs
1.	No Merge	22	6 (+−,++,+−)
2.	Merge A, B	22	5 (++−,+−)
3.	Merge A, C	22	4 (+−,++)
4.	Merge B, C	84	6 (+++−,+−)
5.	Merge A, B, C	84	4 (+++−)

modifiying a system requires examining two types of element: state machines and the connections between them. Features 1 and 2 are concerned with the interface of a partition, while features (3, 4 and 5) are concerned with the size of the resulting partition. The interconnections result in an overhead due to the use of I/O units and communication units. The data-path operators within a partition would result in hardware functional units and local variables imply the use of registers.

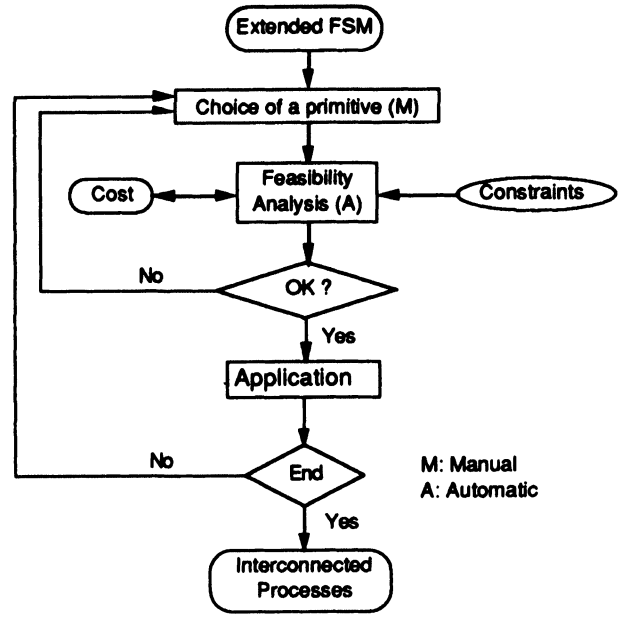

FIGURE 9 The Methodology of PARTIF.

4.4. Method

The methodology used in PARTIF is based on an interactive partitioning that allows a close interaction with the designer. In this approach, partitioning is achieved through the use of five system-level transformation primitives: *MOVE*, *MERGE*, *SPLIT*, *CUT* and *MAP*. These primitives will be detailed in the next section. Partitioning uses some form of evaluation that considers the benefit of merging or splitting processes. Criteria that need to be considered include the six features listed in the previous section. The manner in which these items are shared among the processes is important and also the number of each item takes its importance. For example, the same variable may be used in two different processes and this fact would make these two processes strong candidates to be merged. Similarly, if processes can make use of common hardware this would be very favorable. PARTIF allows the designer to modify the hierarchy and the parallelism of a specification. In general, to go faster we would maximize parallelism and/or minimize the hierarchy and to reduce area, we would minimize parallelism and/or maximize the hierarchy depth. Figure 9 summarizes a typical design process using PARTIF. Once the designer has chosen a primitive to apply, an automatic feasibility analysis is carried out. The feasibility analysis performs the primitive operation, while maintaining the original graph, and then evaluates the result. This evaluation consists of testing if the partitioning meets all the user-imposed constraints. If this is the case, the selected primitive is applied. The designer will be able to repeat this process until an acceptable partitioning is found.

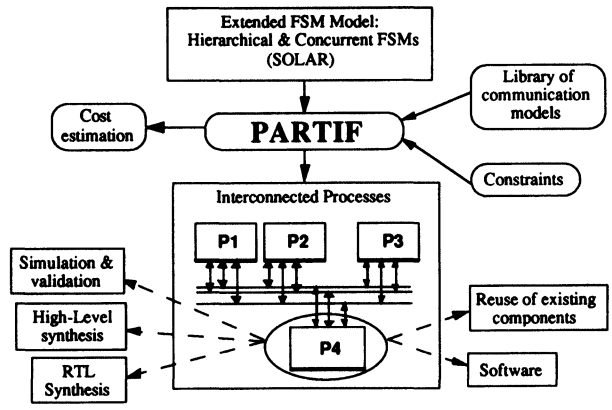

FIGURE 8 The PARTIF Environment.

5. PARTITIONING PRIMITIVES

This section introduces the five primitives: *MOVE*, *MERGE*, *SPLIT*, *CUT* and *MAP*. Only a brief description will be given. The algorithm of each of these primitives is detailed in the appendices.

Primitive MOVE

This primitive allows the transfer of a process (state or statetable) across the system's hierarchy. It may also be used for the migration of code from software to hardware. This operation would normally be a transitory step

before performing one of the other primitives *MERGE*, *SPLIT* or *CUT*.

Definition: We define a path between two machines of different levels in a hierarchical tree representation, as the list of intermediate nodes (machines) belonging to the shortest path between these two machines. This path is said to be sequential (parallel) if all of its own machines are sequential (parallel). ♦

MOVE (P,Q) means that we move machine P to machine Q. This move is possible only in two cases: (1) The path between P and Q is sequential. (2) The path between P and Q is parallel. We assume that P is not a sub-state of Q. Figure 10 shows an example of moving a machine in a parallel path. The algorithm of primitive MOVE is presented in appendix 1.

Primitive MERGE

The Merge primitive fuses two sequential processes into a single process. The objective may be for example, resource sharing among exclusive operations. Registers may be shared by local variables of the merged machines. Similarly, if processes can make use of common functional units (Adder, Multiplier...) they would be strong candidates to be merged. In most cases merging processes implies a better use of resources. This means a reduction in the subsequent hardware required to implement the design.

The *MERGE* primitive has two parameters representing two FSMs. We denote by *MERGE*(A,B), the fusion of the machines A and B. The result is a new machine named AB. The algorithm of primitive MERGE is presented in appendix 2.

Primitive SPLIT

This primitive transforms a sequential machine into a parallel machine. This is achieved through the introduction of idle states and extra control signals. These will be used to control the activation and deactivation of the resulting FSMs. One can note that the value of the control signals gives the global state of the system.

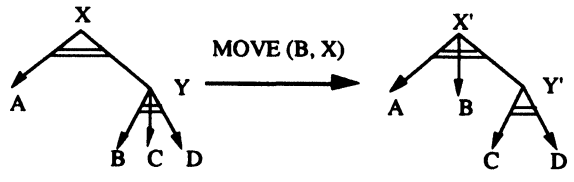

FIGURE 10 Primitive MOVE application Example.

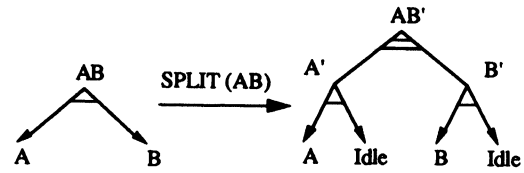

FIGURE 11 Primitive SPLIT application Example.

Figure 11 shows a sequential machine AB with two processes A and B. *SPLIT*(AB) transforms this machine AB into a parallel one named AB′, which contains two states A′ and B′. The state A′(B′) contains two sequential processes A (B) and *Idle* (See Figure 12). In this case two control signals ctrl_A and ctrl_B have been generated automatically.

When machine AB′ is activated, its components A′ and B′ become active. However, if one of them is in its active state (A or B), the other will always be in its idle state.

When the imposed constraints on size and/or the number of interconnections are not satisfied, the primitive *SPLIT* can be used. In the case of a parallel machine, an idle state is added to each component. In fact, to exit a parallel machine, we have to exit all of its components, which means that we enter an idle state. The algorithm of primitive SPLIT is presented in appendix 3.

Primitive CUT

The primitive Cut transforms a set of parallel processes (states) into a set of interconnected processes (Design Units). Parallel processes that share variables will be interconnected through communication channels that contain protocols governing access to these variables. Currently shared variables are restricted to single-dimensional variables (register). A communication channel is assigned to each shared variable. The protocol of this channel depends on the type of access to the shared variable. A variable is one of these cases:

- Exclusive Read Exclusive Write (EREW).
- Concurrent Read Exclusive Write (CREW).
- Concurrent Read Concurrent Write (CRCW).

Only the two cases EREW and CREW are treated automatically by PARTIF. Thus, each time a shared

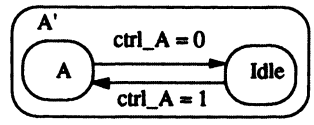

FIGURE 12 The machine A'.

variable has to be used, we call its corresponding channel. As stated above, a channel is composed of a set of services known as Methods (services), a controller acting as a resolution function and an interface linking the Methods' parameters to the channel's controller. The choice of the channel is left to the designer. It depends on the case of the shared variable (EREW or CREW). A channel governing access to a shared variable offers two services, *READ* and *WRITE*. This communication is achieved through RPC calls. The SOLAR channels corresponding to the cases EREW and CREW only differ by the protocols executed. The primitive *CUT* resolves the problem of interleaving. In the case of CRCW, it is left to the controller of the corresponding channel to deal with concurrent writes to shared data.

The primitive *CUT* defines interfaces between sub-systems and also the communication protocols needed. After applying the primitive *CUT*, each sub-system will communicate with the others by means of its I/O (signals and channels). The primitive *CUT* can only be applied to the root of a hierarchical tree representation. Figure 13 shows the result of cutting a system composed of two concurrent machines (A and B) that share a common variable. These two machines can only access the shared data through the channel named CTRL_DATA. The algorithm of primitive CUT is presented in appendix 4.

Primitive MAP

The objective of the primitive *MAP* is to transform communicating systems into interconnected sub-systems. This primitive performs communication synthesis which may be required after application of the primitive *CUT*. It then realizes a procedure in-line expansion and replaces channel accesses by I/O signals. Figure 14 presents the result of the application of the primitive *MAP* on machine AB of Figure 13. In this example, "ctrl_data" is a signal corresponding to the shared variable, named "data", between both machines A and B. We have also BG, BREQ and MODE which are hand-shaking signals. CTRL_DATA becomes a sub-system that controls access to the shared data. For example, if the shared data is in mode EREW, then at any given time only one process can read or write to it. The algorithm of primitive MAP is presented in appendix 5.

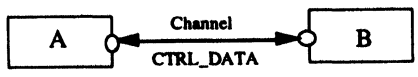

FIGURE 13 Machine AB after application of CUT(AB).

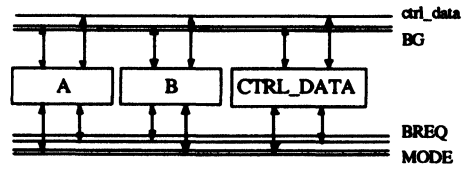

FIGURE 14 Machine AB after application of MAP.

6. AN EXAMPLE

As an example of system-level partitioning, Figure 15(a) shows a simple host-server including a request manager sub-system. This sub-system controls the flow of data between the host and the server. Figure 15(b) shows the hierarchy corresponding to the request manager.

The aim is to partition the request manager sub-system into two partitions. The constraints to be considered for each target partition could be for example:

(1) Total number of interconnections ≤ 8.
(2) At most, one shared variable between target partitions.

As stated above (§ 4.4), partitioning with PARTIF is an interactive process. Several solutions can be obtained by applying different sequences of primitives. The choice of the sequence of primitives is left to the designer. The rest of this section comments a partitioning session including the following sequence of primitives:

- Merge (recv, init).
- Merge (recv, waitstate).
- Split (Request_manager).
- Cut (Request_manager).
- Map.

The result is a solution with two sub-systems. Figure 16 shows the result of each partitioning step. The feedback concerning the two generated partitions is presented in Table III. The objective of the two first merges is to re-organize the top node (Request Manager) into exactly two sequenced machines. In addition, the first merge allows to save one functional unit (Adder), while the second merge allows to save two registers (used by local variables). The primitive Split is then applied to parallelize these two machines. The primitive Cut is used to transform the target parallel machines into interconnected sub-systems. This primitive Cut inserts a new channel in the specification to control access to a shared variable. The last step performs communication synthesis by means of the primitive Map.

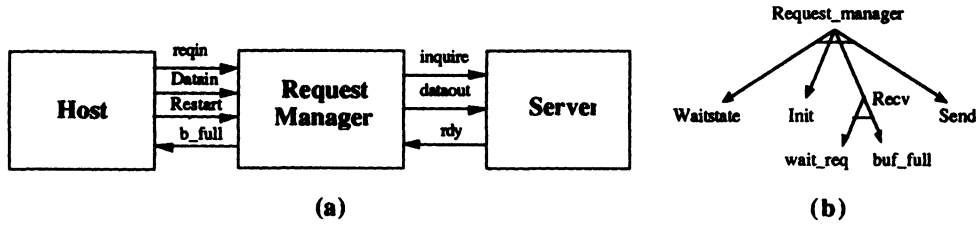

FIGURE 15 (a) The host-server system, (b) Hierarchical representation of the request manager.

7. FUTURE WORK

Future work includes the extension of the PARTIF tool to realize hardware/software partitioning. This implies extending the evaluation functions and also the treatment of user constraints. The evaluation criteria listed in section 4.3 are more targetted towards hardware. For software we should develop algorithms for evaluating the computation time, which provides a means to assist hardware/software partitioning. This extension may require a library of channels that allow communication between hardware and software modules. The physical implementation of the communication scheme depends on the technology used by the communicating modules. If a module is to be executed on a standard microprocessor, communication calls are expanded into system calls,

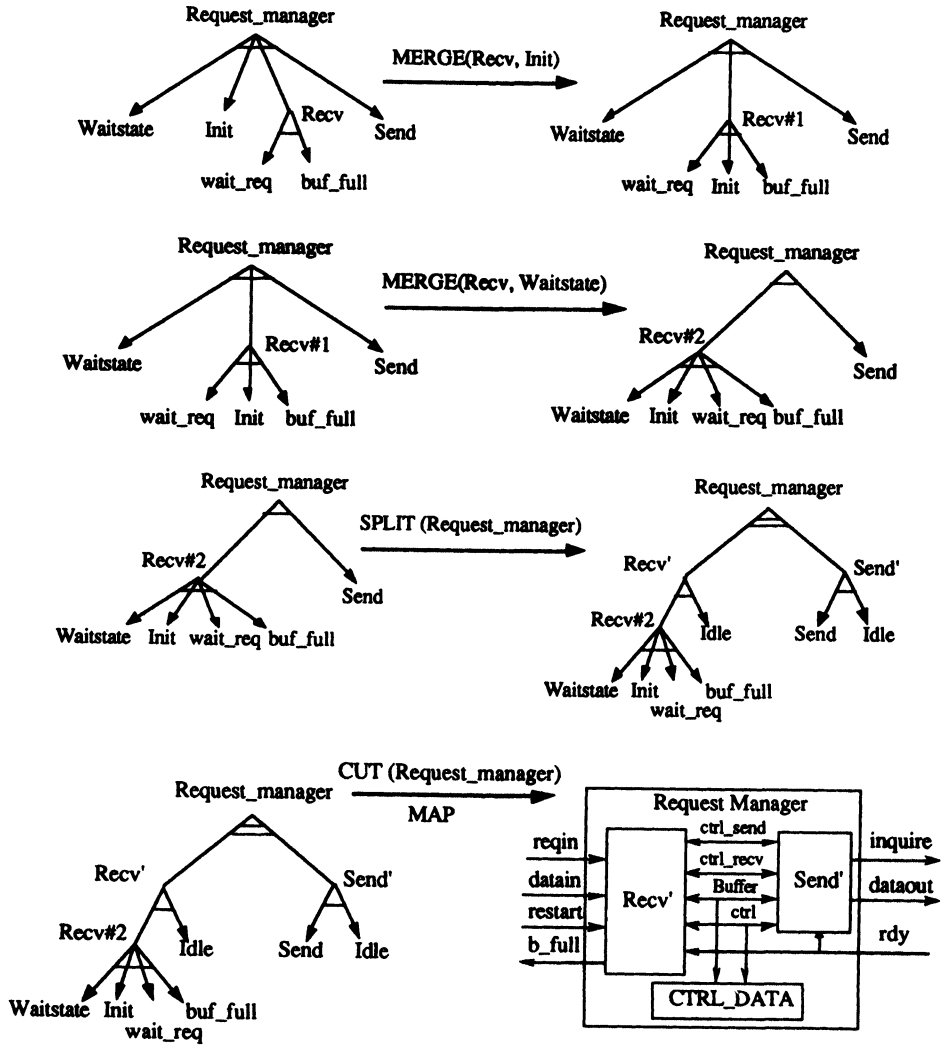

FIGURE 16 Partitioning of the Request Manager.

TABLE III
Feedback of the partitioning

Cost	Partition: Recv'	Partition: Send'
#Inputs	4	1
#Outputs	3	4
#Shared Variables	1	1
Total Interconnections	8	6
Hardware area (mm^2)	1.53	0.74
#Operators (or,+,mod)	3	2
#Local Variables	2	2
Hw Computation time (cycles)	66	34

making use of existing communication mechanisms within the system (for example, SUN's Remote Procedure Calls). However, if a module is to be implemented entirely in hardware, each communication call is expanded in the calling module (procedure in-line expansion). Other research is geared towards the automation of certain partitioning tasks, such as the choice of a primitive to apply.

8. CONCLUSIONS

In this paper an interactive system-level partitioning approach has been presented. The PARTIF tool allows both partitioning and communication synthesis to be carried out. The partitioning uses a flexible communication scheme that separates the communication from the rest of the design, thereby allowing an easy extension to hardware/software partitioning. PARTIF and its partitioning primitives have been detailed. These primitives allow the designer to interactively partition a system-level specification. PARTIF generates a detailed analysis of the partition helping the designer to evaluate the quality of the design. In addition, PARTIF allows the designer to compare different partitioning alternatives taking into account hardware constraints at an early stage of the synthesis process.

References

[1] A.A. Jerraya, K. O'Brien, I. Park, and B. Courtois, "Towards System-Level Modeling and Synthesis", *Proc. VLSI'92*, India, Feb. 1992.

[2] A.A. Jerraya, and K. O'Brien, "SOLAR: An Intermediate Format for System-Level Modeling and Synthesis", *Ed. J. Rozenblit, Computer Aided Software/Hardware Engineering, IEEE Publisher*, chap. 10, To appear in 1994.

[3] R. Saracco, and P.A.J. Tilanus, "CCITT SDL: An Overview of the Language and its Applications", *Computer Networks & ISDN Systems Special Issue*, Vol 13(2), 1987.

[4] "IEEE Standard VHDL Language Reference Manual", *IEEE, New York*, N.Y., 1988.

[5] D. Harel, et al, "Statecharts: A Working Environment For The Development of Complex Reactive Systems", *IEEE Trans. on Software Engineering*, Vol. 16-4, pp. 403–413, Apr. 1990.

[6] R.A. Tierney, "Modelling Complex Systems", *VLSI System Design*, May 1988.

[7] N. Dutt et al, "A user interface for VHDL behavioural modeling", *Proc. CHDL'91*, Marseille, France, April 1991.

[8] D. Gajski, F. Vahid, and S. Narayan, "A Design Methodology for System Specification Refinement", *Proc. of European Design Automation Conf.*, Paris, France, February 1994.

[9] F. Vahid, S. Narayan, and D. Gajski, "SpecCharts: A Language For System-Level Synthesis", *Proc. CHDL'91*, pp. 145–154, April 1991.

[10] M.P.J. Stevens, and F.P.M. Budzelar, "System Level VLSI Design", *Microprocessing & Microprogramming*, Vol. 30, pp. 321–330, 1990.

[11] S. Antoniazzi, and M. Mastretti, "An Interactive Environment For Hardware/Software System Design at The Specification Level", *Microprocessing & Microprogramming*, Vol. 30, pp. 545–554, 1990.

[12] M.B. Srivastava, and R.B. Brodersen, "Using VHDL for High-Level, Mixed-Mode Simulation", *IEEE Design & Test of Computers*, pp. 31–40, September 1993.

[13] U. Wienkop, "Behavioral Circuit Description On The System Level", *Microprocessing & Microprogramming*, Vol. 30, pp. 561–566, 1990.

[14] Buchenrieder, A. Sedlmeier, and C. Veith, "HW/SW Co-Design With PRAMs Using CODES", *Proc. CHDL '93*, Ottawa, Canada, April 1993.

[15] K. Buchenrieder, and C. Veith, "CODES: A Practical Concurrent Design Environment", *Int'l Wshp on Hardware/Software Codesign*, Estes Park, Colorado, October 1992.

[16] T.S. Payne, and W.M. van Cleemput, "Automated Partitioning of Hierarchically Specified Digital Systems", *Proc. of 19th Design Automation Conf.*, June 1982.

[17] M.L. Resnick, "SPARTA: A System Partitioning Aid", *IEEE Trans. Computer Aided Design*, Vol 5(4), October 1986.

[18] S.C. Johnson, "Hierarchical Clustering Schemes", *Psychometrika*, Vol 32(3) September 1967.

[19] E. Dirkes-Lagnese, and D. Thomas, "Architectural Partitioning for System-Level Synthesis of Integrated Circuits", *IEEE Trans. Computer Aided Design*, July 1991.

[20] L. Józwiak, "Simultaneous Decomposition of Sequential Machines", *Microprocessing and Microprogramming*, Vol. 30, pp. 305–312, August 1990.

[21] L. Józwiak, and J. Kolsteren, "An Efficient Method for the Sequential General Decomposition of Sequential Machines", *Microprocessing and Microprogramming*, Vol. 32, pp. 657–664, August 1991.

[22] K. Küçükçakar, and A.C. Parker, "CHOP: A Constraint Driven System-Level Partitioner", *Proc. of 28th Design Automation Conf.*, 1991.

[23] M. McFarland, and T. Kowalski, "Incorporating Bottom-Up Design into Hardware Synthesis", *IEEE Trans. Computer Aided Design*, September 1990.

[24] F. Vahid. "A Survey of Behavioral-level Partitioning Systems", UC Irvine, TR ICS 91-71, 1991.

[25] F. Vahid, and D.D. Gajski. "Specification Partitioning for System Design", *Proc. of 29th Design Automation Conf.*, June 1992.

[26] E. Barros, and W. Rosenstiel, "A Method for Hardware Software Partitioning", *IEEE Comp Euro*, 1992.
[27] International Standard, ESTELLE (Formal description technique based on an extended state transition model), *ISO/ DIS 9074*, 1987.
[28] "LOTOS a formal description technique based on the temporal ordering of observational behavior", *ISO, IS 8807*, February 1989.
[29] T. Bolognesi, and E. Brinksma. "Introduction to the ISO Specification Language LOTOS". *Computer Networks and ISDN Systems*, Vol. 14, No. 1, North-Holland, pp. 25–29, 1987.
[30] G. Berry, and L. Cosserat, "The ESTEREL Synchronous Programming Language and its Mathematical Semantics", *Technical report*, ENSM de Paris, 1984.
[31] C.A.R Hoare, "Communicating Sequential Processes", *Comm. ACM*, Vol. 21, No. 8, 1978.
[32] G. Jones, "Programming in Occam", *Prentice-Hall, Englewood Cliffs, N.J.*, 1987.
[33] E.M. Clarke, D.E. Long, and K.L. McMillan, "A Language for Compositional Specification and Verification of Finite State Hardware Controllers", *Proceedings of the IEEE*, Vol. 79, pp. 1283–1292, September 1991.
[34] I.S. Levin, "A Hierarchical Model of the Interaction of Microprogrammed Automata", *Avtomatika i Vychislitelnaya Technika*, Vol. 21, No 3, pp. 77–83, 1987, (Translation from Russian by Allerton Press).
[35] G. J. Holzmann, "Design And Validation of Computer Protocols", *Prentice-Hall, Englewood Cliffs, N.J.*, 1991.
[36] K. O'Brien, T. Ben Ismail, and A. A. Jerraya, "A Flexible Communication Modelling Paradigm For System-Level Synthesis", *Handouts of Int'l Wshp on Hardware-Software Co-Design*, Cambridge, Massachusetts, October 1993.
[37] D. Drusinsky, and D. Harel. "Using StateCharts For Hardware Description And Synthesis", *IEEE Trans. Computer Aided Design*, Vol. 8, No. 7, pp. 798–807, July 1989.
[38] W. Wolf. "An Algorithm for Nearly-Minimal Collapsing of Finite-State Machine Networks", *Proc. ICCAD'90*, pp. 80–83 IEEE, 1990.
[39] W. Glunz, T. Kruse, T. Rossel, and D. Monjau, "Integrating SDL and VHDL for System-Level Hardware Design", *Proc. CHDL '93*, Ottawa, Canada, April 1993.
[40] W. Wolf, A. Takach, and T-C. Lee. "Architectural Optimization Methods for Control-Dominated Machines", *Ed. R. Camposano & W. Wolf, High-Level VLSI Synthesis, Kluwer Academic Publishers*, pp. 231–254, 1991.

APPENDICES

NOTATION:

Let parent (A,B) be the machine containing the two FSMs A and B.
A precede B in parent(A,B) only if the default entry state of parent(A,B) is A.
Let '+' denote the concatenation operator and 'U' denote the union operator
{N_X is the name of machine X}
{G_X is the set of globalactions of machine X}
{D_X is the default next state of machine X}
{R_X is the default reset state of machine X}
{E_X is the entry states list of machine X}
{V_X is the variables list of machine X}
{S_X is the statenames list of machine X}
{$Block_X$ is the list of states and/or statetables of machine X}

Appendix 1

This appendix presents the algorithm of primitive MOVE.

Algorithm:

MOVE (P, Q)
begin
case of path between P and Q
Parallel : $Block_Q := Block_Q$ U {P}
Remove machine P of $Block_{parent(P)}$
Sequential : $S_Q := S_Q$ U { N_P }
if ($E_Q = N_{parent(P)}$) and ($E_{parent(P)} = N_P$)
then $E_Q := \{ P \} U E_Q$
endif
$V_Q := V_Q$ U $V_{parent(P)}$
$V_{parent(P)} := \emptyset$
$Block_Q := Block_Q$ U { P }
Remove N_P of $E_{parent(P)}$
Remove N_P of $S_{parent(P)}$
Remove machine P of $Block_{parent(P)}$
endcase
if P *is a StateTable* then
{ P inherits the *globalaction* of parent(P) }
$G_P := G_P$ U $G_{parent(P)}$
endif
end-move

Appendix 2

This appendix presents the algorithm of primitive MERGE.

Algorithm:

MERGE (A, B)
begin
case of B
statetable : $N_{AB} := N_A + N_B$
$G_{AB} := G_A$ U G_B
if A precede B in parent(A,B) then
$D_{AB} := D_A$
$E_{AB} := E_A$ U E_B
else
$D_{AB} := D_B$
$E_{AB} := E_B$ U E_A
endif
$R_{AB} := R_A$
$S_{AB} := S_A$ U S_B
$V_{AB} := V_A$ U V_B
$Block_{AB} := Block_A$ U $Block_B$

state: $N_{AB} := N_A$
$G_{AB} := G_A$
$D_{AB} := D_A$
if B precede A in parent(A,B) then
$E_{AB} := \{ B \} \cup E_A$
else
$E_{AB} := E_A$
endif
$R_{AB} := R_A$
$S_{AB} := S_A \cup \{ N_B \}$
$V_{AB} := V_A$
$Block_{AB} := Block_A \cup \{ B \}$
endcase
end-merge

Appendix 3

This appendix presents the algorithm of primitive SPLIT. Let n be the number of components in a machine X to be splitted. Each component of machine X is denoted by X_i ($i \in [1..n]$. Let $ctrl\_X_i$ be the control signal that controls machine X_i.

Algorithm:

SPLIT (X)
begin
for i = 1 to n
$V_X := V_X \cup \{ ctrl\_X_i \}$
endfor
for i = 1 to n
case of X_i
sequential : ActionSeq (X_i)
parallel : ActionPar (X_i)
endcase
endfor
end-split

Each of the functions ActionSeq and ActionPar has one parameter, which represents a sequential machine for ActionSeq and a parallel one for ActionPar. Both functions append an idle state and transform the nextstate transitions into assignment of control signals. For example, if we have in machine X_i, a transition to a machine X_j, this transition becomes the following sequence of assignments: { $ctrl\_X_i := 0$, $ctrl\_X_j = 1$, next state = *Idle*}.

Appendix 4

This appendix presents the algorithm of primitive CUT. We assume a parallel machine M to be cutted is composed of n machines denoted by M_i ($i \in [1..n]$).

Algorithm:

CUT (M)
begin
for each machine M_i in M
create a DU_i (DesignUnit)
append M_i to DU_i
for each shared Port used by M_i
create a port in Interface of DU_i
endfor
for each shared variable V used by M_i
append in the specification a CTRL-_DATA channel corresponding to V
append in Interface DU_i an access to the channel CTRL_DATA
for each utilization of V
if (mode = read) then
call the Read method of the channel CTRL_DATA
else { mode = write }
call the Write method of the channel CTRL_DATA
endif
endfor
endfor
endfor
endcut

Appendix 5

This appendix presents the algorithm of primitive MAP.

Algorithm:

MAP
begin
for each DU (DesignUnit)
for each access to a channel of DU's interface
identify the corresponding channel
for each ChannelUnit call to this channel
declare the used method as a procedure in DU
insert the used ports in the interface of DU
transform method calls into procedure calls
endfor
remove the access from the interface of DU
remove all the methods of the channel
declare the channel as a DU
endfor
endfor
end-map

Biographies

TAREK BEN ISMAIL received a Computer Science Engineering degree from Tunis University of Science, Tunisia, in 1991, and a DEA degree from the National Polytechnical Institute of Grenoble (INPG), France, in 1992. He is presently a PhD candidate at the TIMA/INPG Laboratory, with the System-Level Synthesis Group. His research interests include System-Level Synthesis, Design, Specification, Synthesis and Simulation of mixed Hardware/Software systems.

KEVIN O'BRIEN received his B.Eng. and M.Eng. Degrees in Electronic Engineering from the University of Limerick, Ireland in 1987 and 1989 respectively. He then joined the National Polytechnical Institute of Grenoble (INPG), France, where he studied behavioural and system-level synthesis. He obtained his PhD from the INPG in 1993 and has since been working on VHDL-based simulation and synthesis tools with LEDA S.A. based in Meylan, France.

AHMED JERRAYA obtained an Engineering degree in Tunisia (1980) with the award of "President de la republique", a Docteur-Ingenieur Degree in 1983 and a Docteur d'Etat Degree in 1989 from the University of Grenoble in France. He has held a full research position with the CNRS (Centre National de la Recherche Scientifique) in France since 1986. Dr. Jerraya has participated in several successful projects such as the LUCIE system in the early 80s, the symbolic layout STYX and the behavioral synthesis tools APOLLON and SYCO. He then spent one year at Bell Northern Research in Ottawa, Canada, where he worked on system-level modelling and synthesis. He was also the director of the development of the high-level synthesis system AMICAL. He is head of the system-level synthesis group within TIMA/INPG.