2.

Related Work

This chapter presents a brief introduction on the main issues that will be discussed throughout this dissertation. In particular, the state-of-the-art in hardware-software co-design is presented.

2.1 Hardware-Software Co-Design

In recent years the interest in hardware/software co-design has been revived. Although research in co-development of hardware and software is not new (e.g. [43]), its revival occurred mainly due to the availability of techniques such as high-level synthesis of hardware which put hardware development at an abstraction level nearer to that found in software development [117]. At the same time, market pressures make it necessary to produce more adaptable and changeable hardware devices (hardware to software migration) and more performance efficient software systems (software to hardware migration) in order to make products more attractive while decreasing time-to-market. The profitability damage due to a six month delay in a product release can be worse than a 50 per cent cost overrun during design cycle [103].

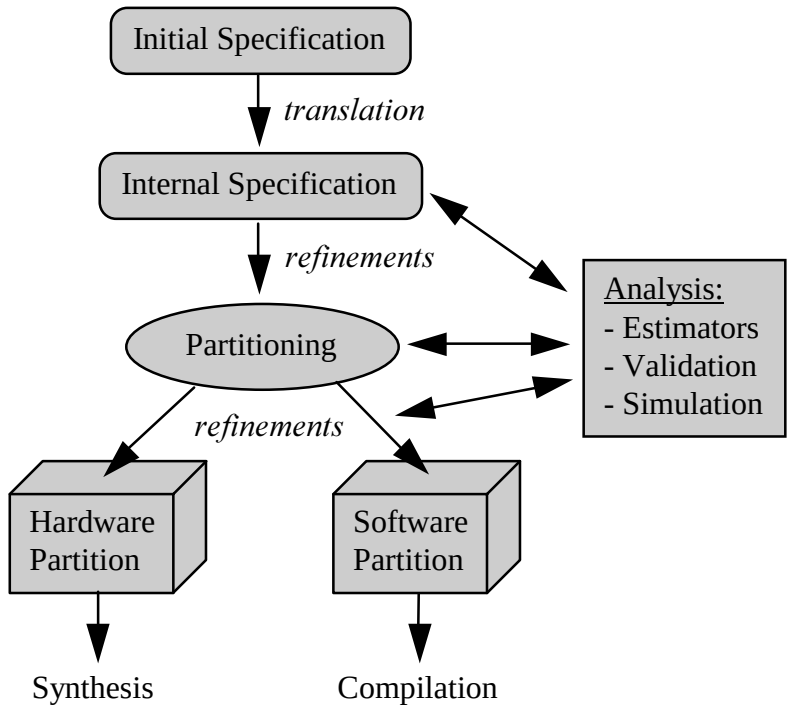


Figure 2.1. Basic co-design approach

Co-design involves several sub-areas as will be shown in the following sub-sections. Figure 2.1 shows a sketch of a basic co-design approach. Although those sub-areas are not as independent from each other as the figure shows, they will be used to emphasise different aspects of co-design. Before going any further, in order to better distinguish between processors where software runs and ASICs, which can also be considered as processors, the following definition is provided, which will be used throughout this dissertation.

Definition 2.1: A *Software Host* is a processor where software is executed.

2.1.1 Co-Specification

This sub-area is related to the design representation and requirements capture problems for mixed hardware-software systems. Generally the design is initially specified at high functional levels, abstracting from implementation and technology details. Successive refinements, interspersed with partitioning, bring the design down to the synthesis and compilation levels. This sub-area is also concerned with the best internal representation for the design. Nowadays, graph-based representations are often used mainly due to the analysis capabilities and expressiveness of the method, while using simple constructors, facilitating automation of the process.

Formal techniques are also being used in co-design. *Transformation based co-design* can guarantee refinement correctness, which is among the main benefits such techniques bring to the area, since one of the major concerns is related to guaranteeing that the final hardware-software implementations and interfaces follow the initial specifications (e.g. see section 2.2.4).

2.1.2 Partitioning

Although co-design has flourished mainly due to the maturity of hardware high-level synthesis techniques, the definition of partitioning for co-design differs from the one found in that area. In hardware-software co-design, partitioning is the stage where implementation paradigm decisions, between hardware and software, are made. Structural partitioning is the one normally found in high-level synthesis, where the design is split in several blocks or chips, in order to fulfil speed, area or pin-count constraints.

Partitioning can be done either manually by the designer, or through automatic tools, or yet interactively using a mixture of both. Interactive partitioning is normally an additional feature present on automatic systems, i.e., the co-design environment can perform automatic partitioning but it is also able to perform several steps towards partitioning and leave the final decision to the user. A good example is the method developed by Barros et al. (see section 2.2.3). Since this is generally an added capability of automatic systems, it will not be discussed in this thesis.

Several different approaches are being used for automatic partitioning. Among those there are: list partitioning (see section 2.2.1), iterative methods (e.g. simulated annealing, section 2.2.2), clustering techniques (section 2.2.3), and min-cut graph procedures (section 2.2.5).

In any case, the existence of good performance estimations for the final design is paramount. Without them the task of choosing the right objects to allocate in each partition becomes very hard since no good design quality measures exist. Many people are primarily concerned with the production of accurate, yet fast estimators (e.g. [49, 88, 122]).

2.1.3 Hardware-software Co-Synthesis

This sub-area is intimately related with partitioning. Co-synthesis is the design stage where synthesis decisions are made. Hardware-software interfacing and scheduling are good

examples of problems related to this area. Its relation with partitioning is clearly noticed. For example, for partitioning purposes it is necessary to define a basic target architecture. In doing so, the designer is already making interfacing decisions. Another example is scheduling. Not all scheduling decisions can be postponed to be done after partitioning, since partitioning affects the system's time response and, consequently, scheduling. However, the designation *co-synthesis* is being used by some authors when the emphasis is on the actual hardware-software interfacing, scheduling, and implementation and not on the partitioning process.

2.1.4 Analysis

In order to make good partitioning decisions it is necessary to have a good knowledge about the system and the final implementation. Due to the excessive time that it would take to actually implement the system, fast estimators are used to provide predictions of final implementation results. However, without reasonable accuracy an estimator provides little or no benefit in the design space exploration, and the research for good estimating techniques is an important co-design sub-area [49, 122].

At the same time, in a methodology where there are several abstraction levels it is important to know if the refinements that lead from one level to the other are in keeping with the initial requirements. Formal techniques can be used either to validate those refinements or, as said before, to carry on the refinements through formal transformation with no need to provide extra validation, in a *correct-by-construction* fashion.

2.1.5 Co-simulation and Prototyping

The problem of simulating systems comprising software and hardware components is a great challenge. The system is not only composed of different software and (digital and analogue) hardware components, but also is presented at several levels of abstraction, such as functional level, register-transfer level, etc. A good example of a co-simulation environment is Ptolemy [68], an environment for heterogeneous systems capable of dealing with several different paradigms using discrete or continuous time models. Design blocks are encapsulated in structures called *Domains.* Each Domain has a scheduler and provides interfaces between internal and external blocks, defining a computational model appropriate for a particular type of sub-system. Currently four types of domains are supported: Synchronous Dataflow (SDF), Dynamic Dataflow (DDF), Discrete Event (DE),

and Digital Hardware Modelling Environment (Thor). In [27] it is shown how to map existing specification methods over these domains.

In the prototyping area, many authors are using re-configurable FPGAs. Rosenstiel, for example, makes use of these components in the COBRA project [73]. Basic modules are used as building blocks to accommodate arbitrary prototypes. Several types of basic modules are available, from FPGA-based modules to micro-controller or DSP and RAM modules. Each module has several connectors for external communication between modules. One special module is attached to the serial and parallel interfaces of a computer and provides the means for configuration downloading and analysis [116]. Nevertheless, the mapping of specifications into FPGAs, mainly due to gate-count and communication bus limitations, is still a major research issue.

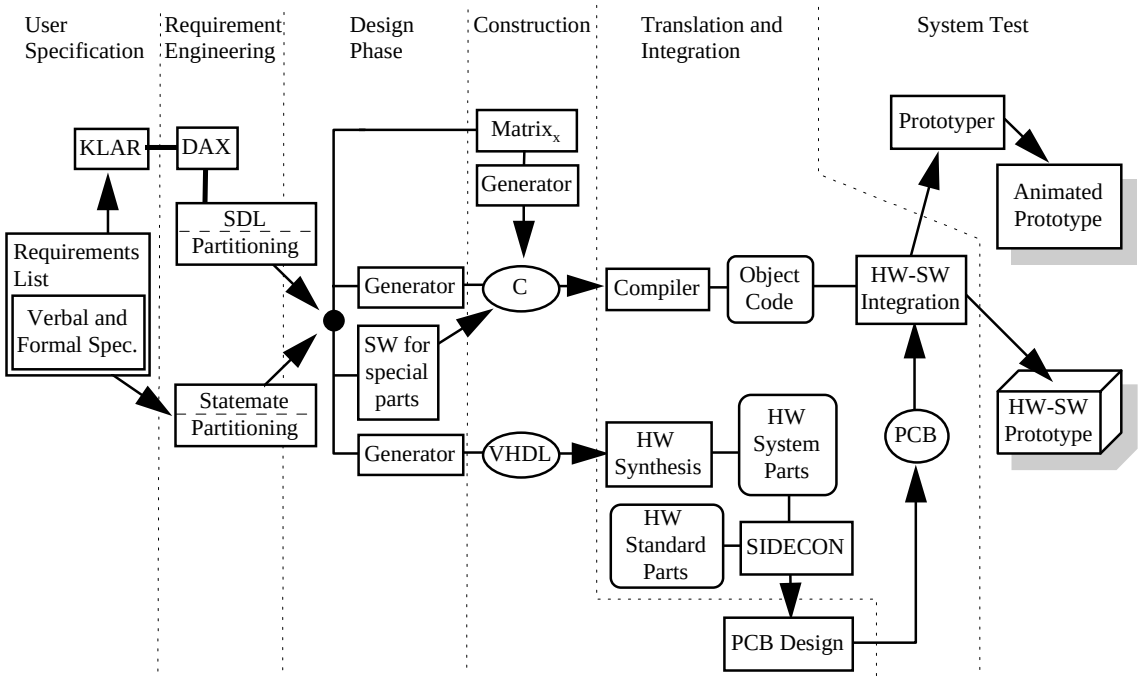


Figure 2.2. CODES Environment

CODES [17, 18] is a co-design prototyping environment that uses the concept of formal PRAMs (Parallel Random Access Machines) which are modelled using either Statemate (based on StateCharts), or SDL tools (based on the CSP model), depending on the features the designer wants to enhance (see Figure 2.2). External modules can also be imported into the model. Each Random Access Machine consists of a finite-state transducer, and local storage. They are encapsulated in an I/O frame used for external communications, which uses a channel-based model. Other tools like Matrix$_x$ (for time-

continuous component design), and SIDECON (to integrate existing hardware components into the system) are used for design, implementation, and integration of the modules. In order to keep track of the design process, a Petri net based framework provides a solution for problems such as design data consistency, synchronisation and identification of project states, and the proper sequencing of project activities. After partitioning, software and hardware modules are output in C and VHDL, respectively, which are them used to build visual or hardware-software prototypes.

2.2 Existing Co-design Approaches

In this section, previous work in the co-design area is presented. A multitude of different approaches are addressed with some comments.

2.2.1 Stanford University

A re-programmable processor assisted by ASIC components with a common memory and a single communication bus form the target architecture of this Stanford University's research group [53]. HardwareC [76] - a C-like HDL - is used as initial specification language for co-design [55], but it is claimed that other hardware description languages could also be used.

Hardware and software standard compilation tools are used together with specific co-design tools, as shown in Figure 2.3. From HardwareC the design is translated to an internal representation based on a set of hierarchical sequencing graphs [56]. A hardware-driven approach is imposed since all initial specification is done in a HDL (HardwareC) and operations are gradually moved from hardware to software. Nevertheless, data dependent delay operations (e.g. data dependent loops, which are considered as unbounded delay operations) are initially placed in software, since they cannot be subject to deadline constraints. All other operations, including external synchronisation operations, are initially placed into ASICs [55]. Groups of operations that are moved to software are serialised and represented as program threads. Each thread contains at most one non-deterministic delay operation (NDO) - e.g. data-dependent loops and receive operations - which is placed as their first operation.

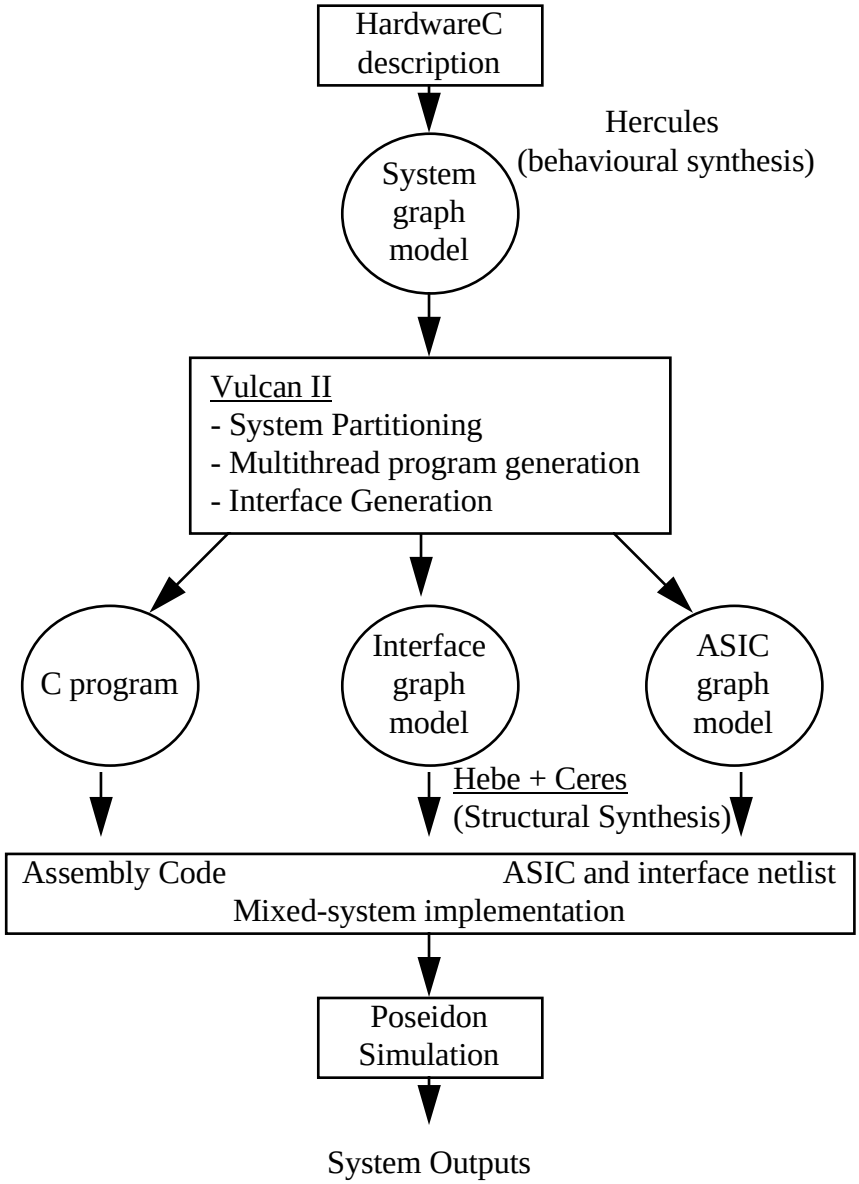


Figure 2.3. Vulcan co-synthesis system [34]

A temporal analysis is carried out and if the initial partitioning is considered feasible, a list partitioning technique is used where the suitability of moving operations from the hardware to the software partition is evaluated based on a cost function. The partitioning continues until either all design constraints are fulfilled or there are no more suitable tasks to move to hardware, in which case the design failed [55]. However, in the event of having an unfeasible initial partitioning then the algorithm fails [52].

Threads are implemented as a set of program routines with their execution sequence being dictated by a control FIFO, which signals the availability of input data [56]. However, a non-pre-emptive scheduling method is used and a worst-case scenario is

considered where all threads are activated at the same time and no attention is given to precedence or mutual exclusion relations [51, 54].

2.2.2 COSYMA (Ernst and Henkel)

[41] presents COSYMA (COSYnthesis for eMbedded Architectures), a software-oriented co-synthesis system for embedded controllers. Software-orientedness is chosen in order to avoid the design restrictions normally found in hardware-driven approaches. These restrictions are normally imposed due to difficulties in implementing certain software constructs in hardware, e.g. dynamic data structures and recursion.

The partitioning approach is called *hardware extraction* since the software description is analysed to select (and extract) appropriate parts for implementation in hardware. This extraction is based mainly on timing constraints.

The input language is C^x, a superset of ANSI C. At the beginning, the initial C^x description is split into basic blocks and translated to a *process dependency graph* (PDG). Sets of communicating tasks are serialised using an algorithm which takes time constraints into account [13], although it uses a non-pre-emptive method. Also, the hardware-software communication mechanism resembles a remote procedure call and causes the blockage of the software host while waiting for completion of the called hardware task. This causes a huge decrease in the solution space since no real parallel solution can be searched. It is worth mentioning that in COSYMA the communication overhead needs to be specially considered since it can surpass the gain obtained in moving a task from one partition to another. In systems that do not have this restriction, the gain due to the increase in parallelism may compensate the loss caused by added communication overheads.

From C^x the design is translated to an extended syntax graph (ES graph, a control data-flow graph), which is a syntax graph extended by a symbol table, local data, and control dependencies. An internal representation based on directed-graphs is used during performance analysis. A two-level loop simulated annealing algorithm is used for partitioning, in order to speed-up this process. The inner loop uses faster less accurate estimations while the outer loop uses actual implementation data [58]. A simulator for the ES graph supports verification and profiling from C^x descriptions, using standard GNU C compiler for software and the OLYMPUS synthesis system for hardware [36], since the software and hardware partitions are presented in C and HardwareC, respectively.

2.2.3 Barros et al.

UNITY is a theory for specifying parallel computations used by Barros and Rosenstiel for co-design purposes [8, 9]. It provides ways to specifying and validating the design by means of formal proof techniques. The asynchronous and synchronous behaviour as well as non-determinism can be expressed in UNITY. A multi-stage clustering algorithm is used [10], which is based on the method used in the APARTY system for architectural partitioning [78]. This method takes into account criteria such as the improvement of parallelism, re-usability of hardware structures, and minimisation of communication costs, each at separate consecutive stages [9]. At each stage a clustering tree based on the current closeness criterion is built using the objects to be partitioned as leaves (see Figure 2.4). Then, the designer places a cut line at a desired height in order to define the resulting clusters, which will then be used as candidate objects for partitioning at the next stage.

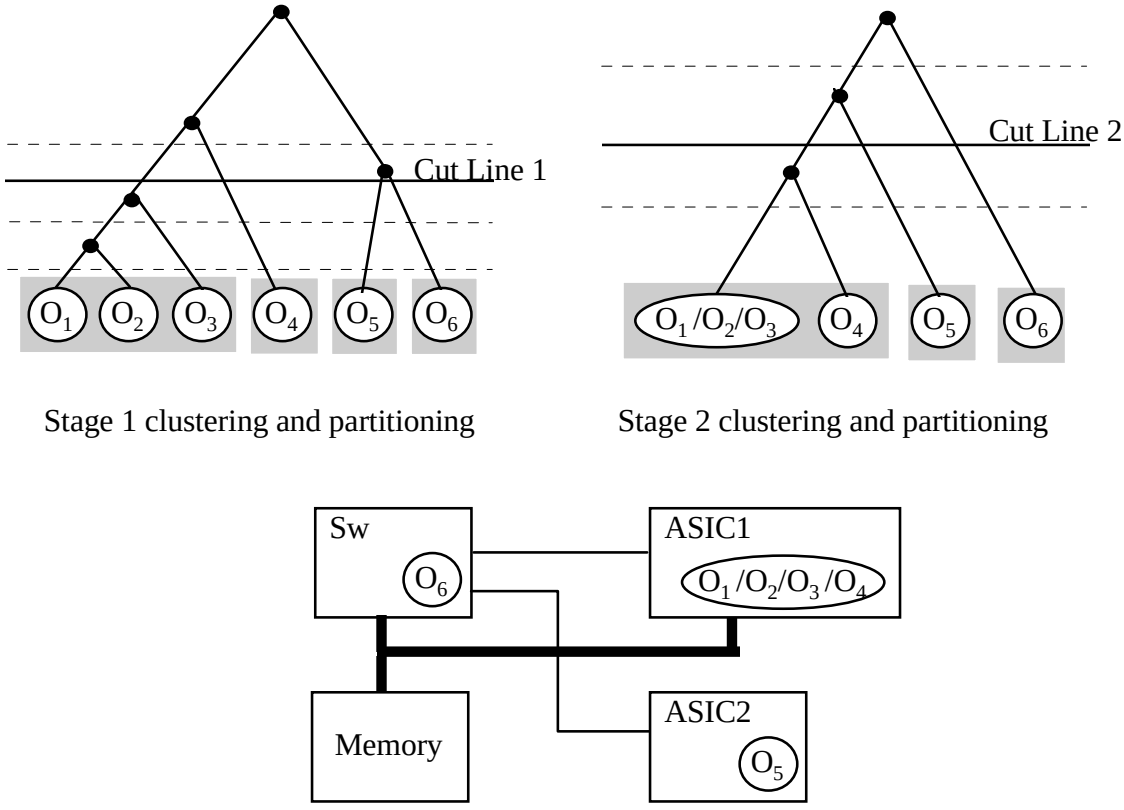


Figure 2.4. Multi-clustering partitioning and implementation [8]

Multi-stage clustering, when compared with traditional clustering methods, reduces the problem of weighting each criterion against each other, making it easier to add new criteria, and reducing the complexity. Conversely, design constraints are not completely independent. Indeed, the infamous area-speed trade-off is a good example. By analysing

each feature separately, there is the risk that one stage prevents a later stage of producing a feasible solution, when one exists.

More recently, Barros has moved to use Occam as an initial specification language [11] which seems to be better tailored to embedded applications while keeping the formal approach. Formal transformations are being applied on partitioning so that the final result reflects the hardware and software interaction while keeping the semantics of the initial specification. Real-time constraints are being considered, although under a no-resource-constraints approach since no scheduling is considered for the software host [86]. The temporal analysis is carried out by transforming the initial Occam specification into a Petri Net and using critical-path search and other worst-case assumptions.

2.2.4 Oxford Hardware Compilation Group - Oxford University

Page et al. are using a formal approach to co-design. Handel-C is a language developed at Oxford for describing computations normally to be compiled into hardware [97]. However, Handel-C is not a hardware description language. It is part of Oxford's research into a single programming language to be used effectively for creating systems with both hardware and software components [5]. It incorporates a subset of C, parallel and communication constructs based on Hoares's CSP [59], and arbitrary width variables and expressions. By having a formal semantics, Handel-C can be used to produce the initial executable design specifications, avoiding the burden caused by later translations [94].

A direct map from Handel-C constructors to hardware is used to compile the initial specification to a mix of combinational logic and flip-flops. The data path formed by these components is an exact match of the initial software dataflow [96]. Automatic transformations can be used to transform the source program to specific types of datapath, since no modifications in the hardware can be done explicitly by the user. For example, there are transformations that result in a combination of machine code program and an application specific micro-controller [95].

By using direct mapping and formal transformations based on re-writing rules, the final design is guaranteed to be an exact implementation of the initial specification. Prototyping is also considered and a computer module called HARP - consisting of a dynamically reconfigurable FPGA, a T805 transputer, and RAM memory - is used as a basic building block [80, 97]. The four transputer serial links and a parallel port are used for external connection (see Figure 2.5).

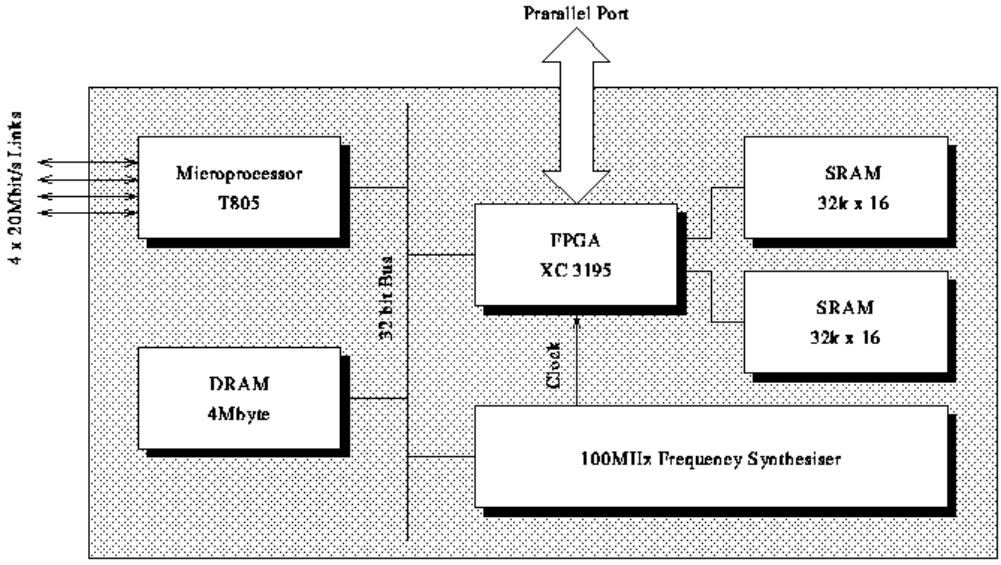


Figure 2.5. The HARP Reconfigurable Computer Module [97]

It is arguable that automatic transformations may lead to oversized implementations when compared to man-made designs, since the later could be specifically tailored for each desired feature. In Oxford's approach this is aggravated by some lack of flexibility since no intervention is allowed except at the source code level, thus eliminating the possibility of directly building particular hardware structures as part of the solution. Hence, some loss of performance and area may be expected when compared to more flexible approaches. However, time-to-market pressures and the design cost incurred in verification and validation make it difficult for designers to seek optimal solutions. Indeed, some experiments using this approach have shown quite acceptable results (e.g. [98]).

At the moment automatic partitioning is not dealt with by the system. Real-time constraints are also not considered.

2.2.5 Eles et al.

In this approach the design is initially specified using VHDL [39, 40]. Then a pre-partitioning phase takes place. It involves three steps:

Step 1. Loops and subprograms are extracted and transformed into new processes together with those previously defined by the user to form a set of interacting VHDL processes.

Step 2. These processes and their interconnections are represented as a graph, where processes are nodes and edges designate the existence of a direct communication

between the connected nodes. Weights are assigned to edges and nodes. Edge weights are proportional to the amount of communication between nodes, while node weights are related to the associated process computation load, uniformity of operations, potential parallelism, and software-implementation suitability degree. Some of these parameters are extracted from simulation results. A simulated-annealing graph partitioning then takes place with the objective of minimising the sum of weights of cut edges.

The node weights are used as constraints for the partitioning algorithm such that only nodes with weights between two pre-defined limits are candidates for moving across partitions. Those with weights smaller that the lower limit should go to the software subgraph, and those with weights greater that the upper threshold are put in the hardware partition.

Step 3. If as a result of the previous steps, a parent process is left in the same partition of one or more of its child nodes, they are merged back in this step.

After this phase, the VHDL partitions specifications are translated to ETPN (Extended Timed-Petri Nets), a Petri Nets extension which includes data-flow descriptions and is used as an internal unified representation of the design [37, 91]. At this point the design can be further refined by using formal transformation and optimisation techniques. Re-partitioning is also considered again by simulated-annealing [99].

Step 4. Finally, the partitions are implemented. A high-level synthesis system called CAMAD is being used to hardware generation, and a VHDL to C compiler is used in the software partition [38].

In this approach the pre-partitioning phase occurs at a high abstraction level. This brings both advantages and disadvantages. The idea of giving an early insight of the possible final partitions to the user is interesting, since it may help to make some decisions about the design at that stage. However, the partitioning may not be very good since rather rough estimators are used [39]. This can mislead the designer by presenting partition results that may be far from the optimal or desired solution.

More recently, the partitioning cost function has been reviewed and two terms were added [40]:

1) One is proportional to the sum of process node weights from each partition, in an attempt to push time-critical processes into hardware.

2) The second tries to increase the degree of parallelism inside the hardware partition by stimulating processes with high computation load, when compared with the amount of communication they have with the rest of the system, to be placed in hardware.

Hardware design area and software memory size have also started to be used as constraints to the annealing algorithm. This parameters are calculated using CAMAD and a VHDL-to-C compiler, respectively.

2.2.6 CFSM

A new specification formalism called Co-design Finite State Machines (CFSMs) is used for hardware-software co-design in an attempt to provide a neutral approach [28]. It is under development at the University of California, Berkeley, and is particularly suited to a specific class of control-dominated embedded systems with relatively low algorithmic complexity. The basic idea is to use a network of interacting FSMs that communicate through very low-level primitives called events, which are emitted by a CFSM or by the environment and may have arbitrary propagation times [30]. A broadcast communication model is used and so events may be detected by one or more CFSMs.

The design process can be summarised as follows:

1) A specification is created in one or more high-level languages that can be mapped into CFSMs. CFSM networks are represented by a specific intermediate format called SHIFT (Software-Hardware Intermediate Format) [29].

2) Partitioning, which currently is a manual task, is performed by the user.

3) Each CFSM is then implemented in the chosen style. In this stage, the non-determinism is removed from hardware partitions which are implemented as fully synchronous circuits. Each software partition is implemented as a C stand-alone program which runs together with a simple operating system in one of the system's micro-controllers [29]. All partitions share the same base clock although software hosts can use a multiple of it.

4) Interfaces are generated following one of the seven pre-defined types [29].

5) Once implementation has been done, verification can be performed. For small designs, this can be done formally by checking FSM equivalence, but problems still remains for large systems due to state space explosion.

Although this methodology lends itself to real-time design and the target architecture may have more than one software host, ways of specifying real-time constraints and the choice of scheduling algorithm to be used are still not defined [30].

2.2.7 Princeton University

OOFS - Object-Oriented Functional Specifications language - is being used in Princeton University for co-specification [60]. It is claimed that OOFS provides means of producing an unbiased design specification. At the beginning of the design process, objects are divided in three groups: hardware, software, and co-design, depending on whether the object implementation is already decided or not. For each object in the software group, a class is defined using C++. Objects on the hardware group also have C++ classes defined which are used in simulations of the design. The objects and functions which implementation paradigms are still uncertain, i.e., which belong to the co-design group, are specified in OOFS. Whenever an object from this group is marked to be hardware implemented it has its specification translated to BESTMAP-C for synthesis, and to C++ for simulation purposes. Objects that are moved to software have their C++ specifications produced. The final output is a C++ program for the software objects and a BESTMAP-C program for the hardware ones.

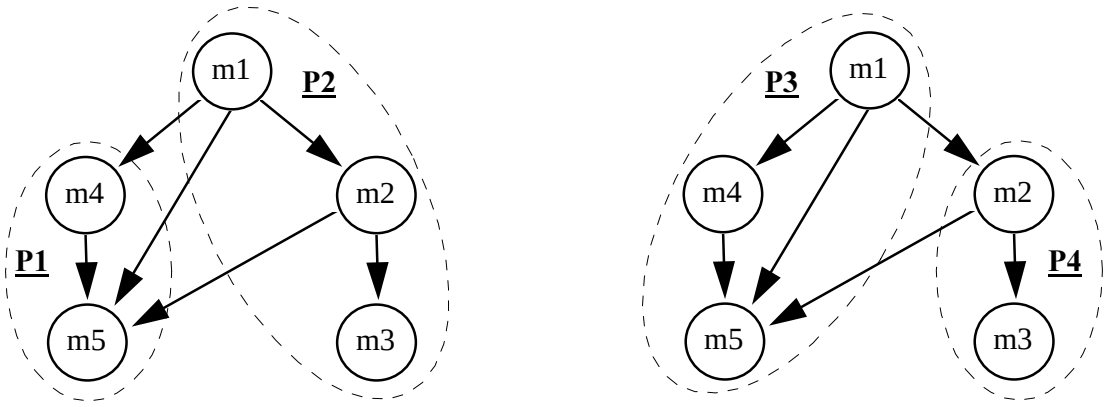
Although the Princeton's group have used object orientation methods for co-specification purposes, recently data flow graphs have started to be used for analysis and partitioning. The design is represented as a set of sub-graphs, where vertices are computational *modules*. The target architecture is composed of several software hosts, and ASICs.

Graph partitioning is performed in an attempt to minimise computation and communication costs, and maximise the advantage of allocating modules to specific software hosts. Each partition is formed by module clustering and is required to form a *completely executable* process, i.e., it may not start until all its input data have arrived, and will not output any data until it has completed execution, which causes some restrictions in the partitioning and allocation scheme. Modules may be moved among software hosts as well.

Figure 2.6.a shows an example of a legal partitioning. Processes P1 and P2 can be formed by the clustering of modules (m4, m5) and (m1, m2, m3), respectively. However, a

completely executable process cannot be formed by clustering (m1, m4, m5) since this would require that m1 sent data to m2 before m5 executes, otherwise a deadlock occurs, and so this would an illegal partitioning (see Figure 2.6.b).

Although the idea of considering not only automatic hardware-software partitioning but also the allocation of modules in software hosts is interesting, since less expertise is needed from the designer, it requires that estimations of each module computation time exist for each kind of processor in the system. Also, no mutual exclusion and release time constraints or communication among sub-graphs is considered, and no time constraints can be specified for individual modules, but only to an entire sub-graph.



a) Legal Partitioning b) Illegal Partitioning

Figure 2.6. Module Partitioning

2.2.8 TASSIM

This methodology is being developed by Hu and D'Ambrosio at General Motors and Western Michigan University based on hierarchical refinement of the design space [61]. The entire design space is considered first, and as decisions are made, model refinements are performed which further restrict the space of solutions. Real-time issues are considered and the primary verification method is based on simulation. At the configuration level, a tool called TASSIM (Task Allocation and Scheduling Simulator) is used to allocate software tasks to hardware processors, and verify schedule feasibility and resource utilisation. At the behavioural level, more detailed models are built, based on results from the previous level, and the hardware-software interaction can be analysed. Verilog is used to model the processor and some aspects of the surrounding environment. The real-time

operating system and the tasks that run on the processor are then simulated by feeding the processor model with instructions streams generated by a C compiler.

Partitioning is performed manually at the configuration level. TASSIM can be used to simulate different scheduling strategies, such as pre-emptive and non-pre-emptive, as well as static and dynamic priority assignment strategies.

GOPS (Global Optimal Part Selection) is a tool for configuration-level hardware synthesis. GOPS uses a branch-and-bound search to produce a set containing the optimal mix of parts to implement those functions, given a set of functions, a library of parts that implement the functions, a set of constraints, and evaluation attributes [33].

A *feasibility factor* named λ_p is defined which is proportional to the probability of a processor *P* being able to host a set of tasks and meet all their timing constraints [33]. GOPS uses this factor as an attribute and a constraint in its search for optimal part sets. However, λ_p is calculated under the assumption that tasks are completely independent from each other, i.e., no mutual exclusion or precedence constraints are considered. TASSIM is then used to verify any results presented by GOPS. This technique relies too much on simulation for verification, and can make the design process go back and forth too many times, mainly when a reasonable number of task inter-dependencies exist. Also, the dependability on the system may be compromised since, although useful, simulation cannot guarantee that all constraints are satisfied unless performed to exhaustion [88].

2.2.9 Chinook

Chinook is aimed at control-dominated designs using off-the-shelf components and is being developed by Borriello et al. at the University of Washington [16]. The design process starts with a Verilog specification, consisting of a structural and a behavioural description (see Figure 2.7). The structural part instantiates the main components such as processors, peripherals and standard interfaces. Devices, processors, interface specifications, and simulation models are kept in libraries, which are used for generating customised drivers for the processor based on timing constraints and for software run-time estimation. Hardware and software components for inter-process communication are created by the *communication synthesiser*. Then, I/O resources are allocated by the *interface synthesiser* to handle processor-peripheral communication, together with any necessary access routines.

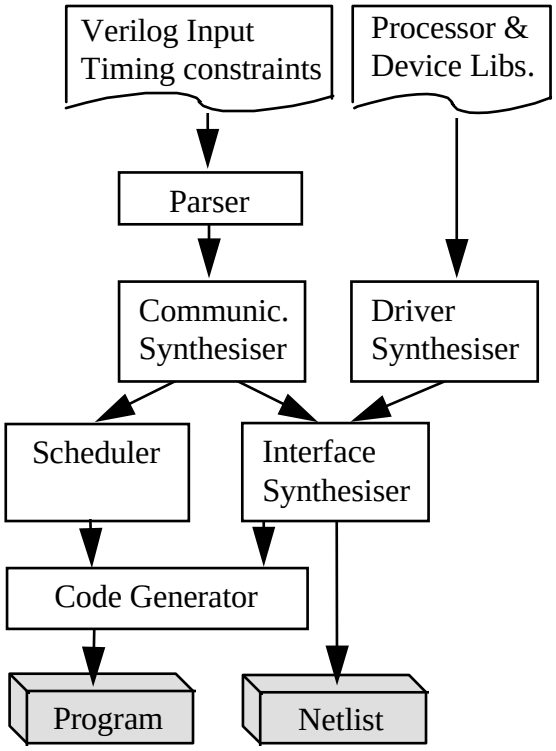


Figure 2.7. The Chinook System

Scheduling is performed at two levels, although both use non-pre-emptive pre-runtime techniques: at the fine-grain operation level (intra-modal) and at the process level [31]. Operation level scheduling is used to guarantee fine-grain constraints, such as time separation between operations and rate constraints. A cluster of operations scheduled together at this level is considered as a single object for the higher-level scheduler [32]. It is not clear how parallelism is handled. It seems that if a task in software calls another in hardware, its software host remains blocked until the hardware finishes the execution.

Partitioning is the designer's responsibility, even though some mechanisms are present to assist the designer.

2.3 Co-design Taxonomy

Table 2.1 shows a summary of the co-design systems described in the former sections to help in clarity of understanding. See below the elements used in this classification.

- *Target Application* defines the main objective of the co-design system being either ASIP (Application Specific Instruction Set Processors) design, or embedded digital signal processing systems (DSPs), or embedded control systems.

- At the *Target Architecture* row, a distinction is made towards those approaches which consider the use of one or several software hosts at their final implementation. Even though many systems claim to allow several processors, it does not mean that all of them specify how to deal with inter-processor interactions and scheduling.

  Special attention is given to those systems that provide co-simulation environments for co-design. They are also mentioned in the literature as *heterogeneous* systems, since they allow for the use of several specification methods at once (see Section 2.1.5).

- *Specification Language*, *Internal Language*, and *Automatic Partitioning* are self-explanatory.

- *Formal Approach* row shows whether formal transformations or validation are being used in the system.

- *Real-Time Specification* defines if real-time constraints and scheduling are explicitly used.

- *Specification Approach.* Some authors consider that co-design systems are characterised by the way the initial partition is thought of, either as completely in software, or hardware, or a mix of both (e.g. [41]). The implication of the paradigm used initially is that in order to have an all-hardware partition some constructs that are commonly used in some software languages need to be excluded since they are difficult to implement directly in hardware, such as recursion and dynamic data types. So, if the initial specification is hardware-driven, it will not present those constructs and vice-versa. Thus, the main advantage of software-driven approaches is the greater design flexibility it provides. On the other hand, not everything specified in such a system can be a candidate for partitioning, as happens in hardware-driven systems.

  However, some groups of systems, such as real-time systems, also take advantage of the lack of those software constructs in order to improve the analysability of the design by producing more deterministic definitions of their components. Therefore, instead of considering the initial partition implementation method, the classification shown here focus on the specification method and separates between those systems that allow and those that do not allow such

software constructs. A systems will be classified as *neutral* when it uses several specification languages with mixed characteristics.

Features		Co-Design Systems										
		Barros	CFSM	Chinook	CODES	COSYMA	Eles	Oxford	Princeton	Ptolemy	Stanford	TASSIM
Target Application	ASIPs							✓				
	Emb. DSPs									✓		
	Emb. Control	✓	✓	✓	✓	✓	✓	✓	✓		✓	✓
	Co-Simulation				✓					✓		✓
Target Architecture	µ	✓				✓	✓				✓	
	Several µPs		✓	✓	✓			✓	✓	✓		✓
Specification Language		Occam	CFSM	Verilog	Several	C[x]	VHDL	Handel C	OOFS	Several	Hardw. C	N/S[1]
Internal Language		Occam	CFSM	Graphs	VHDL&C	Graphs	ETPN	Handel C	Graphs	Several	Graphs	Veril.&C
Formal Approach		✓	✓									
Real-Time Specification		✓		✓		✓			✓		✓	✓
Partitioning/ Specification Approach	Sw-Driven					✓						
	Hw-Driven	✓	✓	✓			✓	✓			✓	
	Neutral				✓				✓	✓		✓
Automatic	Partitioning	✓				✓	✓		✓		✓	

Table 2.1. Summary of co-design approaches

2.4 Real-Time Systems

Definition 2.2: A *real-time system* is any information processing activity or system which has to respond to externally-generated input stimuli within a finite and specified period [123].

[1] Not specified. The definition of this feature was not found in the literature.

Following this definition, it is easy to see that many computer systems fall into the category of real-time systems. A printer control software which needs to output 4 pages per minute and an aircraft flight controller are examples of such systems. It is clear that there is a great difference in criticality between the examples used. In order to provide a more precise classification, other definitions are necessary.

Definition 2.3: *Hard-real-time systems* are those where failure to meet the specified deadlines can lead to intolerable system degradation, and can, in some cases, result in catastrophic loss of life or property [120].

Definition 2.4: *Soft-real-time systems* are those where response times are important but the system will still function correctly if deadlines are occasionally missed [20].

So, reviewing the former examples, a printer control software is considered a soft-real-time system, whereas a flight controller is a hard-real-time system. Soft-real-time systems are normally designed regarding average-case behaviour, since a statistical distribution of response times is acceptable [85]. At the other hand, hard-real-time systems need to have their timing characteristics defined during all executions. Since it is very unlikely that a system process always takes exactly the same amount of time to execute, in the design of hard-real-time systems it is necessary to consider the *worst-case execution time*, or WCET, of the processes involved.

2.4.1 Process Characterisation

Processes which occur regularly at defined intervals of time are called *periodic processes*, whereas *sporadic processes* may be triggered at any time, although a *minimum inter-arrival time* between consecutive invocations exists [92]. *Aperiodic processes* is another class where no information, except possibly for a probabilistic triggering distribution, can be assumed. Since such distributions do not prevent the occurrence of any inter-arrival period, no worst-case estimation can be performed and only soft deadlines may be guaranteed [19].

- Periodic process are characterised by a tuple (p, r, c, d) where:

  p is the period,

  c is the worst-case computation time,

  d is the deadline for a task to complete related to the start of its period,

  r is the release time also related to the start of its period;

- Sporadic Tasks are characterised by (*minp, c, d*) where:

  minp is the minimum inter-arrival time between execution requests,

  c is the worst-case computation time,

  d is the deadline for a task to complete related to the arrival of the request.

2.4.2 Scheduling Methodologies

Besides the process characteristics, real-time systems need to ensure that inter-process relations are considered. These include mutual exclusion and precedence relations. Schedulers are provided to guarantee these constraints, although the search for a solution is an NP-hard problem [44].

Pre-runtime schedulers are those which produce a complete schedule before the system execution starts. *Priority-driven* schedulers use assign priorities to the processes to define their execution order. They imply the existence of real-time kernels which are used at runtime to schedule the tasks. Priority-driven schedulers can be further divided into *fixed* or *dynamic*, depending on whether the priority assignment is performed before or during execution, respectively.

Most priority-driven methodologies are based on same variation of the *rate-monotonic algorithm*, where fixed priorities are assigned to tasks in the inverse order of their periods, i.e., a task with a period greater than another has lower priority. Such a scheme, when applied to a set of completely independent, pre-emptable tasks, with deadlines equal to their periods, is optimum in the sense that if any feasible priority assignment exists, the rate-monotonic assignment is also feasible for that task set [85]. Liu and Layland also found theoretical processor utilisation limits such that if such a set of tasks has a processor utilisation smaller than that limit the rate-monotonic priority assignment is feasible. Their results created the basis for the so-called *rate-monotonic analysis* (for a comprehensive introduction, see [72]).

When comparing pre-runtime and priority-driven schemes, there are advocates on both sides. Although improvements in the rate-monotonic technique have been made so that some of the initial restrictions have been removed [7, 81, 113, 114] this methodology presents poor results when dealing with systems containing inter-process relations. Xu and Parnas show that static and dynamic priority-driven schemes may not be able to find a feasible solution, even if one exists [118]. This position is reinforced by Mok in [92]. In

general, priority-driven techniques lack the ability to properly tackle the problems imposed by mutual exclusion and precedence constraints, release times and deadlines, when different of the beginning and end of period, respectively.

On the other hand, pre-runtime scheduling algorithms are less receptive to changes than dynamic schedulers. Changes are normally made off-line, i.e. a new schedule is produced and the system stopped for upgrading, although some techniques have been developed to overcome some of these limitations [19]. At the same time, sporadic tasks need to be transformed into periodic tasks in order to be statically scheduled. Transformations to this end are defined in [92]. Depending on task timing parameters, these transformations may create equivalent periodic tasks which allocate much more resources than really used. Surveys on process scheduling can be found in [22, 109, 120].

2.4.3 Scheduling in This Dissertation

Both techniques shown above have their application scope. In this work the objective is to tackle hard-real-time systems while minimising costs. To do so, it is considered that software hosts shall be used to their maximum extent, leaving to hardware only the strictly necessary tasks. Therefore, it was decided to use a pre-runtime scheduling algorithm. No definition about task implementation is required since this is performed separately by the partitioner. Since the target architecture consists of several software hosts and several hardware processors (ASICs or FPGAs), and all possible solutions should be available, this algorithm needs to be able to perform multi-processor pre-emptive scheduling over a set of pre-allocated tasks.

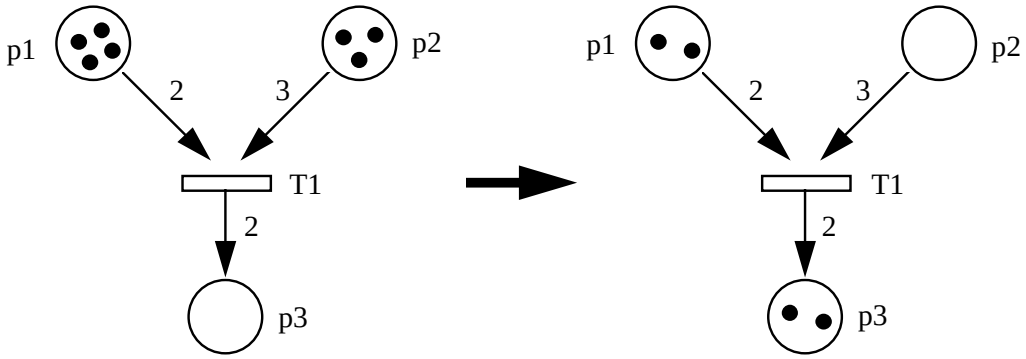
Some authors claim to have produced such an optimum algorithm to solve the problem [107]. However, there seems to be a problem with their algorithms since their technique is not able to always find a solution when one exists. They use an *earliest-deadline-first algorithm* to produce an initial schedule and try to improve it using a branch-and-bound technique where new inter-process dependencies are introduced in an attempt to find a feasible solution, i.e., all timing and inter-process dependencies are fulfilled. However, when processes are allocated to different processors, inter-process dependencies are not considered. Possible solutions may not be found if these dependencies are disregarded in multi-processor architectures. This will be shown in Chapter 5, where a new algorithm is presented to solve the problem.

2.5 Petri Nets

The Petri net is a graph-based mathematical formalism introduced by C.A. Petri in 1962 which easily expresses concurrence, non-determinism, communication and synchronisation (for a comprehensive introduction to Petri Nets see [93]). Petri nets provide many analysis techniques, while keeping a simple way of modelling parallelism and concurrence [104].

Basically, a Petri net is formed by a directed, bipartite, weighted graph with two kinds of nodes, *places* and *transitions*. A weighted arc can connect either a place to a transition or vice-versa (in which case the place is called *input place* and *output place* of the given transition, respectively), and a weight *n* represents *n* parallel arcs with weight 1. Places can hold *tokens* and the disposition of tokens over the places in a net is called *marking*, and represents the state of the net. A transition *t* is *enabled* if each input place *p* is marked with at least *w(p,t)*, where *w(p,t)* is the weight of the arc from *t* to *p*.

Once enabled a transition *t* may *fire* (but it is not obliged to), removing *w(p,t)* tokens from each input place *p* of *t*, and adding *w(t,p)* tokens to each output place *p* of *t*, where *w(t,p)* is the weight of the arc from *t* to *p*. If it is the case that more than one transition is enabled, one or more is randomly chosen to fire. See Figure 2.8.



a) Before firing b) After firing

In a) transition T1 is enabled, since the number of tokens in p1 and p2 is equal or greater than the weight of their correspondent arcs leading to T1. In b), after the firing of T1, two tokens from p1 and three from p2 have been removed, and two new tokens have been created and placed in p3.

Figure 2.8. Petri Nets

Petri nets have proven to be a successful modelling and analysis notation in several areas, including hardware modelling and design [3, 15]. Nevertheless, they offer no support for hierarchy, timing specification, data types and transformations, or general requirements

capture [23]. Many extensions and variations have been proposed to Petri nets to overcome those limitations, as it is seen in the following sections.

2.5.1.1 High-Level Petri Nets

The extensions to Petri nets which permit data transformations are called High-Level Petri nets [48]. Generally, subsets of existing programming languages (such as C or Pascal) are used as annotation languages which are associated to the transitions, places, and arcs.

Jensen proposed *Coloured Petri Nets* [66] where tokens can have different *colours* representing different data types. A similar approach is found in *Predicate/Transition Nets* [46]. If the number of colours is finite, a HLPN can be considered as a folded version of a regular Petri net [93]. Due to their formalism many researchers have started to use them to model and analyse systems for different applications and at various abstraction levels (e.g. [102]).

A good example of a modelling environment based on Coloured Petri Nets is SystemSpecs [64, 65]. A HLPN called Spec Nets is used in the system and SpecsLingua, a Pascal-like annotation language, is employed to determine data transformations (actions) and predicates (conditions). SystemSpecs is tailored to design and simulation of *safe nets* which means that a transition can fire only if there are no tokens in its output places [104]. Timing specification is not considered. Hierarchical nets can be created by transforming parts of the net into subnets called *channels* or *agencies*, depending on the sort of interface the subnet has (places or transitions, respectively). Analysis does not benefit from this hierarchy. Since data transformation is present, a quite realistic model animation can be performed where inputs and outputs are visualised by means of gauges and buttons on the screen.

Recently object-oriented HLPNs Petri Nets made their debut [79] in an attempt to provide better modularisation and hierarchical descriptions. However, the full power for object-orientedness, as provided in PNO [111], allow for dynamic object creation, which is not suitable for hardware synthesis or hard-real-time design. At the same time, *access functions* as presented in LOOPN [71] are contrary to the Petri nets basics, since they provide ways of accessing subnet states without the need of arc connection between the nodes.

2.5.1.2 Time Modelling in Petri Nets

Basically two classes of Petri nets present time modelling capability: Timed Petri Nets (TPN), and Stochastic Petri Nets (SPN). The primary difference is that TPNs have deterministic delays, and in SPN delays are probabilistically defined.

Stochastic Petri Nets are based on Markov Chains [89]. Therefore, Stochastic Petri nets are targeted to average performance analysis, making this approach unsuitable for verifying time critical properties of real-time systems [83]. The main use of stochastic Petri nets is in performance evaluation, not in specification and verification of time-critical issues [48]. Time-Petri nets, on the other hand, are being used for hard-real-time system specification and analysis [2, 82].

2.5.2 Petri Nets in This Dissertation

In order to be able to use a single specification formalism as a means for input as well as internal representation the formalism must deal with hierarchy, timing specification, data types and transformations. So, it is necessary to combine Time Petri nets with HLPNs. *High-Level Timed Petri Nets* (HLTPN) combine the extensions existent in HLPNs and Timed Petri Nets. Appendix A presents Time Environment/Relationship (TER) nets [48], which is the HLTPN used as the basis for modelling throughout this dissertation. As it can be seen in Chapter 3, it was extended in order to deal with specific needs regarding hierarchy and hard-real-time systems. Cabernet is an environment developed at the Politecnico de Milano which implements the ideas proposed in TER nets [100]. A link between the specification tool proposed in this work and Cabernet was developed in order to use the later as a simulation and profiling medium for co-design specifications. This link is also discussed in Chapter 3.

2.6 Summary

This chapter has given an introduction to hardware-software co-design, Petri nets, and real-time systems. In particular, the state-of-the-art in co-design has been presented and a multitude of different approaches briefly explained. It is clear that none of the approaches described solves the problem of having several software hosts working in a pre-emptive fashion and co-operating with hardware components able to operate in parallel, and yet fulfilling critical temporal requirements. This is the primary objective of this dissertation.