3.

Co-Design Methodology

This chapter provides an overview of the co-design system proposed in this dissertation. In particular, it presents details of the co-specification methodology that has been employed in the co-design environment shown in this thesis.

A specification tool called Time Wizard [25] was built to help enforce this methodology and is also presented in this chapter.

3.1 Co-Design System Overview

The co-design method proposed here is tailored to the development of hard real-time embedded control systems consisting of several processes that execute over a set of software hosts and hardware processors. Given the need for deterministic timing analysis required for hard-real-time designs, a hardware-driven approach is utilised. Partitioning in the system is performed automatically. A formalism based on Petri nets is being used as an initial specification medium and internal representation language. Formal transformations are not being employed although this formalism gives room for its use in the future.

Besides the features mentioned above the system presents the following characteristics:

- Full exploitation of the parallelism present in the design, while keeping all timing and inter-process dependencies;
- Maximum use of software hosts (minimum hardware implementation);
- Use of a single, simple, graphical specification formalism.

3.1.1 Design Process

Figure 3.1 exhibits a general outlook of the proposed environment. The design process starts with the specification phase, during which timing requirements and inter-task dependencies are specified, and analysis (simulation, profiling, etc.) is carried out [26]. Analysis tools are also employed in order to guide the transformations and refinements so that the output of this phase is a set of processes with timing information and requirements, as well as process inter-dependencies, in a format suitable to be used by the scheduling and partitioning algorithms. A tool called Time Wizard was built in order to aid the specification process [25].

The architecture selection is also carried out during the specification phase. The number of software hosts and hardware processors in the initial architecture is defined. Although the number of software hosts remain the same throughout the co-design process, more hardware processors may be created when tasks are moved from the software to the hardware partition. Each task initial allocation is also defined here.

In the next step partitioning and scheduling are performed in a loop. Partitioning is accomplished in two phases. The first phase occurs just once and takes into account only the data related to timing requirements and task inter-dependencies. Then scheduling is performed and analysed. If the solution found is not feasible, the second partitioning phase takes place where scheduling generated information is also considered, and the loop starts again. The process continues until a feasible solution is found or no tasks are available to move to hardware, in which case the design has failed and may need re-design.

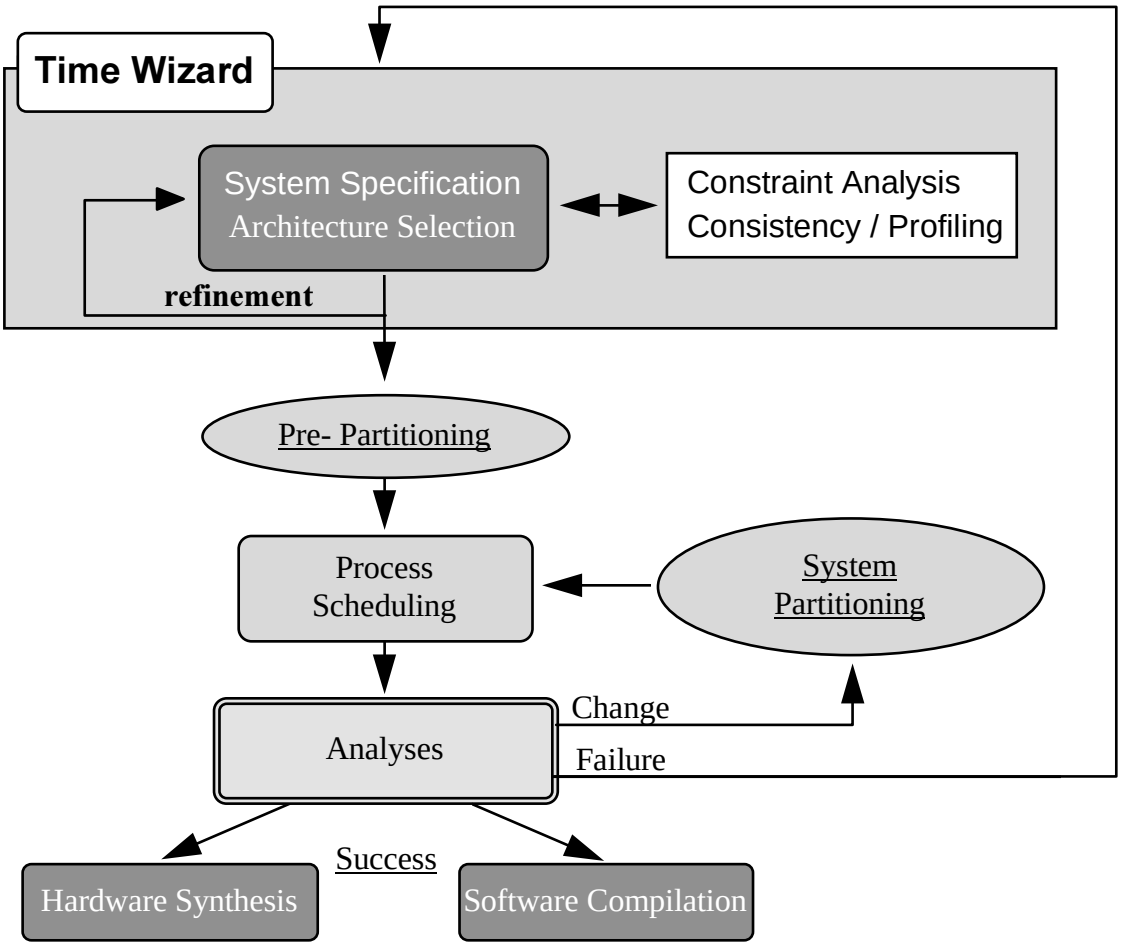


Figure 3.1. Proposed co-design environment

3.2 Design Specification

This section concentrates on the specification phase. Petri nets are used as the basis of the formalism, although some extensions are proposed in order to make it more suitable to the target applications. The basic idea is to allow the user to design the system in a relatively free way and guide the refinements towards the production of a final model suitable to be used by the scheduling algorithm shown in the next chapter. This suitability should be checked by automated tools, thus helping the designer during the refinement stages.

At the same time the model must be able to provide a unified treatment for hardware and software modules, and this is achieved by keeping the design at a high abstraction level and by considering both software hosts and hardware components as processors. The basic difference between them is in the fact that the number of software hosts is known a priori, what does not happen with hardware processors.

Definition 3.1: *Scheduling level* is the level at which the design is suitable to be used as input by the scheduler.

At the scheduling level the specification must be completely deterministic, i.e., all computation times are based on worst-case scenarios and no conditionals are allowed.

3.3 System Modelling

It is considered that the description of a set of tasks with the following characteristics is sufficient to model the design for scheduling purposes:

- inter-task dependencies: precedence and mutual exclusion relations;
- timing information: release times, execution times, deadlines, and periods, since all tasks are periodic at the scheduling level.

Nevertheless, in order to calculate the worst-case execution time and have a better knowledge of the system, it is also necessary to express and possibly simulate the internal data transformations performed by each task. Moreover, it is desirable that the methodology does not restrain the designer's creativity by allowing only periodic tasks at the initial specification levels. Sporadic tasks, although possible to be transformed to periodic tasks (see Section 3.3.6.1), should be available to the designer since they provide a more natural way than periodic tasks of specifying some events, such as interactions with the external environment from which not much knowledge may be known at such an initial design stage. Therefore, the specification method also needs to allow for sporadic task and data transformation modelling.

TER nets are being used as the starting point of the formalism proposed here [48] (see Appendix A). Some additional structures were incorporated and their graphical representation can be seen in Figure 3.2.

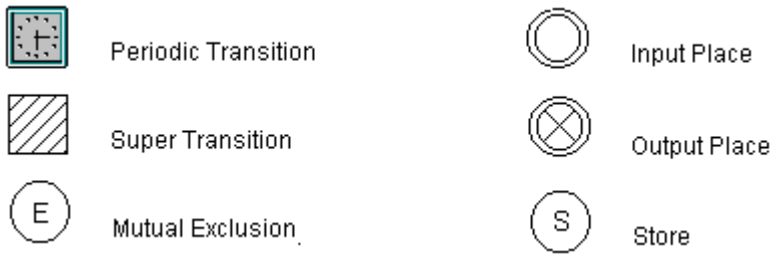


Figure 3.2. Graphical representations of the Petri Net extensions

Each proposed constructor can be modelled using basic TER net primitives and so does not inflict any loss in analysability when compared with pure TER nets. The TER equivalents of the extensions shown above are not important to the understanding of this chapter and so are explained separately in Appendix B. The semantics and importance of each of these extensions will be informally explained in this chapter.

3.3.1 Basic Net

Tokens carry data values which can be used in predicates and actions. Places have associated data types and can only hold tokens with a matching type. Each transition has an associated *predicate* which may be dependent on the value of the tokens in its input places. The evaluation of predicates produce boolean values which are used in the enabling of transitions. When it fires, a transition performs an *action* which may use the values of the tokens in its input places to define the values of the tokens generated to its output places. Only one single edge is allowed in each direction between a place and a transition.

3.3.2 Architecture Specification

Each user-defined processor is declared by specifying its name and whether it is a software host or a hardware processor. If necessary, new hardware processors will be automatically created during partitioning (see chapter 5). Each transition also carries information about the name of the processor to which it is allocated, and a tag called *fixedImp* which determines if this transition can be moved from the current processor to another one. Transitions allocated to software hosts may be moved to hardware if their fixedImp tag is *false*. Tasks allocated to hardware processors cannot be moved to software hosts and so have their fixedImp tag always marked as *true*.

3.3.3 Timing Specification

Each transition carries timing information representing *release time*, software-host maximum execution time (*SwWCET*), hardware maximum execution time (*HwWCET*), and *deadline*. All these values are integers and added by the designer. *SwWCET* is the maximum time a task would take to execute if implemented in software. Conversely, *HwWCET* is the maximum time a task would take to execute if implemented in hardware. This timing information is similar to the model for real-time systems presented in Section 2.4.1.

All timing information is supplied by the user, including WCET values since there are no estimators present in the system. Therefore, *SwWCET* and *HwWCET* need to be supplied by the user, e.g. by retrieving them from component libraries.

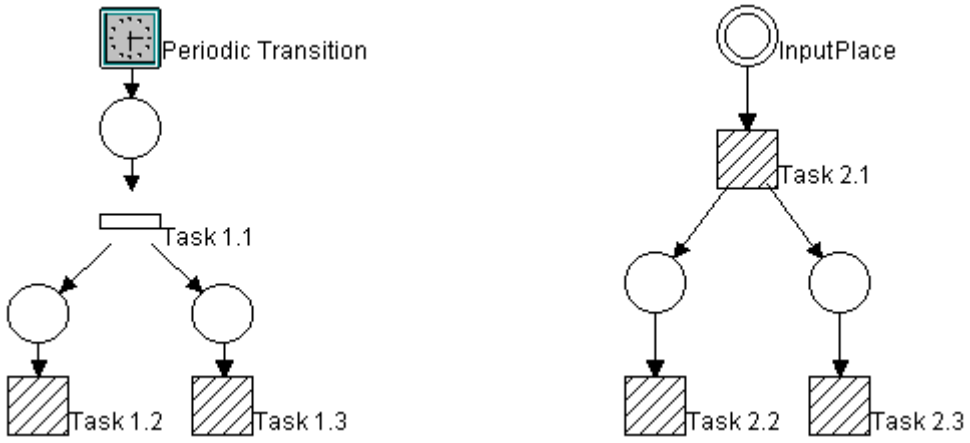
A time variable is attached to each token, representing the time it has been produced. Tokens created during the firing of the same transition carry the same value on their time variables and the firings occur in a monotonically non-decreasing sequence. A transition is enabled if there exists at least one token in each of its input places, its predicate evaluates to *true*, and the current time is within the interval [*release time*, *deadline - WCET*), where *WCET* is equal to *SwWCET* or *HwWCET*, depending on whether the transition's processor is a software host or a hardware processor, respectively.

A transition's deadline and release time is expressed in relation to the triggering of the process net to which it belongs. All deadlines and release times internal to subnets are disregarded at the scheduling level.

3.3.4 Process Nets

The design is represented as a set of interacting *processes*. A *process* is specified as a *process net* which is composed of a set of *tasks* that are triggered by the same triggering action. There are two sorts of process nets (see Figure 3.3):

- ***Periodic net:*** triggered at periodic intervals of time by the firing of a *periodic transition*.
- ***Sporadic net:*** triggered at times not previously defined every time a token arrives on its *InputPlace* .



a) Periodic Net b) Sporadic Net

Figure 3.3. Process nets

A *periodic process net* is composed only of *periodic tasks*, and a *sporadic process net* comprises only *sporadic tasks*. Tasks are represented by super-transitions or yet by simple transitions, depending on the desired hierarchical representation (see section 3.3.5).

3.3.4.1 Periodic Transitions:

Each *periodic process net* has a *periodic transition*. A periodic transition fires at regular time intervals and has no input places. Periodic transitions govern the periodic execution of the associated process nets, and their presence in a process net specification characterises the nature (periodic or sporadic) of that process net. Because the definition of the triggering mechanism is external to tasks, a separation between functionality and time behaviour is achieved that makes it possible to reuse the task definition more easily in different triggering contexts.

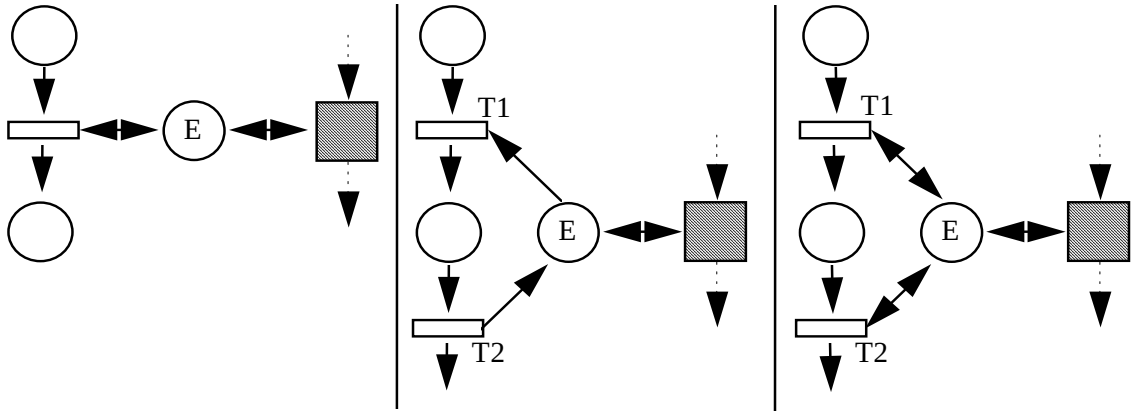
3.3.4.2 Input and Output Places:

Each *sporadic process net* starts with an *input place*. Its main purpose is to explicitly define the beginning of that sort of process net, and so facilitate not only visualisation by the designer but also the development of consistency check tools. *Output places* are used as an output medium between the design and the external environment, when the user wants to emphasise that the token put in that place should not be overwritten, as it would happen if a *store* was used, as it will be explained later. Both input and output places help define the boundaries of process nets and semantically they are similar to common places.

3.3.4.3 Mutual Exclusion:

In order to facilitate not only consistency analysis but also task inter-dependencies representation, another special node called *mutual exclusion* was created. As its name says, it is used to represent mutual exclusion among tasks so that all tasks connected to a same mutual exclusion node share a common resource and so cannot operate in parallel.

Figure 3.4.a shows an example of the correct utilisation of the construct. Figure 3.4.b shows an structure which is not allowed due to problems it would bring to the scheduling algorithm. The solution is then to merge tasks T1 and T2 into a single task and use an approach similar to Figure 3.4.a or to Figure 3.4.c, if it is possible.



a) Consistent specification b) Inconsistent specification c) Consistent specification

Figure 3.4. Mutual exclusion representation

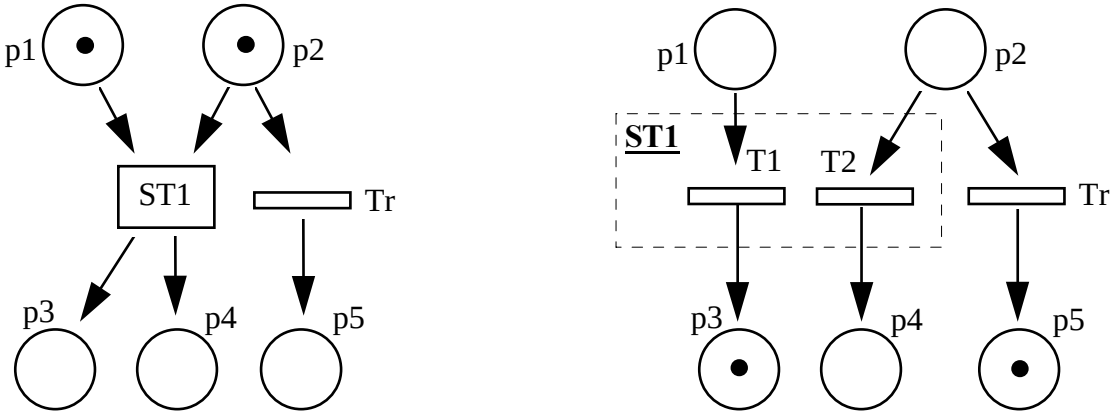
3.3.5 Hierarchy

As shown in Chapter 3, Petri nets have been extended to allow for more concise representations which would permit a better handling of complex system specification and analysis. In this direction, two main extensions were proposed: Coloured Petri Nets (CP-nets) [66] and Predicate/Transition Nets (PrT-nets) [46], which can be considered as "two slightly different dialects of the same language" [67]. These extensions rely basically on folding techniques in order to reduce design complexity. However, folding does not provide abstraction since all parts of the design remain explicitly defined in the specification, just in a more concise way. Also, as pointed out in [62] "folding is well suited, when we have exactly identical objects, but less suited for asymmetrical arrangements".

Special hierarchy constructs were introduced for CP-nets [62] in an attempt to overcome the limitations in complexity handling. Nevertheless, these constructs are better tailored for modularisation purposes than for handling analysis capabilities in complex systems. Consider the design shown in Figure 3.5.a below. The subnet associated to the *Substitution Transition* ST1 (as it is called in [62]) is shown in Figure 3.5.b.

Although this hierarchy may work as a visual aid, it cannot be directly used if an analysis of the system behaviour is required, since the reachable markings at the higher abstraction level (when the ST1 internal subnet is disregard) is different from the markings that can be reached when the subnet description is used instead of the transition. For example, if the net above is analysed considering ST1 as a simple transition, it is

impossible that the marking shown in Figure 3.5.b occurs, although it becomes possible when the subnet specification is considered.



a) ST1 as a single transition. Marking before firing.

b) ST1 subnet. Marking after firing.

Figure 3.5. CP-nets hierarchy

3.3.5.1 Super-Transitions

In order to overcome those problems, a possible solution is to consider that all subnet inputs occur at once and the same is also true for its outputs. Although this may seem restrictive, it is exactly the same consideration used in programming languages for functions and procedures. In this work the structure used to provide this hierarchy capabilities is called *super-transition*. When it comes to real-time modelling, those structures are equivalent to the ones used in other real-time design approaches (e.g. see [72, 120]) for encapsulation of temporal behaviour. It is common practice to abstract from the inner details of processes and concentrate specifically on the timing constraints associated with each process as a whole. If a process internal time constraint is important, then it cannot be abstracted from and should be explicitly represented at a higher hierarchical (abstraction) level. In this sense, the highest hierarchy level (which here is called Scheduling Level, as explained in section 3.2) is composed of a set of objects (in this work transitions and super-transitions) with each one having a defined worst-case execution time and resource amount needed for execution.

Loops are also expressed by using super-transitions as well, as shown in section 3.4.2. Recursion is not allowed and loop iteration is bounded in order to allow for worst-case estimations. Other parallel languages (e.g. Occam) also rely on limitation of recursion

and non-determinism of loop constructors in order to have a better knowledge of the maximum resources needed during program execution.

If it is necessary to specify a design that cannot fit into the restrictions imposed by super-transitions, e.g., the need of specifying several input or output points at different places (as in the example of Figure 3.5), then that design cannot be abstracted and so cannot take place in a hierarchical construct, and should be explicitly specified at the Scheduling Level.

Tasks are represented either as simple transitions or as *super-transitions*. These special transitions have associated subnets which are characterised by having two special transitions called *InputTransition*, and *OutputTransition*, and a special place called *ControlPlace*. Super-Transition subnets are safe nets (assured by the ControlPlace), i.e., it is not possible to have more than a token in any place at any time. All synchronous communication between internal and external media goes through the InputTransition and OutputTransition (see Figure 3.6). The use of ControlPlaces are not exactly necessary to keep the subnets safe in the approach used in this dissertation, since only a task deadline smaller or equal to its period is allowed, which guarantees safeness. However, they are used to remind the user of the non-reentrant nature of those subnets, and also because it is desired to use this work in the specification of other kind of system which may require the removal of this limitation (e.g. in designs where it is allowed to have task overloads for small periods of time).

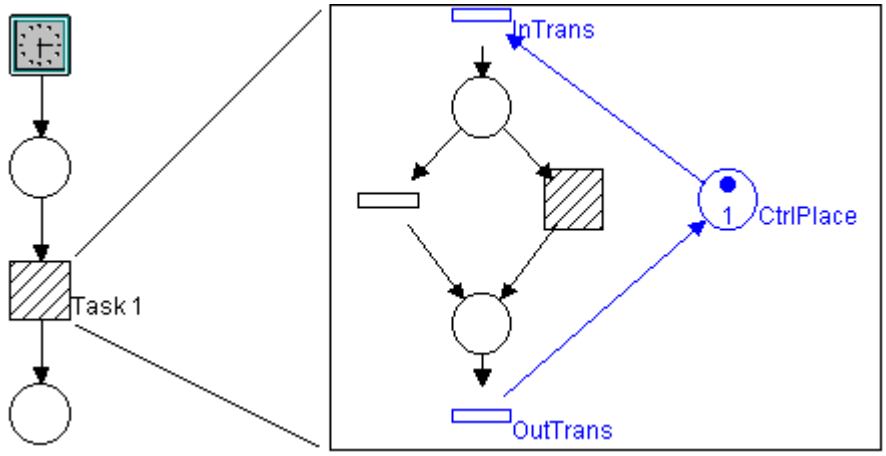


Figure 3.6. Hierarchical modelling: a super-transition and its subnet

Each super-transition also has an associated transition specification with the purpose of providing descriptions that are functionally equivalent to the super-transition

subnet. Therefore, super-transitions can be replaced by their functional and time equivalent transitions, and carry on analysis over the collapsed Petri Net, such as a multi-level simulation. This feature also permits to attach different subnets to the same functional specification, which makes it possible to try different implementation alternatives for the same super-transition. During analysis, the designer chooses whether to use either the subnet specification or its associated transition specification.

Super-transitions may have nested super-transitions. This allows for a reasonable decrease in the number of states to analyse since these nested super-transitions may also be collapsed into their functionally equivalent transitions whenever necessary. This hierarchical capability also permits the use of top-down or bottom-up design approaches, or yet a combination of both.

3.3.5.2 Stores

In order to provide a higher level of abstraction, another kind of node called a *store* was devised which is based on places. However, differently from a place, a store cannot carry more than one token at a time. Every time a task wants to send data to a store the store's previous data is removed. Also, operations over stores take a negligible amount of time. Therefore, a store acts as a device that contains data that may be overwritten when new data arrives, and perform asynchronous communications. By being safe, stores do not overload any task by accumulating data for computation. In other words, places trigger the transitions execution, since anytime a token arrives in a place it should be consumed as soon as possible, whilst stores carry information that is retrieved only when needed.

The non-blocking communication model for hard-real-time tasks provided by stores is in agreement with other works in the area (e.g. [4]), allowing for the decoupling of process nets in order to provide a more predictable time behaviour.

Stores can also be used to carry information between transitions internal to super-transitions and external transitions without passing through the InputTransition and OutputTransition, as shown in Figure 3.7.

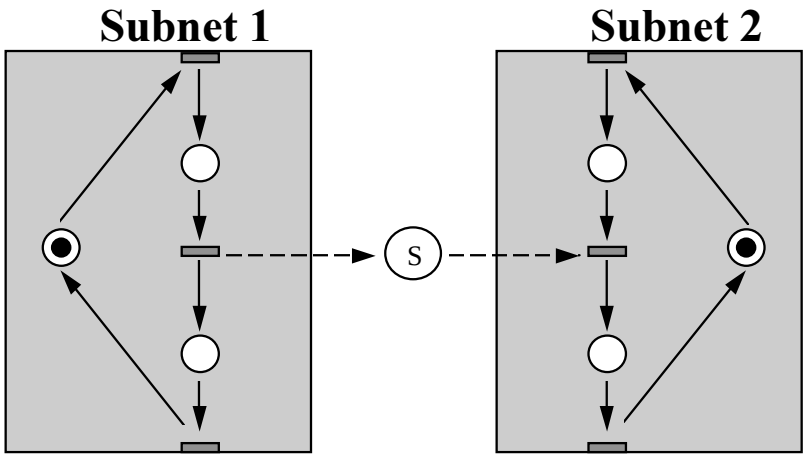


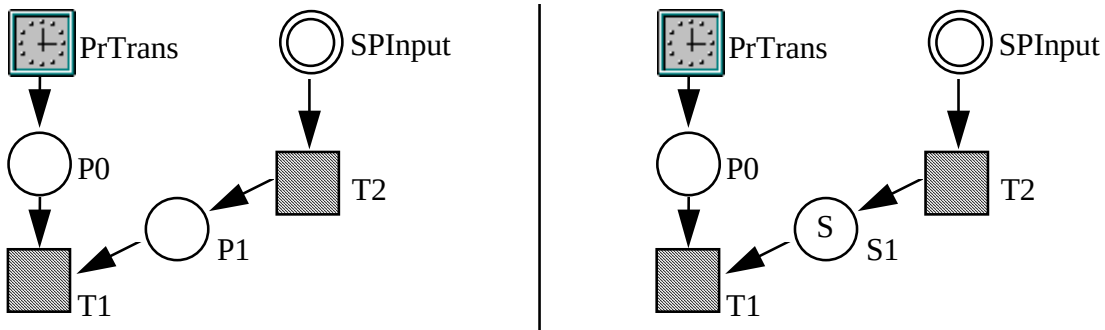
Figure 3.7. Using stores for subnet communication

3.3.6 Design Consistency

A process net specification is considered inconsistent if its triggering mechanism is not properly defined. For example, a periodic process net cannot receive data from other process nets through places. Due to the Petri nets firing rules, the status of a place (empty or not) defines transition firing conditions. So, a place connecting one net to another could interfere in the latter net execution timings by acting as a triggering control. Therefore, periodic tasks can only receive data from tasks belonging to other process nets through stores (since they cannot interfere in the firing rules) whereas sporadic tasks may also receive data sent directly to their InputPlaces.

Stores play a major role in the definition of the control flow among process nets. They decouple the communication among process nets leading to a better definition of their triggering mechanism.

For example, in Figure 3.8 transition *T1* is being driven by both places *P0* and *P1*. Hence it is required that both have tokens in order to enable *T1* to fire. So, *T1* is dependent on both events, the firing of the Periodic Transition *PrTrans* and the arrival of a token in the InputPlace *SPinput*, and not only on the firing of *PrTrans*, as it would be desirable, and constitutes an inconsistent design. This sort of error can be easily spotted by automated consistency check tools.



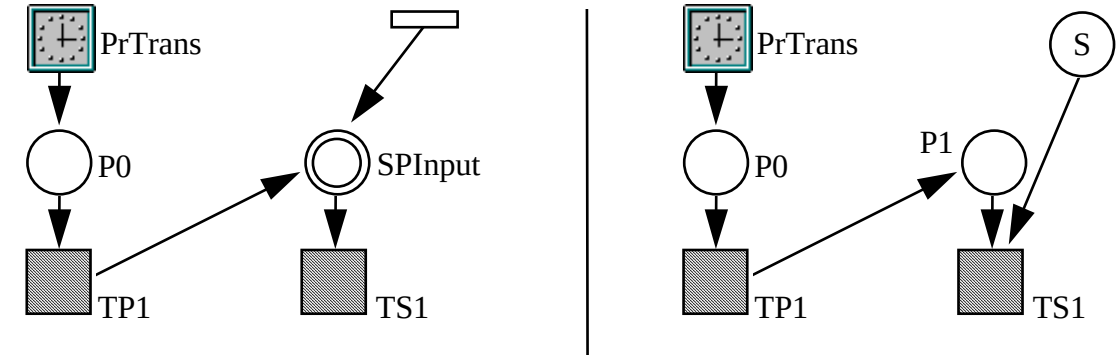
a) Inconsistent specification

b) Consistent specification

Figure 3.8. Periodic-sporadic communication consistency

The consistency checks become tighter as the design process moves towards the production of the scheduling level output. All nets at the scheduling level need to be acyclic when Mutual Exclusion places and Stores are disregarded. Furthermore, the specification must be completely deterministic, i.e., all computation times are based on worst-case scenarios and no conditionals are allowed.

Figure 3.9.a shows a situation in which a periodic task sends data to a sporadic task via an input place. This structure is acceptable during the initial specification stages. However, at the scheduling level only periodic tasks are allowed and so the net would have to be transformed. Figure 3.9.b shows a possible transformation which leads to a consistent specification at the scheduling level.

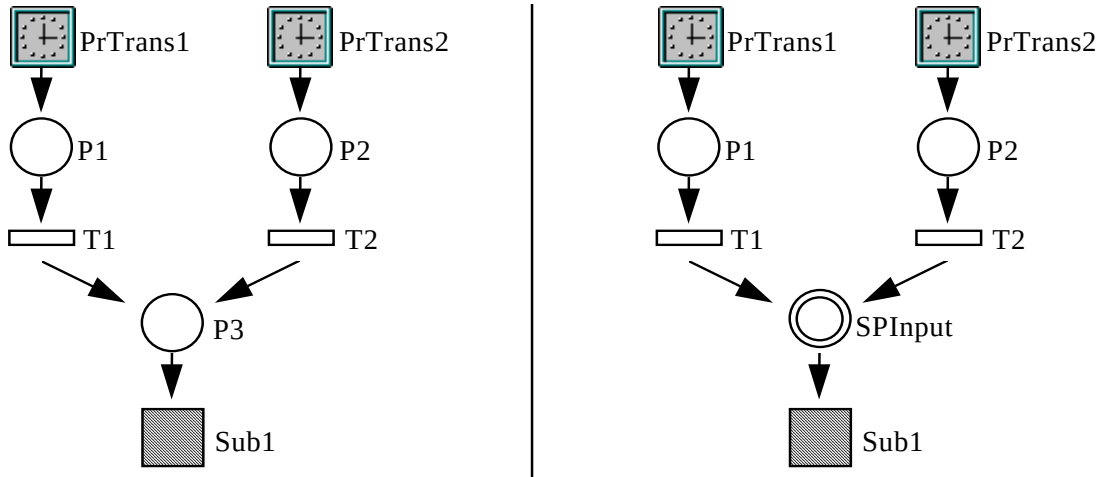


a) Although this specification is accepted at the initial level, it is considered inconsistent at the scheduling level

b) Consistent specification

Figure 3.9. Subnets consistency and transformation

Figure 3.10.a shows a design inconsistency since place P3 is being driven by both transitions T1 and T2, which belong to distinct process nets. There are many possible ways of correcting this inconsistency, e.g. the one shown in Figure 3.10.b.



a) Inconsistent specification

b) Consistent specification

Figure 3.10. Dual triggering inconsistency

3.3.6.1 Sporadic Tasks Transformation

As explained in Chapter 2, sporadic tasks are those which may request execution at any time, although a minimum inter-arrival time (*minp*) between consecutive invocations exists. Since the scheduling algorithm proposed in this dissertation do not deal with sporadic tasks, those need to be converted to periodic tasks. Mok has shown that a sporadic task *s* represented by ($minp_s$, c_s, d_s) can be replaced by a periodic one *p* represented by (p_p, r_p, c_p, d_p) as long as the following conditions are satisfied [92] (for an explanation about these tuples, see Section 2.4.1):

1. $c_s \leq d_p \leq d_s$
2. $c_p = c_s$
3. $p_p \leq d_s - d_p + 1$

Suitable transformation rules are $r_p = 0$, $c_p = c_s$, $p_p = \min(minp_s, d_s - c_s + 1)$, $d_p = c_s$, if $d_s \geq 2.c_s$, and $c_p = d_p = p_p = c_s$ otherwise [101]. This transformations are not unique but are optimal in the sense that no other transformation exists that would provide a period greater than p_p [92].

3.4 Time Wizard

Time Wizard is a design tool which uses the model proposed in former sections. It takes advantage of the extended constructors proposed to provide automatic consistency checks.

Its main features are:

- **Requirements capture:** timing constraints and inter-task dependencies can be easily expressed by means of dialogue boxes and graphical representations.
- **Validation and verification:** Simulation and consistency check tools are available for verification and assessment. Profiling tools can be used for the improvement of worst-case bounds.
- **Graphical interface:** Time Wizard follows the trend in the use of graphical formalisms for specification to facilitate the design and analysis process, and provide a neutral specification approach.
- **Predicate and Action Specification Language:** In order to define these features, C++ is being used since there is a strong link between Time Wizard and Cabernet, and the later uses C++. Cabernet is being used as a simulation tool for designs specified in Time Wizard. An automatic translator is built in Time Wizard which performs the necessary model transformations in order to allow it to be simulated in Cabernet. In the future, when Time Wizard has its own simulator built in, a formal language, such as Occam, is intended to be used in place of C++.

3.4.1 Design Flow

The design starts by defining the places and tokens data classes (in C++). Data classes are defined in a special window which keeps information about the classes interfaces and methods (or member functions, as they are called in C++) implementations. All classes have to be defined using a template called *TIMED_CLASS()* which is used to incorporate a special variable used to represent time (see Section 3.3.3 and Appendix A). TIMED_CLASS() creates a new class based on the class used as parameter. For example, Figure 3.11 shows a Time Wizard window used to define a very basic class called *NKvoid*. All tokens must have a class and so NKvoid is useful when tokens with no specific values (e.g. integers or floating point variables) are required. After using TIMED_-CLASS(NKvoid) and TIMED_CLASS(int), two new classes name *timed_NKvoid* and *timed_int* are created. These classes carry the same information as their basic classes plus an extra variable named *time* which is used to represent the time when a token was produced after a transition firing. Since all *places* have associated data types (see Section 3.3.1), timed_NKvoid is the default class associated by Time Wizard to every new place.

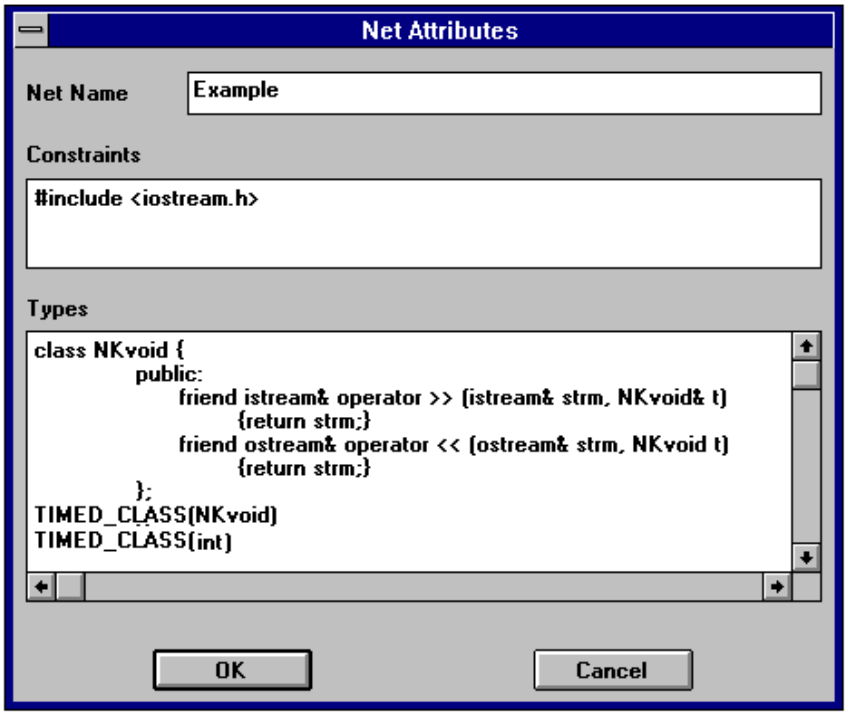


Figure 3.11. Class definitions

3.4.1.1 Processors Definition

The second design step is the initial processor set definition. At this stage the user only needs to specify the processors names and implementation, either software or hardware. The implementation paradigm information is used later during partitioning, since all tasks that are initially allocated to hardware processors cannot be moved to software by the partitioning tool. Figure 3.12 shows an example of processor definition where there are two software hosts named DSP1 and GP1 and one hardware processor named FPGA. Since DSP1 is selected, its implementation paradigm (software) is shown in the right-hand side of the figure below.

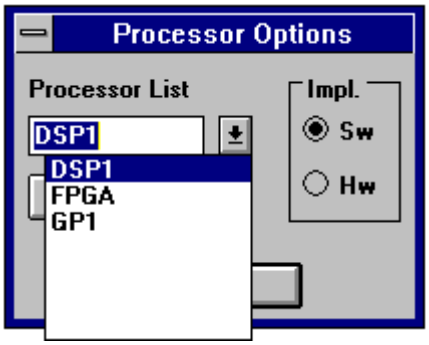


Figure 3.12. Processors definition

3.4.1.2 Design Specification

When starting a design the user is presented with two main options: to design a net or a subnet. The basic difference is that a subnet starts with a default structure which is

automatically placed by Time Wizard and which cannot be removed by the user. This structure consists of the InputTransition, OutputTransition, and ControlPlace primitives mentioned in Section 3.3.5.1, which have the function of guaranteeing the safeness property of a subnet, i.e., its non-reentrant nature (see Figure 3.6 in Section 3.3.5.1 for an example).

The designer can use the drawing buttons present in Time Wizard in order to produce a system specification, as in Figure 3.13 (the design shown has some inconsistencies that will be used later for illustration purposes). There are many auxiliary options, such as zoom in and out, drawing grid, and print preview. The definition of each primitive's attribute (such as transitions release times and deadlines, or places data types) are performed by double clicking over the desired object, which opens the respective attribute window.

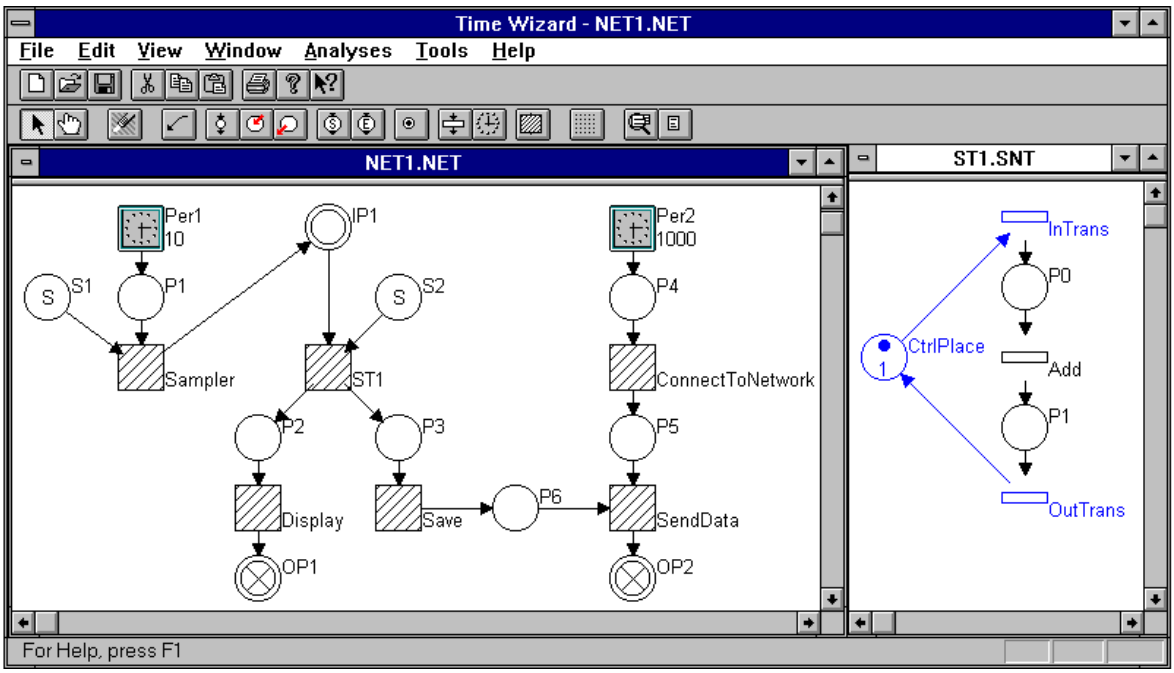


Figure 3.13. Time Wizard main window showing an initial stage design

For example, Figure 3.14 shows the transition attributes window associated to super-transition ST1 shown in Figure 3.13. This window includes the transition's name, internal identifier, firing predicate and action (in this case it uses tokens coming from places IP1 and S2, and produces tokens to place P2 and P3), release time, software and hardware WCETs, deadlines, allocated processor and implementation. All data is input by the designer. Although Time Wizard provides tools to help the user estimate the WCET of

process nets and subnets (see section 3.4.2), it does not produce this information automatically.

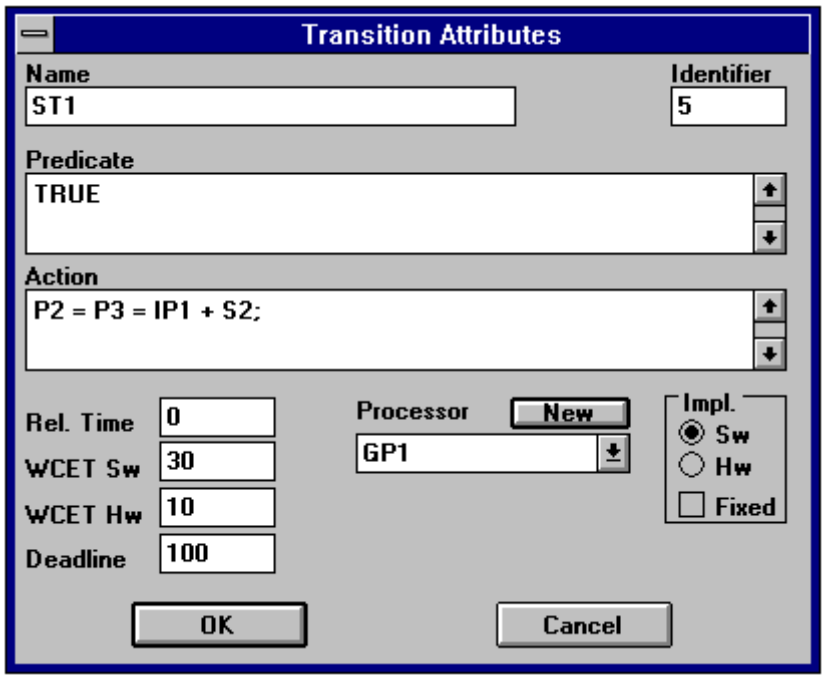


Figure 3.14. Transition attributes window

Super-transitions, as explained before (see Section 3.3.5.1), have associated subnet and simple transition descriptions. These descriptions are equivalent (although this equivalence is not validated formally). Textual descriptions are used, for example, when the designer wants to simulate the system without regarding the internal details of a super-transition. In this case, the super-transition is replaced by a simple transition which uses the super-transition textual definition with its predicate and action. Figure 3.15 shows the attributes window of a super-transition named ST1. Its associated textual description is stored in its transition equivalent, which was already shown in Figure 3.14.

Some parameters are similar to those found in simple transitions, such as name, internal identifier, action and predicate. However the *action* parameter is used here to create a link between the formal and the actual parameters used when a super-transition fires and data (tokens) have to be taken form outside to inside the super-transition subnet, and vice-versa. The names in the *formal parameters* list are used inside the subnet. For example, consider again the simple design shown previously in Figure 3.13 which includes super-transition ST1 and its associated subnet. If the designer wants to simulate the system using ST1's transition equivalent, the option *use transition attribute for simulation* in ST1's attributes window should be selected, which would cause the generation of a file

which disregards ST1's subnet details and uses its associated transition specification. However, if the designer opts to simulate in a more detailed level and use the associated subnet (i.e. not selecting the option *use transition attribute for simulation*), Time Wizard uses ST1's formal parameters to link ST1's subnet to the net where it is embedded. In systems with many super-transitions or with nested super-transitions the user may choose different abstraction levels to different super-transitions during simulation.

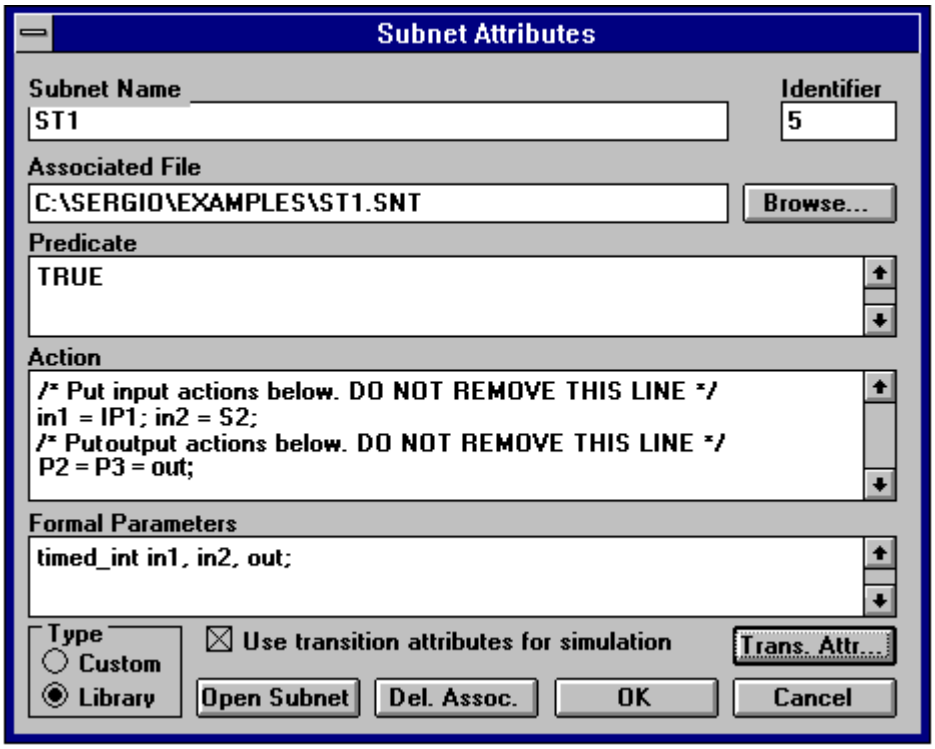


Figure 3.15. Super-transition attributes window

3.4.1.3 System Profiling and Simulation

After the initial specification is done by the designer, the system may be simulated in order to gather more information about the system's dynamics and temporal requirements. The technique proposed here for system profiling is based on the premises of reactive systems [14], in which the only behaviour of interest is that related to its responsiveness to environment demands. So, the surrounding environment is modelled as timed nets while it is considered that the system under development is being executed over a computer architecture with infinite computation capacity, i.e., all task execution times are zero to all tasks that are part of the system. The only system transitions that keep their execution times are periodic transitions.

Consider that the designer wants to simulate the example shown in previous sub-sections using ST1's subnet description. Then, using the option *Export to Cabernet* (under the FILE menu) Time Wizard automatically creates a flat net where ST1 is replaced by its subnet and all other subnets are replaced by simple transition equivalents containing textual descriptions of their intended behaviour. Figure 3.16 shows the resulting flat net which is generated automatically by Time Wizard (some objects were moved from their original position to promote a better understanding).

In order to consider the infinite computation capacity mentioned before, the execution times are removed from all transitions except periodic ones[2]. As explained in Section 3.4, Time Wizard does not incorporate a simulator but uses the one found in the Cabernet environment instead. So, the flat net description is translated to a format that can be understood by Cabernet, where the simulation occurs. Cabernet was modified in order to accommodate a simulation recorder which stores all data generated during simulation for later analysis.

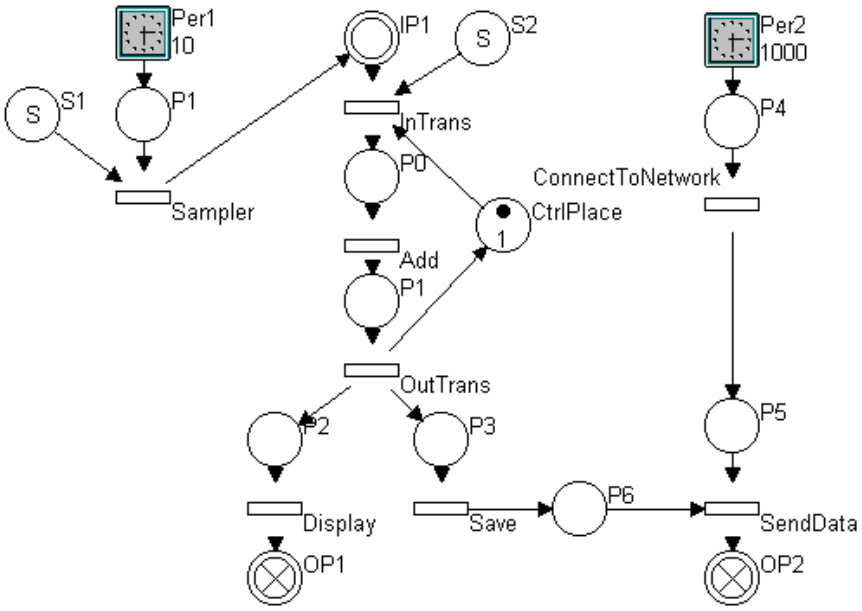


Figure 3.16. Flattened specification

3.4.1.4 Consistency and Refinements

Time Wizard is tailored to produce an output that can be used by the scheduling and partitioning tool to be presented in Chapters 4 and 5. However, not all system

[2] The user can override this removal for simulation purposes.

specifications are suitable to generate such an output. Although currently refinements are done manually by the user, Time Wizard has an analysis tool based on the consistency rules defined in Section 3.3.6 which may help the designer to adapt and refine the design towards the production of a suitable format. It searches all InputPlaces and Periodic Transitions and verifies if there is any net node, other than stores and InputPlaces, which is connected to more than one process net. The algorithms used for consistency checking are based on a simple depth-first traversal algorithm.

For example, if the consistency check tool is used on the design shown in Figure 3.13, the result would be the following warnings:

1. "The node named *SendData* is inconsistent. It is connected to the nets starting in *Per2* and *IP1"*.

2. "The node named *IP1* starts a sporadic net. When generating scheduling data, all sporadic nets must be transformed to periodic nets" (this warning is only generating when the user wants to produce scheduling data, since sporadic nets are only considered inconsistencies at the scheduling level).

These warnings help the designer to spot any structure that would prevent the production of a proper output. Figure 3.17 shows a possible solution to avoid the above warnings and from which is possible to produce data for the scheduler and partitioner.

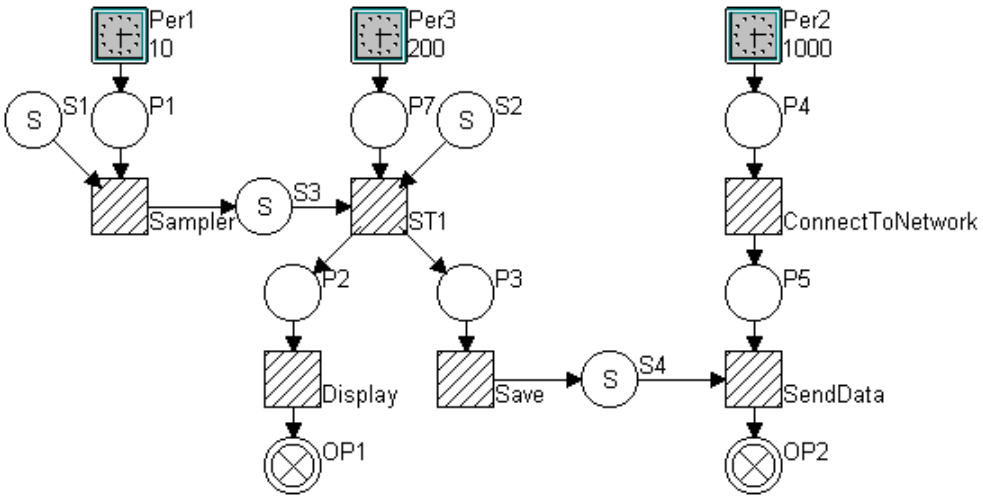


Figure 3.17. Consistent specification

3.4.2 Additional Features: Intra-Task Assessment

Much work has been done for worst-case analysis, such as [50, 88]. Time Wizard does not attempt to produce worst-case execution time estimations and relies on information generated externally. Nevertheless, Time Wizard provides tools to help closing the gap between average and worst-case computation times, in order to avoid over-allocating the processor to tasks that in the general case will not use all the allocated resources. The less this difference is the more efficiently the processor is used in a hard real time system implementation.

In order to do this, Time Wizard uses simulation data in order to find branching correlations and maximum number of loop iterations. It gives a good understanding of tasks' dynamic behaviour, and as such allows for refinements of the system with more knowledge than by using only static information. This is not a perfect solution since simulation results always depend on the input data set, which for large systems normally does not cover all possibilities. Other techniques have been proposed, e.g. [1], which uses symbolic execution to avoid false paths. It provides a trade-off between analysis results and analysis efforts and as such does not produce exact results. Integer linear programming (ILP) and user annotations have also been used [88]. However, these technique suffer from the huge complexity when dealing with large systems which is also the problem when trying to find exact solutions. The technique used in Time Wizard gives some insight into the problem and yet is very simple.

As an example of its use, consider that there is some dependency between the two conditionals shown in Figure 3.18 such that only the paths PT_1={inTrans, T1, T3, T4, outTrans} and PT_2={inTrans, T2, T3, T5, outTrans} can occur. Now consider $e_{t1} = e_{t5}$, $e_{t2} = e_{t4}$, and $e_{t1} >> e_{t2}$, where e_i is the execution time of transition (or path) *i*. It is clear, from the above assumptions, that the worst case execution time of the task as a whole is e_{PT1} (which is equal to e_{PT2}). Time Wizard tries to find the possible paths through simulation and these can then be used in the calculation of the worst case computation time. If only the graph description had been considered, then the paths PT3 = {inTrans, T1, T3, T5, outTrans} and PT4 = {inTrans, T2, T3, T4, outTrans} would also be regarded as possible. In this case, the worst case execution time would be e_{PT3}, which is equal to $e_{PT4,}$ and is much greater than e_{PT1}.

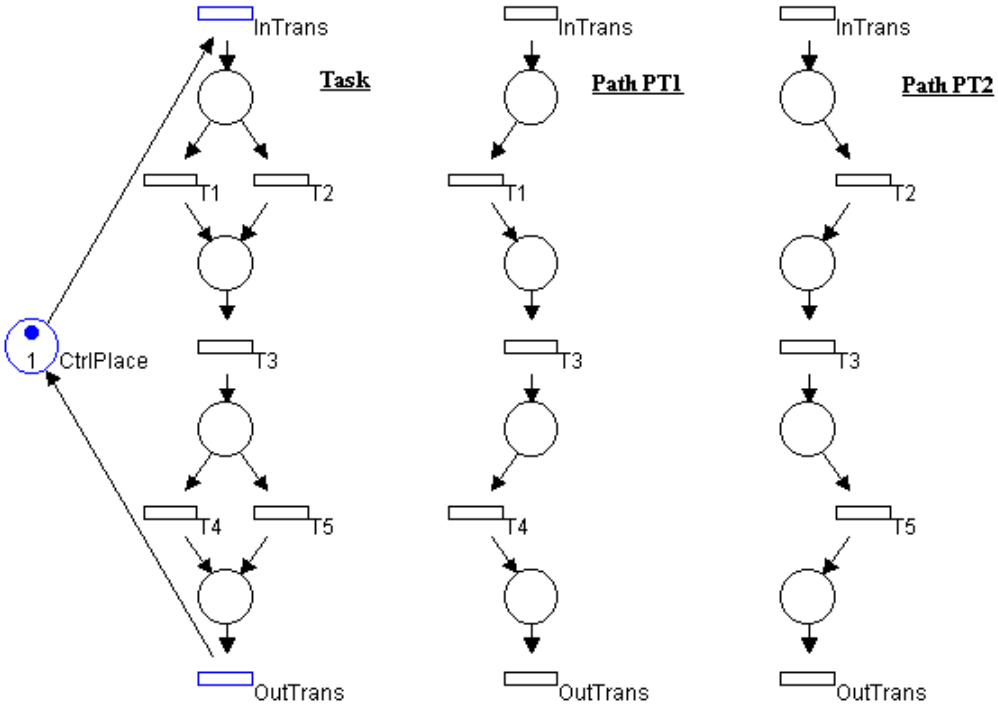


Figure 3.18. Path tracing analysis

Figure 3.19 shows another simple example in which two parallel conditionals are analysed. Transitions T0, T1, T2, and T3, fire depending on their predicates. The design is automatically exported to Cabernet's simulation tool, and a specially designed tracing tool which was incorporated to Cabernet is used in order to perform profiling analysis of the nets (see Appendix A). A tracing file is generated which is read and analysed by Time Wizard in order to produce a Firing Correlation Matrix, which relates the number of times two transition fires at the same cycle, as depicted in Figure 3.20.

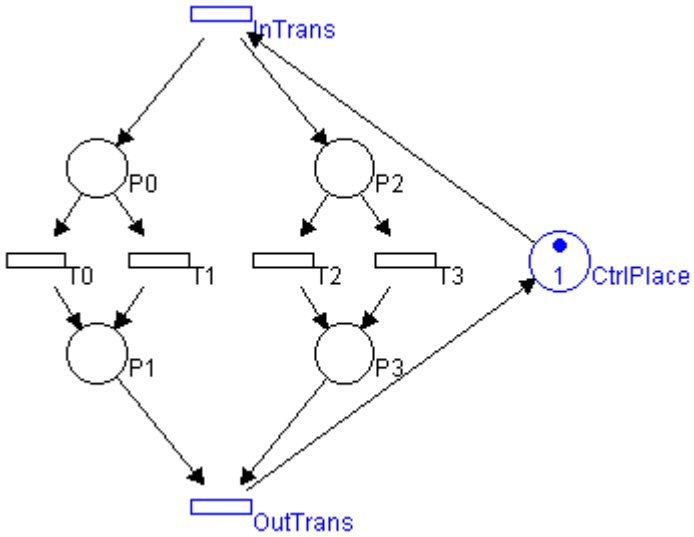


Figure 3.19. Conditional tracing analysis

```
Results of the Firing Correlation Analysis

Net name:
Net file: D:\SERGIO\PROJETO\EDITPN\TESTE\ANATEST1.NET

Control Transition: InTrans

Place:   P2
         Transitions:
            T2 fired 11 times
            T3 fired 13 times

Place:   P0
         Transitions:
            T0 fired 14 times
            T1 fired 10 times

Correlation Matrix:
         T0    T1
  T2      8     3
  T3      6     7
```

Figure 3.20. Firing Correlation Analysis

This ability of exporting nets to Cabernet enables the designer to use the analysis capabilities present in that environment. For more details on the sort of analysis available in Cabernet please refer to [112].

A tentative bound for the number of iterations of loops may also be found by Time Wizard. Although all graphs connected by places are acyclic (excluding the control place of tasks), loop representations are an exception. Loops are represented as shown in Figure 3.21. The maximum number of iterations is defined as the maximum rate of firings between the transitions TL and inLoop. Since simulation may not cover all possible cases, the final decision of accepting or not the analysis results produced by Time Wizard remains up to the designer.

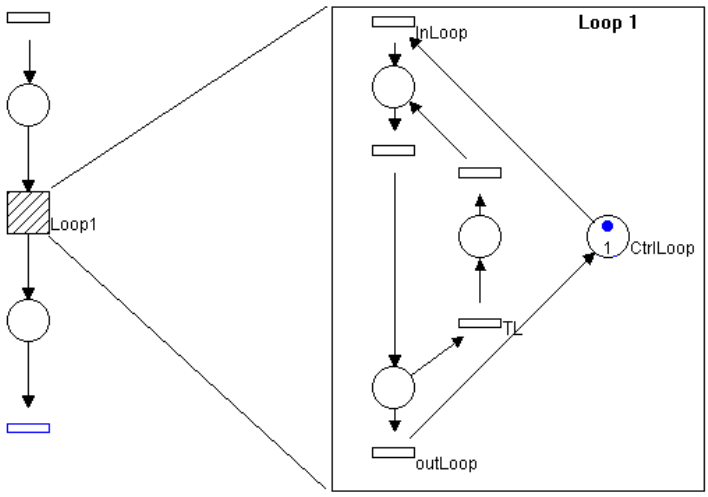


Figure 3.21. Loop iteration analysis

3.5 Summary

This chapter has shown a brief overview of a new co-design process for the development of hard-real-time embedded applications. Other considerations of the method as a whole are drawn in Chapter 7.

A co-specification methodology was also presented as well as the CAD tool that encompasses it. It has shown to be quite simple and yet it provides mechanisms for design assessment at both internal and task interconnection levels. Other examples of its use can be seen in Chapter 6.

It is the first time that Petri nets is used as a specification medium from the initial specification to the output level in the co-design field. It is also the first time a Petri net methodology is used to help the designer specify real-time systems in a format suitable to be used by the major scheduling algorithms, mainly pre-run-time schedulers.

The main difficulties in using Petri nets not only for analysis purposes but at all specification abstraction levels rely mainly on the lack of hierarchical, and modular constructs found in Petri nets, which lead to highly complex specifications when moving down the abstraction level. For the purposes of real-time systems, the lack of temporal and data representation is also paramount. These difficulties were overcome by using a high-level timed Petri net as the basic formalism in conjunction with extra primitive constructors which allowed for better hierarchical specifications and analysis tools.

The use of a single language for initial and internal representations permits a better understanding and control of the design process by the user. In systems with separate representations either the inner details are hidden, or the internal representation may be awkward and difficult to be understood by the designer. Also, the designer would need to know different modelling paradigms to follow the design process.

Although Time Wizard was initially thought for using as part of the co-design environment presented in this dissertation, it can be used separately as an aid for real-time design in general, providing outputs which can be used as input for schedulers in general, and mainly for pre-runtime schedulers, such as the one devised by Xu and Parnas [119].

Time Wizard does not provide automatic refinements and so does not attempt to transform sporadic nets to periodic ones automatically. However, this could be easily accomplished by using the transformations shown in Section 3.3.6.1.

The Petri Net extensions proposed here have the intent of increasing the usability of Petri Net-based design and analysis in practical real-time designs, and promoting the development of automatic tools. At the same time they provide hierarchical modelling while keeping the analysis capabilities found in TER nets.