TECHNICAL REPORT

ertCPN: The adaptations of the coloured Petri-Net theory for real-time embedded system modeling and automatic code generation

Wattanapong Kurdthongmee

Abstract

Kurdthongmee, W.
ertCPN: The adaptations of the coloured Petri-Net theory for real-time embedded system modeling and automatic code generation
Songklanakarin J. Sci. Technol., 2003, 25(3) : 381-394

A real-time system is a computer system that monitors or controls an external environment. The system must meet various timing and other constraints that are imposed on it by the real-time behaviour of the external world. One of the differences between a real-time and a conventional software is that a real-time program must be both logically and temporally correct. To successfully design and implement a real-time system, some analysis is typically done to assure that requirements or designs are consistent and that they satisfy certain desirable properties that may not be immediately obvious from specification. Executable specifications, prototypes and simulation are particularly useful in real-time systems for debugging specifications. In this paper, we propose the adaptations to the coloured Petri-net theory to ease the modeling, simulation and code generation process of an embedded, microcontroller-based, real-time system. The benefits of the proposed approach are demonstrated by use of our prototype software tool called *ENVisAge* (an Extended Coloured Petri-Net Based Visual Application Generator Tool).

Key words : real-time system, embedded system, modeling tool, automatic code generation, coloured Petri-net

Ph.D. (Computer Science), Division of Computer Engineering, Institute of Engineering and Resources Walailak University, Tha Sala, Nakhon Si Thammarat 80160 Thailand.
Corresponding e-mail : kwattana@wu.ac.th
Received, 13 January 2003 Accepted, 26 March 2003

บทคัดย่อ

วัฒนพงศ์ เกิดทองมี

ertCPN: เครื่องมือสร้างโมเดลและโปรแกรมอัตโนมัติสำหรับระบบ real-time ขนาดเล็กตามทฤษฎีส่วนปรับปรุงของคัลเลอร์เพทริเน็ต

ว. สงขลานครินทร์ วทท. 2546 25(3) : 381-394

ระบบ real-time เป็นระบบคอมพิวเตอร์อีกระบบหนึ่งที่มีหน้าที่หลักในการอ่านและควบคุมค่าของตัวแปรภายนอกให้เป็นไปตามเงื่อนไขการทำงานที่กำหนดโดยโปรแกรมงานของระบบ การทำงานของระบบควบคุมชนิดนี้จำเป็นต้องเป็นไปตามเงื่อนไขและกฎเกณฑ์บังคับทางด้านเวลาอื่น ๆ ที่ถูกสร้างขึ้นมาเพื่อกำหนดพฤติกรรมการทำงานของระบบให้ดำเนินอย่างถูกต้อง ความแตกต่างที่ชัดเจนระหว่างโปรแกรมงานของระบบคอมพิวเตอร์ทั่วไป ๆ กับระบบ real-time คือ โปรแกรมคอมพิวเตอร์ในกลุ่มหลังนั้นจำเป็นต้องมีความถูกต้องทั้งในด้านหน้าที่และเวลาของการทำงาน ดังนั้นในการออกแบบระบบ real-time จึงจำเป็นต้องมีการวิเคราะห์อย่างละเอียดเพื่อให้ได้มาซึ่งคุณสมบัติการทำงานของระบบที่ครบถ้วนในทั้งสองด้าน ในบทความนี้ผู้เขียนจะได้นำเสนอแนวทางของการปรับปรุงทฤษฎีคัลเลอร์เพทริเน็ต (coloured Petri Net-CPN) เพื่อให้ง่ายต่อการนำมาประยุกต์ใช้ในการสร้างและจำลองการทำงานของโมเดลของระบบ real-time ขนาดเล็ก และตอบสนองต่อการแปลงโดยอัตโนมัติจากโมเดลที่ถูกต้องและสมบูรณ์ให้เป็นโปรแกรมของระบบควบคุมที่สามารถนำไปใช้ในระบบจริงได้ นอกเหนือจากการนำเสนอทฤษฎีที่ได้รับการปรับปรุงของ CPN แล้วผู้เขียนยังได้นำเสนอซอฟต์แวร์คอมพิวเตอร์ชื่อ *ENVisAge* (an Extended Coloured Petri-Net Based Visual Application Generator Tool) ซึ่งเป็นซอฟต์แวร์ต้นแบบที่อิงทฤษฎีส่วนขยายของ CPN

แผนกวิชาวิศวกรรมคอมพิวเตอร์ สำนักวิชาวิศวกรรมและทรัพยากร มหาวิทยาลัยวลัยลักษณ์ อำเภอท่าศาลา จังหวัดนครศรีธรรมราช 80160

A real-time embedded system is a small, less powerful and less complex computer system comparing to a general purpose computer system (Shaw, 2001). Normally, it is controlled by a single 8/16 bits microcontroller/microprocessor and has been used in monitoring, responding to, or controlling an external environment. This environment is connected to the system through sensors, actuators, and other input-output interfaces. The system must meet various timing and other constraints that are imposed on it by the real-time behaviour of the external world to which it is interfaced. Real-time software differs significantly from conventional software in a number of ways. First, a program must not only produce the correct answer or output, but it must also compute the answer "on time." In other words, a program must be both *logically* and *temporally* correct.

A second distinguishing feature of real-time systems is concurrency. Real-time systems must deal with the inherent physical concurrency that is part of the external world to which they are connected. Systems design becomes especially difficult when one combines the problems of concurrency with those related to time; i.e. deadliness. A third major characteristic of real-time systems is the emphasis on the significance of reliability and fault tolerance, where reliability is a measure of how often a system will fail. Similarly, real-time computer hardware has different requirements from general purpose computer systems. Its run-time behaviour must be predictable, so that software designers can predict applications behaviour. Hardware must be reliable and fault tolerant, so that costly errors are prevented, when possible, and handled predictably, otherwise.

The software "*life cycle*" of a real-time embedded system, which is similar to a general purpose computer software, defines the stages in the life of a software system as it develops from its initial specification through its final deployment and use. It has been classified into five sequential

stages which are: requirements, design, implementation, testing and maintenance. Typically, some analysis is done to assure that requirements and designs are consistent, and that they satisfy certain desirable properties that may not be immediately obvious from the specifications. More often analysis is performed through computer testing of various executable forms of parts of the requirements or designs. Executable specifications, prototypes and simulations are particularly useful in real-time systems for debugging specifications. They can illuminate problems at an early stage for easy and inexpensive correction. In addition, they provide some idea of the run-time behaviour of the system, giving a basis for a designer to refine or change specification.

In an early stage of a real-time software development, a designer has to deal with the problem to decribe the system's behaviour to conform with its requirements. This is iteratively performed by use of either a descriptive or imperative language. Descriptive notations directly specify properties that must be satisfied, rather than directly generate behaviours with the desired properties. They are based on conventional mathematics. Some of the principal descriptive methods include: OBJ (Goquen *et al*., 1988) and Z-notation (Spivey, 1989). In contrast, imperative methods specify behaviours by giving algorithmic descriptions, that is sequences of instructions or actions to be taken by one or more agents, that generates the behaviours. Specifications are directly executable in that they translate easily into corresponding computer procedures. Many of the imperative notations are similar to or derived from programming languages. Examples are data flow diagram (DFD) (Guezzi *et al*., 1991), state machine, statecharts (Harel *et al*., 1990), modecharts (Jahanian *et al*., 1994) and variations of the Petri-net (Murata, 1989).

Within a group of imperative notations described earlier, a Petri-net (or Place/Transition net) offers an attractive graphical way to portray the required functions of a system. In addition, correct behaviours for many scenarios of interest can frequently be demonstrated through simulation because Petri-net, by itself, is an executable graphical/visual modeling language. Unfortunately, a real-time embedded application requires more than the standard elements provided by a Petri-net, or even its derivatives, in order to successfully and easily model the correct behaviours of the system.

The organization of this paper is as follows. In section 2 we give a theoretical overview of a Petri-net and its derivative: a coloured Petri-net. The proposed adaptations to a coloured Petri-net modeling language are explained in section 3. An implementation of a computer tool to assist modeling and automatic code generation for real-time embedded systems and conclusion are given in section 4 and 5, respectively. Finally, in section 6, we give a real-life example of utilizing a proposed extension to model a simple real-time embedded system.

An overview of a Petri-Net theory

In this section, we present an informal introduction to a Petri-net and its derivatives which are timed Petri-net, time Petri-net and coloured Petri-net. This paper focuses only on a coloured Petri-net which is mainly exploited in our modeling tool.

An informal introduction to a Petri-Net

Petri-nets are a graphical and mathematical modeling tool applicable to many systems (Murata, 1989). They are a promising tool for describing and studying information processing systems that are characterized as being concurrent, asynchronous, distributed, parallel, non-deterministic, and/or stochastic. As a graphical tool, Petri-net can be used as a visual–communication aid similar to flow charts, block diagrams, and networks. In addition, tokens are used in these nets to simulate dynamic and concurrent activities of systems.

A Petri-net, or PN for short, is a particular kind of directed graph, together with an initial state called the *initial marking, M_0*. The underlying graph *N* of PN is a directed, weighted, bipartite graph consisting of two kinds of nodes, called *places* and *transitions*, where arcs are either from a place to a transition or from a transition to a place. In graphical representation, places are

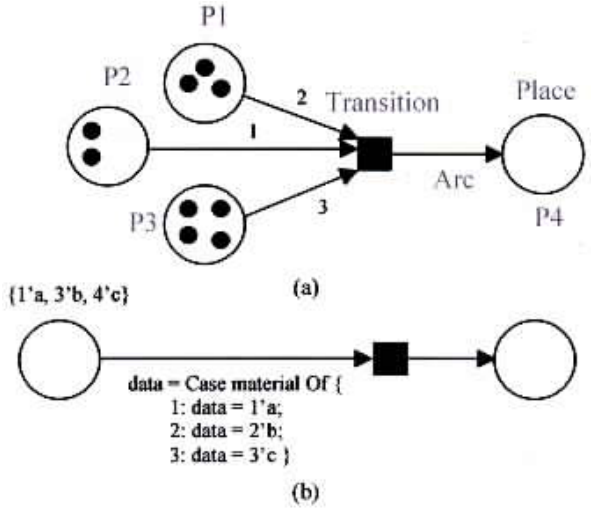

Figure 1. The CPN model in (b) is more compact than the PN model in (a) with similar behaviour

drawn as circles, transitions as bars of boxes as illustrated in Figure 1(a). Arcs are labeled with their weights (positive integers), where a k-weighted arc can be interpreted as the set of k parallel arcs. Labels for unity weight are usually omitted. A marking (state) assigns to each place a nonnegative integer. If a marking assigns to place p a nonnegative integer k, we say that p is marked with k-tokens. Pictorially, we place k black dots (tokens) in place p.

In modeling, using the concept of conditions and events, places represent conditions, and transitions represent events. A transition (an event) has a certain number of input and output places representing the pre-conditions and post-conditions of the event, respectively. The presence of a token in a place is interpreted as holding the truth of the condition associated with the place. In another interpretation, k tokens are put in a place to indicate that k data items or resources are available. Some applications may interpret input places, transition and output places as input data, computation step and output data, respectively. Figure 1a depicts an application of PN to model a product's assembly line with 3 raw materials. The number of black dots within each place is equal to the number of raw material of that type. In this case, each product needs to use 1, 2 and 3 of the first, second and third raw material, respectively.

The behaviour of many systems can be described in terms of system states and their changes. In order to simulate the dynamic behaviour of a system, a state or marking in PN is changed according to the following transition (firing) rule:

1. A transition t is said to be 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 p to t.
2. An enabled transition may or may not fire (depending on whether or not the event actually takes place).
3. A firing of an enabled transition t removes w(p, t) tokens from each input p to t, and adds w(p, t) tokens to each output place p of t, where w(p, t) is the weight of the arc from t to p.

A transition without any input place is called a *source transition*, and one without any output place is called a *sink transition*. A source transition is unconditionally enabled, and the firing of a sink transition consumes token but does not produce any.

An overview of a coloured Petri-Net

A Coloured Petri net (CPN) proposed by Jensen (1992) is a system modeling language which is an extension of a classical PN. It is designed by combining the strength of a standard PN with the strength of programming languages. A PN provides the primitive for describing synchronization of concurrent processes, while programming languages provide the primitives for the definition of data types and manipulation of their data values which are called model inscriptions. A CPN when used to model a system leads to models which are much more compact than the models drawn using elementary PN. The compactness of a CPN model for the same production line as in section 2 is demonstrated in Figure 1b. It is noted that only one input place is needed in this case. The place contains 1, 3 and 4 instances of raw material a, b and c, respectively. The arc's inscription tells us that the transition requires 1 of a, 2 of b and 3 of c raw materials in order to fire.

Informally, CPN is similar to PN which is state and action oriented at the same time – providing an explicit description of both the states and the actions. This means that the modeler can determine freely whether, at a given moment of time, he wants to concentrate on states or actions. The states of a CPN are represented by means of places. Each place has associated *data type* determining the kind of data which the place may contain. For CPN, the terms: type, value, operation, expression, variable, binding and evaluation are used in exactly the same way as these concepts are used on high level programming languages.

A state of a CPN is called a *marking*. It consists of a number of tokens positioned on the individual places. Each token carries a data value which belongs to the type of the corresponding place. The token values are referred to as *token colours* and the data types are referred to as *colour sets*. This is the difference between CPN and PN, the "coloured" tokens of a CPN are considered to be distinguishable from each other. This is in contrast to ordinary PN which have "black" indistinguishable tokens.

The actions of a CPN are represented by means of transitions. An incoming arc indicates that the transition may remove tokens from the corresponding place while an outgoing arc indicates that the transition may add tokens. The exact number of tokens and their data values are determined by the *arc expressions*.

As the definition of tokens within a CPN place is modified from the ordinary PN, the enabling rule of a transition is, therefore, slightly adapted as follows:

1. A transition t is said to be enabled if all incoming arc expressions of t can be evaluated.
2. An enabled transition may or may not fire (depending on whether or not the event actually takes place).
3. A firing of an enabled transition t removes token of type and number specified by the arc expression of an arc connecting p to t for each input place p of t. Additionally, A firing of an enabled transition t adds token of type and number which are also specified by the arc expression of an arc connecting t to p for each output place p of t.

These CPN firing rules are preserved in our proposed extensions to the CPN theory which is presented in the next section.

The proposed adaptations to a coloured Petri-Net theory

In the previous section, we briefly described a PN and its derivative, CPN. Both of these modeling languages have been successfully exploited in many applications. For modeling real-time and real-time embedded systems, however, only a few literatures have been cited. As the objective of our research is to construct a software tool to automatically generate a computer source code from visual models, a classical PN is far from being useful in practice. A CPN, by contrast, provides a set of primitives and inscriptions that could ease the task of designers in system modeling. In addition, its inscription can possibly be used in both model execution and high level source code generation (Jensen, 1991). Unfortunately, a CPN's primitives and inscriptions in a standard form cannot be employed to successfully model real-time systems.

In this section, we propose an adaptation to a CPN theory, called *ertCPN* (*e*mbedded *r*eal *t*ime *c*oloured *P*etri-*n*et) theory for short (Kurdthongmee, 2001), for use in modeling real-time embedded systems. First of all, we give the rigorous definitions of all primitives which are elements of *ertCPN*. Then, we describe new primitives which are introduced specifically for real-time system modeling and automatic code generation.

The definitions of the *ertCPN* primitives

We have extended the definitions of CPN's primitives, which are places, transitions, arcs and inscription languages, to be more flexible for modeling embedded real-time systems. The following are the definitions:

Definifion 1: "A *place* is similar to the variable of a high-level programming language. The attributes of a place consist of *(place name, data type, (optional)initial value)*. The *place name*

will be automatically used to name a variable of type *data type* with the initial value equal to *initial value*. A place with only outgoing arcs (see *definition 2*) is called a *source place*. In addition, a place with only incoming arcs is called a *sink place*."

Definifion 2: "An *arc* is a vector connecting any two different types of primitives, a place to a transition or a transition to a place. It indicates the value of data flowing between the primitives under control of the arc's *arc expression*. The *arc expression* in *ertCPN* is a subset of the ANSI-C. It is normally located next to the arc in order to make the model more readable. There are four types of arc expression, namely *data flow*, *condition, assignment* and *control flow*."

Definition 3: "A *transition* is a piece/block of codes or action that operates on places connecting to it. A transition is ready to fire or execute if all of its incoming arcs can be evaluated; i.e. all variables within an arc's inscription are previously defined and initialised. The result of a transition is evaluated by its outgoing arcs and sent out to the place connecting to it. The type and value of data are also controlled by the arcs' arc expression. Only two attributes, both optional, of a transition are allowed; there are *(transition name, guard)*. A *transition name* will be automatically used to name a piece/block of codes. A *guard* is a condition prohibiting a transition from firing/executing even it has already received all required data flows."

Figure 2(a) demonstrates terms and their existence with respect to the primitives of *ertCPN* which are used for modeling a look-up table process in Figure 2(b).

Typically, a successful concurrent software design methodology views a system in two levels of abstraction, which are *architectural* and *behavioural* levels. The architectural level defines a system structure while the behavioural level defines what a system does. Both a standard PN and a CPN model a system, as a whole, as a set of places and transitions which can be concurrently executed. They have no definition of architecture which is very important for modeling real-time systems. We propose an additional primitive called a task primitive under the following definitions.

Definition 4: "A task primitive is used to group places, transitions and arcs which altogether describe the behaviour of a specific task/process of the system. A connection between places and transitions of any two tasks or between a task' s member and an isolated place (**definition 5**) can be made by use of an arc. This represents data flowing between a pair of primitives connecting to the arc. A task is equivalent to a function or

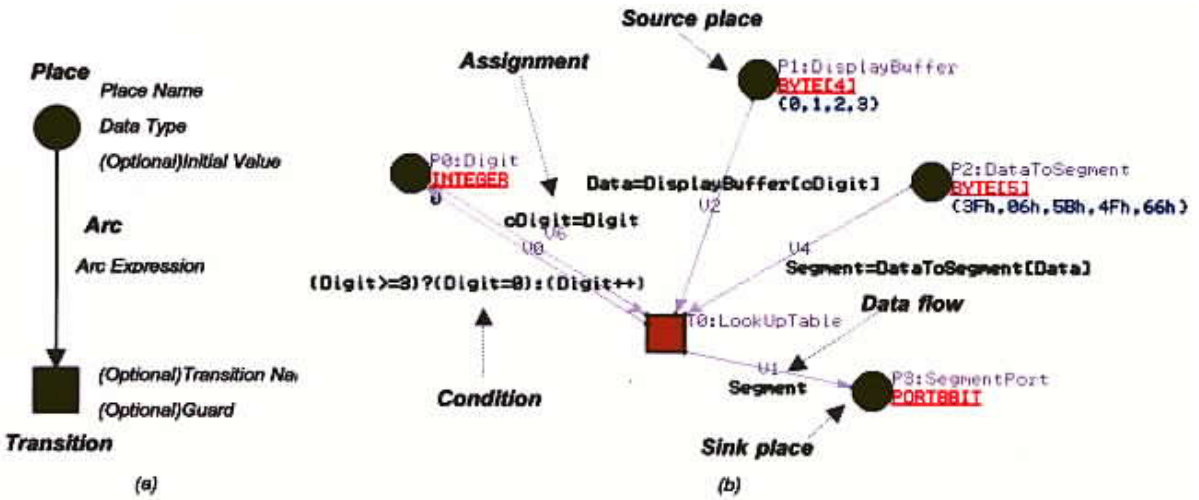

Figure 2. (a) Terms and their existence with respect to the primitives and (b) Example of applying the proposed primitives

routine in a high-level programming language and can be used to model the common function/routine of a system"

Definition 5: "An isolated place is a place which does not belong to any task within a whole model. Conceptually, an isolated place is similar to a global variable in high level language. This is used to represent shared resources of a system. It is noted that there is no definition of an isolated transition."

In real-time systems, there are mainly two kinds of task primitive, which are periodic and sporadic tasks. Periodic tasks are activated on a regular basis between fixed time intervals. They are typically used for systematically monitoring, polling, or sampling information from sensors over a long time interval. In contrast, sporadic task are event driven; they are activated by an external signal or a change of some relationships. A periodic task must have triple predefined attributes, which are *period* or *released time*, *worse-case execution time* and *deadline*. These parameters inform that the task is normally activated or released at the beginning of each period or releases time, must be finished before its deadline and the amount of work it is to do is equal to its worse-case execution time. In practice, the second attribute of a task is optional as it can be automatically calculated by the tool once the task's code has already been generated.

While a periodic task has three attributes, a sporadic task has only two attributes which are *deadline* and *worse-case execution time* with similar definitions to the periodic task given previously. The attributes provided by all tasks in the system are used for schedulability analysis.

Figures 4 and 5 demonstrate two tasks of type periodic and sporadic, respectively. The behaviour of the task in Figure 4 is to periodically sample temperature, determine if it is greater than a set point and signal the temperature indicator appropriately. The task is also responsible for storing the temperature into a globally defined array of integers. The task reads data from the input place of type *port8bit* every 20mS with optional worse-case execution and with relative deadline 50mS.

In contrast, the latter task depicted in Figure 5 is activated once there exists an external event occurring on the transition. Behaviourally, this task only samples a temperature data once the *pushSwitch* is pressed and stored it to a globally defined array of bytes.

The elements of *ertCPN* to support model execution and code generation

The application of an ordinary PN, or even its derivative CPN, to the software engineering community is only for model simulation. Jensen (1991) concluded that the inscriptions of CPN can only be used for guiding programmers in order to manually generate target code. As far as the automatic code generation is concerned, there is no literature on PN and CPN reporting the success in integrating code generation to the modeling tool.

For *ertCPN*, we have designed the arc's inscription and other additional primitives to support code generation which is targeted to be an embedded system. It is obvious that the attributes of a place under *definition 1* are enough to be used for variable's memory allocation or constant declaration. Additionally, a transition is mapped to a piece of codes called "*basic block*" which is operated on data from its incoming places to produce output to its outgoing places. The operation to be performed on the input variables (input places) with respect to the basic block (the transition) depends on the transition's arc expression.

According to definition 2 given in the previous subsection, there are only four types of arc expression, which are *data flow*, *condition, assignment* and *control flow*. The *data flow* expression is mainly used to specify the type of data, variable name, flowing between a pair of place and transition. From Figure 2(b), the *Segment*-variable of type data flow is a result of *LookupTable*-transition execution. During compilation process, the data flow type of inscription is used to produce data flow graph (DFG) and control flow graph (CFG) which describe the connections between basic blocks in the task. This type of inscription does not produce any target code.

The *condition* type of inscription is used for selectively executing its left or right statements as a result of the evaluation of its Boolean expression. During model execution, the condition's Boolean expression is executed in order to select the statement to be performed later. During compilation/code generation processes, all elements of the condition type of inscription are transformed into the target architecture's assembly codes. According to Figure 2(b), the *LookupTable*-transition is connected to the *Digit*-place via the arc with inscription: *(Digit>3)?(Digit=0):(Digit++)*. This inscription is used to control the *Digit*-variable to have its value between 0 and 3.

The *assignment* type of inscription is integrated into the *ertCPN* in order to open an opportunity for designers to assign a model's variable to a new variable which could be the local variable of a task or process. This could also be used to trace a variable's value during model execution. The assignment inscription in Figure 2(b) is *Data=DisplayBuffer[cDigit]* which assigns the *Data*-variable to an element index specified by *cDigit* of the *DisplayBuffer*-array.

The last inscription type is the *control flow* which is responsible for directing flow of program during model execution. During compilation process, the control flow type of inscription is used to produce control flow graph (CFG) which describes the connections between basic blocks of the task. This type of inscription is transformed into branch-statement in the target code which is in contrast to the data flow type of inscription that does not occur as any instruction in the target code.

It is obvious that only four types of inscription as described above, together with the clear declaration of places, are sufficient to make *ertCPN* models successfully describe the system's behaviour and architecture. Additionally, the data within *ertCPN* models provides appropriate information for model execution and automatic code generation. Above all, the structure of *ertCPN* models can be used to semi-automatically evaluate the most important parameter of a task in real-time system which is its worst case execution time (T_{WCET}) (Kurdthongmee, 2002). That is to say, if a task is in the form of a sequential arrangement of basic blocks (transitions), the T_{WCET} of the task is exactly the sum of execution time of all basic blocks. In contrast, if a task partially consists of a conditional arrangement of basic blocks, the T_{WCET} of the task could be the sum of execution time of some basic blocks together with other basic blocks which are the target of the conditional statement. The latter group of basic blocks must be decided by designers.

Sample *ertCPN* models and their - behaviour

At this point, it could be easier to understand the strength of *ertCPN* by re-considering the model in Figure 2(b) as a whole. First of all, we have to check which transition is ready to fire. As this sample model consists of only 1 transition, *LookUpTable*, it is considered at this point. With respect to the *LookUpTable*-transition, there are 3 places supplying token to the transition and according to the arc's expression the *Digit*-place and its arc expression *cDigit=Digit* must be evaluated first (followed by *DisplayBuffer* and, finally, *DataToSegment*-place). The first occurrence of *LookUpTable*-transition, *cDigit* is bound to *Digit* which is currently equal to 0. With a current value of <*cDigit=0*>, the ready-to-be-evaluated arc expression is then from the *DisplayBuffer*-place which is evaluated to be *Data=DisplayBuffer[0]*. As a result of the previous evaluation <*Data=0*>, the *DataToSegment*'s connecting arc can now be evaluated by assigning the *Segment* to *DataToSegment[0]* which is equal to 3Fh. Finally, the *LookUpTable*-transition sends out a <*Segment= 3Fh*> to *SegmentPort* and follows by evaluating the *(Digit>=3)?(Digit=0):(Digit++)*-inscription, which results in a modification of *Digit*-place's value.

If the model is passed through the compilation process, the following sequence of inscriptions must be generated:

```
cDigit = Digit;
Data = DisplayBuffer[cDigit];
Segment = DataToSegment[Data];
SegmentPort = Segment;
(Digit>=3)?(Digit=0):(Digit++)
```

The screenshot of the modeler's code generation pane for this model is shown in Figure 3. The format of a list of an intermediate code is as follows: {*From_primitive Arc_name To_primitive* {*Inscription_detail*} (optional)*Special_tag*} where *Arc_name* is the arc's name within the model. The *Arc_name* is originated from *From_primitive* and pointed to *To_primitive*. The *Inscription_detail* is the *Arc_name*'s inscription in postfix form. Finally, the *Special_tag* (*TERMINATE* or *RELATIVE*) is an optional field used to specify the flow of control within a block. All of the intermediate codes produced by our modeler which are ANSI-C subset can be used as an input for a standard ANSI-C compiler. However, we have found that most commercially available compilers modify the structure of source codes and make them difficult to evaluate the correct worst case execution time. In our prototype tool, a supplementary tool is, therefore, specially implemented in order to compile an intermediate code directly into the target architecture's specific assembly codes.

Let's re-consider the *ertCPN* model illustrated in Figure 4 which consists of two transitions within the task. Having checked the ready-to-fire condition of both transitions, it is obvious that *T0*-transition is ready to fire while *T1*-transition is still waiting for the value of *T*. Consider *T0*-transition, the binding sequence that causes *T0*-transition to fire is:

aIndex=Index;
Temp=TempSensor
ArrayOfData[aIndex]=Temp
(Index>=9)?(Index=0):(Index++)
P7=Temp

Notice that at the end of *T0*-transition's execution seqeunce, it assigns *P7*-place to the value of *Temp*. This opens an opportunity to *T1*-transition to be ready to fire with the following sequence of inscriptions:

```
1 place0 vector0 transition0 {ASSIGNMENT cDigit Digit}
2 place1 vector2 transition0 {ASSIGNMENT Data {@DisplayBuffer cDigit}}
3 place2 vector3 transition0 {ASSIGNMENT Segment {@DataToSegment Data}}
4 transition0 vector4 place0 {CONDITION {{DATAFLOW Digit} >= {DATAFLOW 3}}
5         {ASSIGNMENT Digit 0}
6         {ASSIGNMENT Digit Digit 1 +}}
7 transition0 vector1 place3 {DATAFLOW Segment} TERMINATE
8 END
```

Figure 3. A sequence of inscriptions after compiling the model in Figure 2(b)

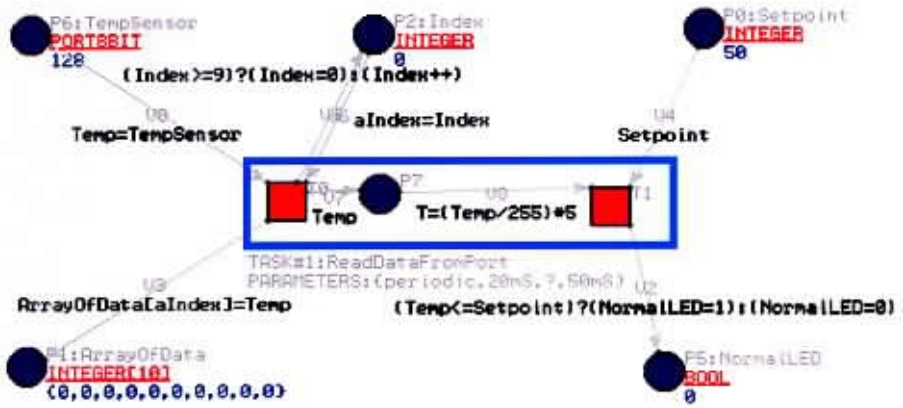

Figure 4. A sample periodic task modeled by use of *ertCPN*

Setpoint=50
(Temp<=Setpoint)?(NormalLED=1):
(NormalLED=0)

The last inscription is of type condition which is used to set/clear status of the *NormalLED*. The sporadic counterpart of this model illustrated in Figure 5 has the same behaviour as described above with only one more condition to be considered before the task can be activated. Such a condition is defined by *V9*-arc, *(PushSwitch==1)?*, which is of type control flow. The Boolean expression within the inscription must be evaluated to be true in order for the task to execute.

The sequences of inscriptions after compiling the models in Figures 4 and 5 by use of our modeler are shown in Figures 6(a) and (b), respectively.

Implementation and Discussion

In order to ensure that the proposed adaptations of CPN are viable in practice (both executable and able to generate code automatically) and easy to use, we have implemented a software tool called *ENVisAGe: Extended Coloured Petri-Net Based Visual Application Generator Tool* by use of a Tcl/Tk scripting language. Nearly all of the *ertCPN* models illustrated in this paper are produced by *ENVisAGe*.

The core module of *ENVisAGe* is written in iTcl/iTk which is an object oriented counterpart of the original Tcl/Tk. Currently, the tool's target architecture is a single processor MCS-51 family of microcontrollers. Although, using these powerful languages requires a steep learning curve since it lacks a supporting document, it eases the graphical user interface design and development. In addition, the resulting script can be executed on many platforms which have the open-source Tcl/Tk interpreter installed. Figure 6 is a screenshot of the model for a case study "a Temperature Monitoring and Control System" detailed in Section 6: Case study.

EnVisAGe provides many capabilities to assist users in order to create a model with minimum effort. It supports standard features of language's primitives: instantiation, manipulation and destruction of primitives and insertion/correction of primitives' inscriptions. In addition, it supports knowledge-based modeling by allowing designers to add the previous constructed modules of *ertCPN* into a current design and store a current design to be used as a module for later use.

Also, a current version of *EnVisAGe* supports task's execution analysis to easily verify the task's behaviour. In addition, an automatic code generation for MCS-51 architecture has currently been integrated into the tool. This also makes it capable to semi-automatically calculate T_{WCET} of tasks (Kurdthongmee, 2002). A schedulability analysis will be incorporated in the coming version of the tool.

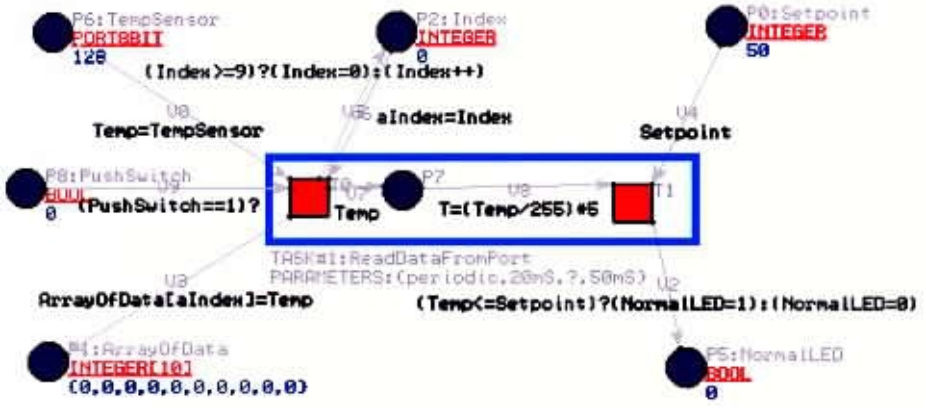

Figure 5. A sample sporadic task modeled by use of *ertCPN*

```
1  {place6 vector0 transition0 {ASSIGNMENT Temp TempSensor}}
2  {place2 vector5 transition0 {ASSIGNMENT aIndex Index}}
3  {transition0 vector6 place2 {CONDITION {{DATAFLOW Index} >= {DATAFLOW 9}}
4          {ASSIGNMENT Index 0} {ASSIGNMENT Index Index 1 +}}}
5  {transition0 vector3 place1 {ASSIGNMENT {@ArrayOfData aIndex} Temp} TERMINATE}
6  {transition0 vector7 place7 {DATAFLOW Temp {RELATIVE 1}}
7  {place7 vector8 transition1 {ASSIGNMENT T Temp 255 / 5 *}}
8  {place0 vector4 transition1 {DATAFLOW Setpoint}}
9  {transition1 vector2 place5 {CONDITION {{DATAFLOW Temp} <= {DATAFLOW Setpoint}}
10         {ASSIGNMENT NormalLED 1} {ASSIGNMENT NormalLED 0}} TERMINATE}
11 END
```

(a)

```
1  {place8 vector9 transition0 {CONTROLFLOW {{DATAFLOW PushSwitch} == {DATAFLOW 1}}}}
2  {place6 vector0 transition0 {ASSIGNMENT Temp TempSensor}}
3  {place2 vector5 transition0 {ASSIGNMENT aIndex Index}}
4  {transition0 vector6 place2 {CONDITION {{DATAFLOW Index} >= {DATAFLOW 9}}
5          {ASSIGNMENT Index 0} {ASSIGNMENT Index Index 1 +}}}
6  {transition0 vector3 place1 {ASSIGNMENT {@ArrayOfData aIndex} Temp} TERMINATE}
7  {transition0 vector7 place7 {DATAFLOW Temp {RELATIVE 1}}
8  {place7 vector8 transition1 {ASSIGNMENT T Temp 255 / 5 *}}
9  {place0 vector4 transition1 {DATAFLOW Setpoint}}
10 {transition1 vector2 place5 {CONDITION {{DATAFLOW Temp} <= {DATAFLOW Setpoint}}
11         {ASSIGNMENT NormalLED 1} {ASSIGNMENT NormalLED 0}} TERMINATE}
12 END
```

(b)

Figure 6. Sequences of inscriptions (a) and (b) are from the models in Figure 3 and 4, respectively.

The drawback of *EnVisAGe* is similar to the drawback of CPN, itself. That is to say for a complex system the resulting model has a huge amount of primitives. This makes the model more difficult to read and interpret. This could partially be solved by utilizing a concept of hierarchical CPN to selectively hide/show detail of some tasks of models. This capability has not been integrated into our modeler yet.

Conclusions and Future Work

The paper has described the modeling framework for an embedded real-time system that has been adapted from a coloured Petri-net theory. Integrating more primitive, a task-primitive, into a set of standard CPN primitives; places, transitions and arcs, and redefining inscriptions makes the resulting modeling framework more suitable for real-time embedded systems. A resulting model obviously shows both architecture and behaviour of the system within the same model. The information provided by the primitive's attribute of a model can be used for schedulability analysis which is the heart of real-time system design. In addition, an inscription language can also be used for system analysis by means of a standard CPN; i.e. state-space analysis.

A current version of the *EnVisAGe* modeler supports model execution analysis on a task by task basis. We plan to integrate a state-space analysis tool into the next version of our modeler. Also, a schedulability analysis, an extended CPN simulation, a worst case execution analysis and automatic code generation tool will be included in the coming version of the tool.

Case Study: *ertCPN* model of a temperature monitoring and control system

In this section, we present a simple example of an embedded real-time system modelled by means of the *ertCPN* theory. Consider a temperature controlled system with capabilities to display a current temperature, allow users to adjust its set point and manually shut down the system. There are three input devices to the system which are a temperature sensor, a push button switch

and a 4×4 matrix keyboard. In addition, there are two output devices of the system which are a 4-digit multiplexing LED and a relay connecting to an external heating element. According to the specification, the system performs by periodically sampling a temperature from the sensor, displaying the current temperature on the LED display unit, comparing the temperature to its set point and setting/clearing the relay appropriately. Under normal condition, the relay is always turned on. At any instant of time, however, if the temperature is greater than its set point, the relay will be turned off. Another responsibility of the system is to periodically scan the 4×4 matrix keyboard. If the system encounters that a key is pressed, it will read the status of both row and column ports which control the operation of the matrix keyboard, then look for the numeric value of the pressed key and finally store the numeric value as the temperature's set point in an appropriate variable for later retrieval. Finally, if users press the "shut down" key, the system must respond immediately by switch-off the relay. It is noted that in order to ease the design and reduce the complexity of the resulting model, an on-off control system is used.

From the requirement, it is obvious that the system has both type of tasks: *periodic* and *sporadic*. Table 1 classifies the type of tasks, the assigned name which is going to be used in the model and the timing attributes in the form of (release time, worst case execution time and deadline). It is noted again that the worst case execution time attribute is optional and will be automatically calculated by the modeler. With the information about the whole system on a task-by-task basis, the *ertCPN* model of the system can be constructed. Figure 7 illustrates the resulting model created by *EnVisAGe* modeler which has been developed as part of our research.

According to Figure 7, *ertCPN* clearly shows the architecture of the system which consists of 4 tasks. It also illustrates the relationships amongst the tasks; i.e.: each time the *(periodic)DisplayScan*-task is activated, it retrieves a byte of data at location *cDigit* from a global variable *data* which is an array of 4 bytes. It also updates the global variable *digit* to always point to the next digit of the LED to be enabled next time the task is activated. The *digit* is of integer type which has a value between 0 and 3. This is clearly specified by the inscription: *(digit>=3)?(digit=0):(digit++)* at the arc connecting between the *digit*-place and the *T0*-transition.

According to *ertCPN* theory given in Section 3, the behaviour of any task is described by a group of places and transitions within the task. Let's consider **Task#0:DisplayScan** first, once the task is activated it retrieves the global variable *digit* and assigns to *cDigit* which is later used to refer to the member of *enable* and *data*. The current value of *enable*, or *enable[cDigit]*, is sent out to the *enablePort*. Additionally, the current value of *data*, which is in decimal form, is converted to its corresponding seven segment code and sent out to the *segmentPort*. It is noted that *cDigit* is globally declared to be of type byte and is used by the **Task3:KeyboardScan**.

Behaviourally, **Task#1:ReadTempAnd Action** is responsible for sampling a current temp-erature value from the *tempSensor*. After changing from 8-bit data to a correct temperature value, the task stores the temperature into the

Table 1. The type of task, the assigned name and the timing attributes of the temperature monitoring and control system. (imm: immediately)

Type	Assigned Name	Time Attributes	Action
Sporadic	UserShutDown	(imm, ?)	Users shut down the system immediately.
Periodic	DisplayScan	(20mS, ?, 20mS)	Scan the multiplexed LED display digit by digit.
Periodic	KeyboardScan	(50mS, ?, 30mS)	Scan the matrix keyboard.
Periodic	ReadTempAndAction	(20mS, ?, 20mS)	Read temperature and control the relay.

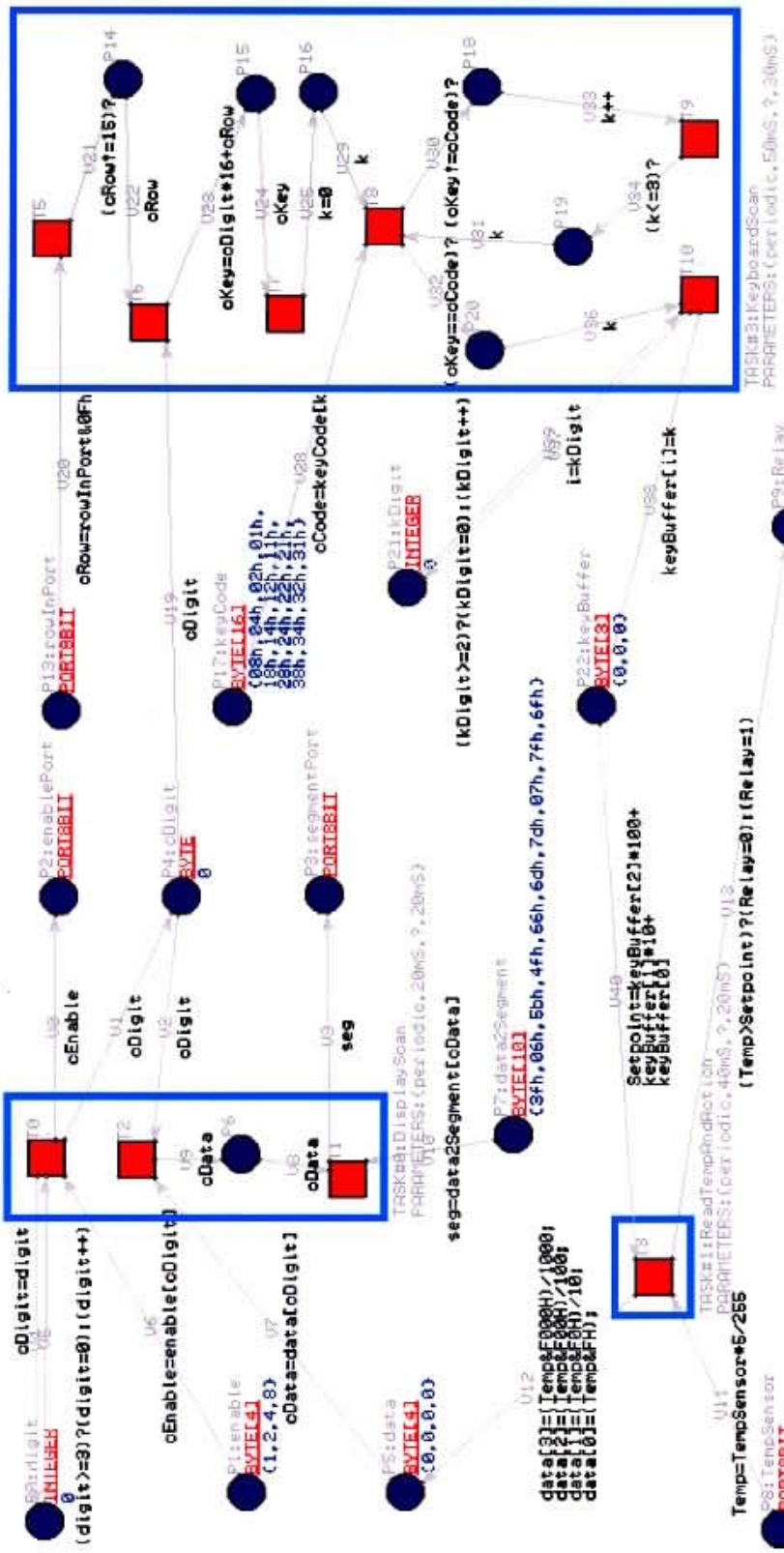

Figure 7. The *ertCPN* model of a temperature monitoring and control system

data-variable. The inscription to convert from an integer to an array of 4 bytes, *V12-arc*, consists of 4 statements which are of type assignment. This task is also responsible for setting or clearing the state of the *Relay* as a result of comparing the variable *Temp* with the *Setpoint* value. The *Setpoint* value used by this task is a result of converting from *(global)keyBuffer* which is an array of 3 bytes into an integer (see *V10-arc*'s inscription).

Task#2:UserShutDown is the easiest to understand task. This task is of type sporadic and consists of only 1 transition. Once the status of the *shutDownButton* (which is 0 by default) is changed to one, the transition clears the status of the *Relay* unconditionally and immediately.

The last task with the most complex structure is **Task3#KeyboardScan**. The main duty of this task is to periodically sample the value of the *rowInPort,* keeps only the last four bits and assigns the result to *cRow*. Logically, pressing any key causes the c*Row* to be any value other than 15 (0Fh). The task tests this condition and returns immediately if no key is pressed. If a key is pressed however, the task converts a current *cKey= cDigit*16+cRow* into a correct key value by use of loop-comparison, formed by *T8-V30-P10-V33-T9-V34-P19-V31*. The key value is stored in *keyBuffer* at the location specified by *kDigit*.

Acknowledgments

The work was carried out as part of the Visual Application Generator for a Small Real Time System project sponsored by the National Electronics and Computer Technology Center (NECTEC) and the Thailand Toray Science Foundation (TTSF).

References

Ghezzi G., Mandrioli D., Mosarea S. and Pezze M. 1991. A Unified High-Level Petri-Net Formalism for Time-Critical Systems, IEEE Trans. On Software Engineering, 17(2) : 160-172.

Ghezzi G., Jazayeri M. and Mandrioli D. 1991. Fundamentals of Software Engineering, Prentice-Hall, Englewood Cliffs, NJ.

Goquen J., Meseguer J., Futatsuki K., Lincoln P. and Jouannaud J. 1988. *Introducing OBJ*, TR-SRI-CSL-88-8, Computer Science Lab, SRI International, Menlo Park, CA.

Harel D., Lachover H., Naamad A., Pnueli A., Poloti M., Sherman R., Shtull-Trauring A. and Trakhtenbrot M. 1990. STATEMATE: A Working Environment for the Development of Complex Reactive Systems, IEEE. Trans. On Software Engineering, 16(4) : 403-414.

Jahanian F. and Mok A. 1994. Modechart: A Specification Language for Real-Time Systems, IEEE Trans. On Software Engineering, 20(12) : 933-947.

Jensen K. 1992. Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use, EATCS Monographs on Theoretical Computer Science, Springer-Verlag.

Jensen, K. and Rozenberg, G. (eds.). 1991. High-Level Petri Nets: Theory and Application, Springer-Verlag.

Kurdthongmee, W. 2002. An Embedded Real-Time System Modeling Tool Based on Extensions of Coloured Petri-Net, Proceedings of NCSEC 2002 conference, Jomtien, Pattaya, Thailand.

Kurdthongmee W. 2002. *ENVisAGe*: An Extended Coloured Petri-Net Based Visual Application Generator Tool for Real-Time Embedded Systems, Proceedings of NSTDA conference, Thailand.

Kurdthongmee W. 2002. Integrating Worst Case Execution Time Analysis to an Open-Source Embedded System C-Compiler, Proceedings of RTAS2002 conference, San Jose, CA.

Murata T. 1989. Petri Nets: Properties, Analysis and Applications, Proceedings of the IEEE, 77(4).

Shaw C. 2001. Real-Time Systems and Software, John Wiley and Sons, New York.

Spivey J. 1989. The Z Notation: A Reference Manual, Prentice-Hall, E Englewood Cliffs, NJ.