Dual Flow Nets: Modeling the Control/Data-Flow Relation in Embedded Systems

MAURICIO VAREA and BASHIR M. AL-HASHIMI
University of Southampton
and
LUIS A. CORTÉS, PETRU ELES, and ZEBO PENG
Linköping University

This paper addresses the interrelation between control and data flow in embedded system models through a new design representation, called Dual Flow Net (DFN). A modeling formalism with a very close-fitting control and data flow is achieved by this representation, as a consequence of enhancing its underlying Petri net structure. The work presented in this paper does not only tackle the modeling side in embedded systems design, but also the validation of embedded system models through formal methods. Various introductory examples illustrate the applicability of the DFN principles, whereas the capability of the model to with complex designs is demonstrated through the design and verification of a real-life Ethernet coprocessor.

Categories and Subject Descriptors: I.6.4 [**Simulation and Modeling**]: Model Validation and Analysis; D.2.2 [**Software Engineering**]: Design Tools and Techniques—*Petri nets*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*Control structures*; *Data types and structures*; B.5.2 [**Register-Transfer-Level Implementation**]: Design Aids—*Verification*

General Terms: Design, Theory, Verification

Additional Key Words and Phrases: Dual flow nets, Petri nets, modeling, formal verification, embedded systems, LTL, CTL, symbolic model checking, tripartite graph

1. INTRODUCTION

The design of embedded systems has significantly improved in recent years. Design techniques, such as hardware/software codesign [Staunstrup and Wolf

Partially supported by the EPSRC (UK), under grant GR/S95770.
Authors' addresses: M. Varea and B. Al-Hashimi are with the School of Electronics and Computer Science, University of Southampton, SO17 1BJ Southampton, UK; email: {mv, bmah}@ecs.soton.ac.uk; L. A. Cortés, P. Eles, and Z. Peng are with the Department of Computer and Information Science, Linköping University, S-581 83 Linköping, Sweden; email: {luico,petel, zebpe}@ida.liu.se.

1997; Wolf 2003] and formal verification [Kern and Greenstreet 1999; Wang 2004], have increased the efficiency and reliability of the design process to an extent that allows the Electronic Design Automation (EDA) industry to widely rely on them. By the use of models, system designers have been able to exploit features related to specific applications and understand the underlying principles associated to a particular behavior, which is a key to obtaining cost-effective solutions.

Although abstraction is a desired feature of models, a designer could reach incorrect analytical conclusions of an embedded system design because of the lack of explicit information. There is a trade-off between the amount of information that should be available to the designer, in order to perform a correct design, and how much detail should be hidden away to avoid confusion and unnecessary complexity. The relationship among different levels of abstraction can be conceptualized by, e.g., Gajski's Y-Chart [Gajski 1987]. The Y-Chart visualization of the design process assumes three orthogonal domains (i.e., behavioral, structural, and physical) which are meant to emphasize distinct properties of the same design, while a set of concentric circles represent the abstraction level. Purely behavioral models are suitable for analyzing the intended functionality of an embedded system design, but they cannot cope with implementation issues. Structural models, on the other hand, are more appropriate for connectivity issues and netlist-type representations, which make them suitable for implementing a design, but have a detrimental effect on its analysis.

Embedded systems usually consist of some transformative parts dedicated to calculate and transfer values through the system (data flow) and some reactive part that curbs the type and order of such transformative operations (control flow). The result of a data flow operation can reside within the data flow or be part of the control flow—the latter one is called *condition*. Similarly, a subset of the control flow interacts with the data flow, and constitutes the set of *control signals*. These control/data–flow interactions are not a trivial matter and have been long neglected in behavioral models for embedded systems—for the sake of simplicity.

Regardless of its abstraction level, we believe that an embedded system model should differentiate between control and data components, not only in the structural model, but in the behavioral one as well. Mechanisms for modeling these interactions and still being able to reach an analytical conclusion from them, should be provided in all cases. This paper defines a model that achieves a satisfactory analysis of an embedded system specification, based on these interactions, and reaches closeness to a structural view of the system. By making these interactions explicit, our model has been conveniently fitted into a formal verification framework, allowing the validation of embedded systems of numerous natures.

The presented work is organized as follows. In Subsections 1.1 and 1.2, relevant work is reviewed and the underlying motivation for this work is outlined. Section 2 defines our model from a structural and behavioral perspective. The verification of such a model is carried out in Section 3, while Section 4 shows the applicability of this model to some real-life problems. Finally, Section 5 outlines some conclusions.

1.1 Related Work

The Finite-State Machine (FSM) model [Gill 1962] has been widely used in control theory and is one of the fundamental pillars in the development of control-dominated representations for embedded systems. However, increased demand for Digital Signal Processing (DSP) and Multimedia functionality have drastically increased the dataflow complexity of current embedded systems, thus the need for data-dominated representations as well.

Formally, the FSM is a tuple $\langle S, I, O, f, h\rangle$, where S, I and O are the finite sets of *states*, *inputs*, and *outputs*, respectively. $f : S \times I \rightarrow S$ is the *next-state* function and h is the output function. The function f takes the current state of the system, i.e., the state at time k_i, together with a subset of inputs, and produces a new state for the system at time k_{i+1}. The output function depends whether the FSM is a Mealy machine ($h : S \times I \rightarrow O$) or a Moore machine ($h : S \rightarrow O$). The Synchronous DataFlow (SDF) model [Lee and Messerschmitt 1987], on the other hand, is a special case of Dataflow Graph, where a given specification is captured by a directed acyclic graph and a scheduling. An $SDF = \langle V, E, d, t, p, c\rangle$ is composed of: a set of nodes or actors (V) that transform input data streams into output streams, a set of channels $E \subseteq V \times V$ on which the data streams are carried, and four functions (d, t, p, and c) that account for the behavioral part of the specification.

Intuitively, one way to extend this FSM definition, in order to support data type information, is by extending each element of S, I, and O from the Boolean to the integer domain. This issue has been discussed in [Gajski 1997], and given the name of Finite-State Machine with Datapath (FSMD). The FSMD model is a tuple $\langle S, V, I, O, f, h\rangle$ where S is the set of states, V is the set of *datapath* variables, and I and O are defined as $I : I_C \times I_D$ and $O : O_C \times O_D$ respectively. State transitions may include arithmetic and logic operations on the set of datapath variables, which is a desired feature in models that deal with the control/data-flow relationship. However, this model lacks an explicit notation for concurrency.

The Petri Net (PN) model [Murata 1989] uses a *bipartite* graph, i.e., a graph with two disjoint set of vertices, to differentiate between passive (*places* $p \in P$) and active (*transitions* $t \in T$) elements of a net (where $P \cap T = \emptyset$). Petri nets have played an important role in the modeling of embedded systems, since they are inherently concurrent, but their insufficient data-flow analysis has motivated the research in High-Level Petri Nets (HLPN), such as Predicate/Transition Nets (PrT-Nets) [Genrich 1987], Coloured Petri Nets (CPN) [Jensen 1992, 1994, 1997], and Object Oriented Petri Nets (OOPN) [Esser 1996]. Works, such as Kleinjohann et al. [1997], and Grode et al. [1998] are a clear indication that PN-based modeling of embedded system is progressing. However, these approaches are somewhat generic (not specifically designed for embedded systems), thus leading to the introduction of another generation of modeling approaches [Peng and Kuchcinski 1994; Strehl et al. 2001; Cortés et al. 2000] that particularly address a variety of issues in the design of embedded systems.

The concept of "separate but related" control/data parts has been considered in the Extended-Timed Petri Net (ETPN) model [Peng and Kuchcinski 1994],

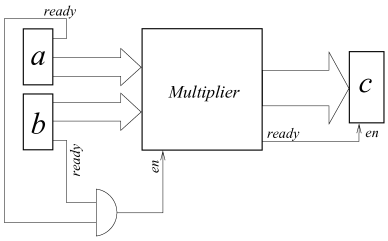
where the control part is captured using a Timed Petri Net (TPN) and the data path is represented as a directed graph. The FunState model [Strehl et al. 2001], on the other hand, models the control flow as an FSM, while the data flow is captured by means of a network of functions and storage units that resembles a Queue Petri Net [Finkel and Memmi 1985]. Whereas the authors of the above works favor the combination of two disjoint semantics, others [Eles et al. 1998; Ziegenbein et al. 1999; Cortés et al. 2000] lean toward a consistent semantics for modeling the complete system. Conditional Process Graphs (CPG) [Eles et al. 1998] capture both control and data flow of an embedded system specified as a net of interacting processes, by means of a data-flow graph that has conditional edges. In the System Property Intervals (SPI) [Ziegenbein et al. 1999] the execution of a process depends on data availability, which captures the conditional behavior of an embedded system due to non-constant data rates. The Petri Net Representation of Embedded Systems (PRES+) model [Cortés et al. 2000] uses data tokens to communicate results among processes, including timing property analysis through an explicit notion of time in the semantics of its underlying PN.

Contrary to Peng and Kuchcinski [1994], Strehl et al. [2001], Ziegenbein et al. [1999], and Cortés et al. [2000], where the underlying *bipartite* graph of a Perti net has been syntactically and semantically extended to support control/data-flow interactions, our aim is to propose a completely new model, namely *Dual Flow Net* (DFN), which is based in a *tripartite* graph instead. By using a tripartite graph $\mathcal{K}_{n,m,h} = \{I \cup J \cup K; (I \times J) \cup (I \times K) \cup (J \times K) | I \cap J \cap K = \emptyset\}$, control states and data activities are modeled by disjoint sets of vertices, which are still linked by the third (also disjoint) set of vertices. This way of dealing with the control/data flow is closer to the behavior of the final implementation than other PN extensions.

1.2 Motivational Example

It is known that systems combining control and data processing functions have quite a significant context-switching overhead [Österling et al. 1997]. This section considers the modeling of a simple integer multiplier based on iterative additions which, besides being orders of magnitude less complicated than real-life applications, still outlines the problem of switching between contexts during the analysis phase. Figure 1 shows both, a diagram and the pseudocode for this multiplier. The start of the multiplier's execution is given by signal en, which holds if both registers (a and b) contain a valid data. After a number of iterations, the multiplier outputs the result of the multiplication, which is stored in register c because of the acknowledgement produced by the ready signal.

Figure 1a shows the schematic implementation of the multiplier, which denotes that structurally there is a clear separation between control flow (ready and en signals) and data flow (a, b, and c operands). This separation is easily visualized by contrasting thick and thin arrows in the schematic diagram. However, it becomes less obvious in the behavioral model presented as pseudocode in Figure 1b.



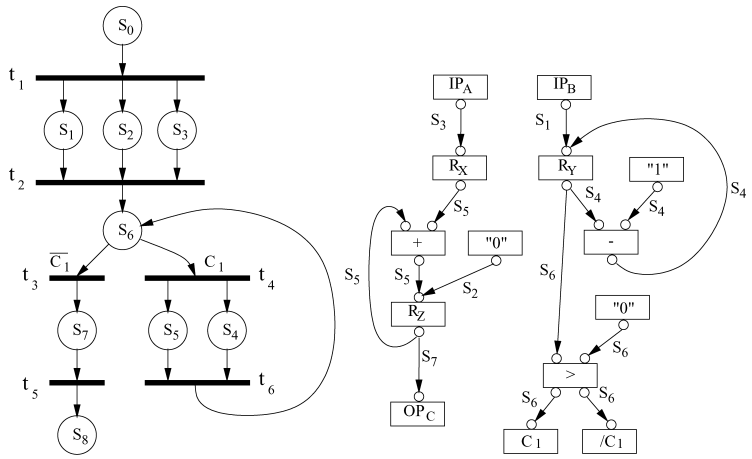
(a) Diagram

```
int mult(int a,int b) {
    int x, y, z;
    x ← a; y ← b; z ← 0;
    while(y > 0) {
        z ← z + x;
        y ← y - 1;
    }
    return z;
}

c=mult(a,b);
```

(b) Pseudocode

Fig. 1. Multiplier.



(a) Control part

(b) Data path

Fig. 2. ETPN representation of the multiplier.

Figure 2 illustrates the use of ETPN [Peng and Kuchcinski 1994] for modeling the multiplier example, where the control part and data path are captured in Figures 2a and b, respectively. The first part is captured as a TPN [Merlin and Faber 1976], while the second part is represented as a directed graph where nodes are used to capture data manipulation and storage. A change in the control part, i.e., changing the marking of the underlying PN, affects the data path in the sense that data is transferred through all edges of the data path that are labeled with the state $s \in S$ that contains a token in the control part. For example, the firing of transition t_1 allocates tokens in places s_1, s_2, and s_3, which leads to data transferring from IP_A to R_X, from IP_B to R_Y, and from "0" to R_Z, in the data path. Then, iterative firings of t_4 and t_6 lead to the desired multiplication result, because places s_4 and s_5 are associated with the subtract and the addition operations in the data part of the model.

Further analysis of this model rises the issue about improvement of the control/ data-flow integration. Despite being ETPN, a representation easy to map to the input of a synthesis tool, this model lacks simplicity during the analysis phase. This trade-off is typical in embedded systems design: being closer to the structure of the design yields a better synthesis process, while proximity to the behavioral model is beneficial for the analysis. Models, such as ETPN or FunState, are closer to the structural domain and, therefore, deriving an implementation from them is feasible, while models, such as PRES+ [Cortés et al. 2000] or SPI [Ziegenbein et al. 1999] are closer to the behavioral domain and favor the analysis. As synthesizing a model could involve various optimization steps, we are interested in reducing the amount of context switching only for analysis. The verification complexity is directly related to the state space of the model, which increases its size depending on this factor. Therefore, context switching in the sense applied to this paper, has an impact on the performance of the formal verification tool used. This can be illustrated by comparing the following trace in an ETPN execution, with its DFN counterpart (c.f., Section 2.4).

1.2.1 *Trace for* $5 \times 3 = 15$. We use the following notation: "$;$" for sequence of events that must change environment, "$|$" to separate events within the same environment, {SUM, SUB, CMP} to indicate an operation, and "$\leftarrow$" for assignments. Assuming that $IP_A \leftarrow 5$, $IP_B \leftarrow 3$, and there is a token in s_0, the trace that leads to the result is given by:

$$t_1;\ R_Y \leftarrow 3|\ R_Z \leftarrow 0|\ R_X \leftarrow 5;\ t_2;\ \circledcirc \leftarrow 3|\ \circledcirc \leftarrow 0|\ \texttt{CMP}|\ C_1 \leftarrow 1|\ /C_1 \leftarrow 0;\ t_4;\ \oplus \leftarrow 0|\ \oplus \leftarrow 5|\ \texttt{SUM}|\ R_Z \leftarrow 5|\ \ominus \leftarrow 3|\ \ominus \leftarrow 1|\ \texttt{SUB}|\ R_Y \leftarrow 2;\ t_6;\ \circledcirc \leftarrow 2|\ \circledcirc \leftarrow 0|\ \texttt{CMP}|\ C_1 \leftarrow 1|\ /C_1 \leftarrow 0;\ t_4;\ \oplus \leftarrow 5|\ \oplus \leftarrow 5|\ \texttt{SUM}|\ R_Z \leftarrow 10|\ \ominus \leftarrow 2|\ \ominus \leftarrow 1|\ \texttt{SUB}|\ R_Y \leftarrow 1;\ t_6;\ \circledcirc \leftarrow 1|\ \circledcirc \leftarrow 0|\ \texttt{CMP}|\ C_1 \leftarrow 1|\ /C_1 \leftarrow 0;\ t_4;\ \oplus \leftarrow 10|\ \oplus \leftarrow 5|\ \texttt{SUM}|\ R_Z \leftarrow 15|\ \ominus \leftarrow 1|\ \ominus \leftarrow 1|\ \texttt{SUB}|\ R_Y \leftarrow 0;\ t_6;\ \circledcirc \leftarrow 0|\ \circledcirc \leftarrow 0|\ \texttt{CMP}|\ C_1 \leftarrow 0|\ /C_1 \leftarrow 1;\ t_3;\ OP_C \leftarrow 15;\ t_5$$

This trace is used for comparison purposes with its counterpart for the DFN model, as discussed in Section 2.4.

2. FORMALIZATION OF THE DUAL FLOW NET MODEL

Unlike other Petri net-based approaches to embedded systems modeling, where the Petri net semantics is either circumscribed to the control domain [Brage 1993; Cortés et al. 2000; Peng 1987] or constitutes the basic structure of the data domain [Ziegenbein et al. 1999; Strehl et al. 2001], our model proposes a concise notation for both domains, leading to a uniform representation of such heterogeneous systems.

We argue in this paper that systems, which have two information flows, i.e., control and data flow, are best represented by models which are isomorphic to a *tripartite* graph. As discussed in the previous section, the suitability of tripartite graphs for systems with two flows of information comes from the fact that these systems have three aspects to be considered in the modeling process:

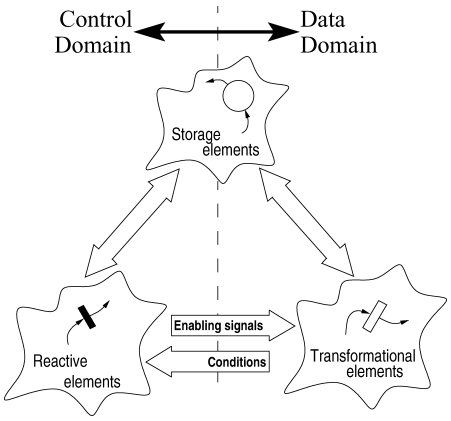


Fig. 3. A tripartite graph for control and data flow.

(1) the general state of the system, (2) the control flow, and (3) the data flow. Figure 3 depicts the interaction between these three sets, and shows that those sets are constituted by individual elements, which are: (1) *storage*, (2) *reactive*, and (3) *transformational* elements.

The DFN model and its elements are introduced in Definition 2.1. This definition is based upon the principles introduced in the following sections, which include the *dual flow structure* and the *initial marking*.

Definition 2.1. A *Dual Flow Net* is a pair $\mathcal{N} = \langle \mathcal{S}, \mu_0 \rangle$, where $\mathcal{S}$ is a Dual Flow Structure and μ_0 is the initial marking. □

In order to formalize the upcoming concepts, the notion of Atomic Propositions (AP) needs to be introduced. These are the most elementary type of evaluation that can exist in a condition, e.g., "$x > 0$." These propositions are composed of two parts, namely *subject* and *predicate*. The subject of an AP is a variable (e.g., "x") whereas the predicate is a property that the subject may or may not satisfy (e.g., "> 0"). This property consists of a symbol taken from the finite set of *conditional comparators*, $\sharp = \{=, \neq, >, <, \geq, \leq\}$, and a constant (e.g., "0"). In order to standardize this concept, we consider an AP to be a function, defined as follows:

Definition 2.2. An atomic proposition is a function of three arguments,

$$\mathsf{AP} : \mathbb{Z} \times \sharp \times \mathbb{Z} \mapsto \mathbb{B}$$

where the first argument is a variable $x \in \mathbb{Z}$, i.e., the subject, and the second and third argument are a symbol $s \in \sharp$ and a constant $K \in \mathbb{Z}$, which constitute the predicate of the atomic proposition.

For instance, the syntax $\mathsf{AP}(x, >, 0)$ expresses "$x > 0$."

The rest of this section introduces the principles of the DFN model, using a *structural model* for the organization and dependence analysis of each set of vertices, while the functional analysis is carried out by means of the *state space* and the *behavioral models*. Finally, we also illustrate the semantics of

the model employing the simple multiplier example introduced in Section 1.2 (a larger, more realistic example is modeled using DFN later in Section 4).

2.1 DFN Structural Model

The structure of the DFN model is based on a notation that comprises three types of vertices (c.f., Figure 3):

1. A set $P = \{p_1, p_2, \cdots, p_n\}$ of vertices that captures the state of the system;
2. A set $T = \{t_1, t_2, \cdots, t_m\}$ of vertices used to capture the changes in such states, i.e., the control flow, and exposes its influence over the third set Q of vertices;
3. A set $Q = \{q_1, q_2, \cdots, q_h\}$ of vertices that captures all transformations that are relevant to the data flow of the system, such as transferring information or performing arithmetical operations among registers.

The cardinality of each set P, T, and Q is given by n, m, and h, respectively; where $n > 0$, $m \geq 0$ and $h \geq 0$ (also, $m + h \neq 0$). The DFN structure is given by a weighted, directed, tripartite graph whose vertices $\mathcal{V} = P \cup T \cup Q$, where $P \cap T = \emptyset$, $P \cap Q = \emptyset$, $T \cap Q = \emptyset$, $P \neq \emptyset$, and $T \cup Q \neq \emptyset$, are used to represent *storage*, *reactive*, and *transformational* elements, respectively. Storage elements ($p \in P$) relate to memory components in the system (e.g., registers, memory cells, latches, and variables), reactive elements ($t \in T$) are associated with components in the control part, and transformational elements ($q \in Q$) refer to arithmetic operations performed among storage elements (i.e., components in a data path). Hereafter, the elements $p \in P$, $t \in T$, and $q \in Q$ are referred as *places*, *transitions*, and *hulls* of the DFN net, and are graphically represented by a circle, a bar, and a box, respectively

Definition 2.3. A Dual Flow Structure is a seven-tuple $\mathcal{S} = \langle P, T, Q, F, W, G, H \rangle$,

where: $F \subseteq (P \times T) \cup (T \times P) \cup (P \times Q) \cup (Q \times P) \cup (T \times Q) \cup (Q \times T)$
is a binary relation, called the *flow relation*;
$W : F \mapsto \mathbb{Z}^+ \cup \mathbb{Z}^-$ is a *weight function*;
$G : T \mapsto \sharp \cup \{\top\}$ is a *guard function*,
where $\sharp$ is the set of conditional comparators, and
$\top$ is an element from the binary set $\mathbb{B} = \{\bot, \top\}$ that means *true*;
$H : Q \mapsto \mathbb{Z}$ is an *offset function*.

For the sake of simplicity, this paper uses the notation $W(x, y)$ to denote $W((x, y))$. From the pictorial point of view, each transition $t \in T$ is labeled with a symbol from $\sharp$ according to the guard function $G(t)$. Hulls $q \in Q$ are also labeled with integers, corresponding to the offset function $H(q)$. In order to reduce notational clutter, transitions and hulls are only labeled in nontrivial cases, i.e., symbols $\top$ and numbers 0 are not explicitly written down across the net. These points have been exemplified in Figure 5 (See later).

The three disjoint sets of vertices in a DFN model, and their interactions, define the structure of the control and data domain shown in Figure 3. This

means that there is a direct relation between the control (data) flow of the original embedded system specification and the control (data) domain of the $\mathcal{S}$ net. Both control and data domains are formalized in Definitions 2.4 and 2.5, where the concepts of pre- and postsets, from the classical Petri net notation, are extended in order to support any element $p \in P$, $t \in T$, and $q \in Q$ of the new DFN model.

Definition 2.4. Given a certain $p \in P$, $t \in T$, and $q \in Q$, the following subsets are defined:

i. The control preset of a place, ${}^\bullet p = \{t \in T | (t, p) \in F\}$;
ii. The control postset of a place, $p^\bullet = \{t \in T | (p, t) \in F\}$;
iii. The control preset of a transition, ${}^\bullet t = \{p \in P | (p, t) \in F\}$;
iv. The control postset of a transition, $t^\bullet = \{p \in P | (t, p) \in F\}$;
v. The control preset of a hull, ${}^\bullet q = \{t \in T | (t, q) \in F\}$;
vi. The control postset of a hull, $q^\bullet = \{t \in T | (q, t) \in F\}$.

Definition 2.5. Given a certain $p \in P$, $t \in T$, and $q \in Q$, the following subsets are defined:

i. The data preset of a place, ${}^\circ p = \{q \in Q | (q, p) \in F\}$;
ii. The data postset of a place, $p^\circ = \{q \in Q | (p, q) \in F\}$;
iii. The data preset of a transition, ${}^\circ t = \{q \in Q | (q, t) \in F\}$;
iv. The data postset of a transition, $t^\circ = \{q \in Q | (t, q) \in F\}$;
v. The data preset of a hull, ${}^\circ q = \{p \in P | (p, q) \in F\}$;
vi. The data postset of a hull, $q^\circ = \{p \in P | (q, p) \in F\}$.

Inherited from classical PNs [Murata 1989], a transition t, such that ${}^\bullet t = \emptyset$ is called *source transition* and a transition t such that $t^\bullet = \emptyset$ is called *sink transition*. The same pattern is followed in the data domain, i.e., we name *source hull* a hull with ${}^\circ q = \emptyset$, while its dual ($q^\circ = \emptyset$) is called *sink hull*. There are, however, some issues to be considered with regard to these four special cases. For instance, a sink hull $q \in Q$ may only exist[1] if there is at least one transition $t \in q^\bullet$, such that $G(t) \neq \top$.

The control and data flow of an embedded system are not completely independent. On the contrary, a model which deals with embedded systems must have a mechanism that allows the representation of *interdomain* effects. By interdomain effects we mean the influence of the control flow over the data flow and vice versa, e.g., the execution of a conditional branch (where the next operation to be executed is data dependent). Such influence is modeled in DFN by means of arcs in $T \times Q$ and $Q \times T$ and the guard function G. The guard function G plays an important role in the behavioral DFN model, since it allows a transition t to have a functionality that not only depends on the control domain of the model, but also on its data domain. Section 2.3 elaborates this concept in further detail.

[1]In order to avoid performing an operation (c.f. Section 2.3) that will not be used, neither in the data domain (a place $p \in q^\circ$) nor in the control domain (a transition $t \in q^\bullet$).

2.2 DFN Marking Functions

Bridging the gap between the structural (Section 2.1) and behavioral (Section 2.3) models has lead to the current section, where the dynamical aspects of the modeling are tackled. This section puts forward a new concept for the state evolution.

Bipartite graph-based models, such as ETPN or PRES+, analyze the state space evolution by means of a marking function that is isomorphic to $\overrightarrow{\mu} : P \mapsto \mathbf{N}$. Instead, we propose to make use of a tripartite graph and, consequently, make the marking function isomorphic to $\overrightarrow{\mu} : P \mapsto \mathbb{Z}^*_{[\mathbf{i}]}$, where $\mathbb{Z}^*_{[\mathbf{i}]}$ is a new numeric system based on Gaussian integers [Dimiev and Markov 2002] and polar representation of complex numbers (Definition 2.6). Because of its two-dimensional nature, the *Modified Gaussian Integer* (MGI) notation defined below combines two independent parameters and is suitable for capturing both control and data flows.

Definition 2.6. A modified Gaussian integer is a tuple $\mathbb{Z}^*_{[\mathbf{i}]} = \langle \rho, \theta \rangle$,

where: $\rho \in \mathbb{Z}$ $\theta \in \mathbb{Z}_n$ $n \in \mathbf{N}$
$\mathbb{Z}_n = \{0, 1, 2, \ldots, (n-1)\}$
$\theta + n = \theta$

The inherent periodicity that Definition 2.6 imposes on θ is beneficial for two reasons: first, the data domain stays bounded, as in real-life applications, where registers do not have an infinite capacity and, second, the same overflow behavior occurs when an arithmetic operation exceeds the capacity of a register. Thus, the applicability of this numeric system to the codomain of a marking function comes from using $\rho\, \mathbf{e}^{\mathbf{i}\cdot\theta}$ to indicate that θ is the periodic part (hence, applied to capture data), while ρ is an integer used in the control domain.

Definition 2.7. The marking function $\overrightarrow{\mu} : P \mapsto \mathbb{Z}^*_{[\mathbf{i}]}$ captures the state of the system. The marking $\mu(p)$ of a place $p \in P$, also called *marked place*, is:

$$\mu(p) = \gamma \cdot \mathbf{e}^{\mathbf{i}\cdot 2\pi \cdot \frac{\alpha}{R(p)}}$$

where $\gamma \in \mathbf{N}$, $\alpha \in \mathbb{Z}$, and $R(p)$ is defined in 2.8. The following notation is used hereafter:

$$\overrightarrow{\mu} = \begin{bmatrix} \mu(p_1) \\ \mu(p_2) \\ \vdots \\ \mu(p_n) \end{bmatrix} = (\mu(p_1), \mu(p_2), \cdots, \mu(p_n))^{\mathrm{T}}$$

where T denotes the *transpose* operation.

The new marking function scheme introduced in Definition 2.7 is based on a subset of complex numbers, i.e., the MGI numeric system defined in 2.6. Unlike classic PNs [Murata 1989], the DFN marking function defined in 2.7 is capable of considering the effects of both control and data domains when analyzing the dynamics of a system, because of its extended structure that allows two independent but related sets of *quanta*, i.e., γ and α, to share a place at

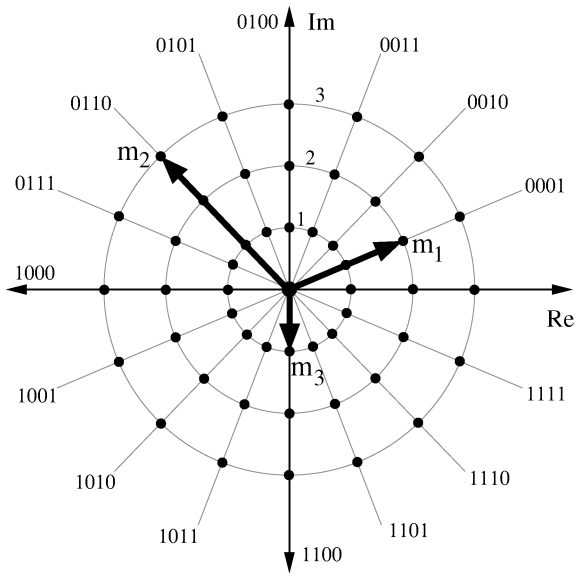


Fig. 4. Complex plane mapping of the state space.

any time. A search through the state space can thereby combine effects of two disjoint domains, the control domain captured as the *modulus* $\gamma = |\mu(p)|$ of a complex number and the data domain as its *argument* $\alpha = \angle\mu(p)$. In this way, a marked place $\mu(p)$ contains all the information needed for the net $T \cup Q$ to perform modifications in the state of the system. A transition $t \in T$ operates at the *modulus* level of $\mu(p)$ while hulls $q \in Q$ tackle the *argument*. Next, Definition 2.8 introduces three attributes associated to a place, which are used in both control and data domain in order to bound the representation.

Definition 2.8.

$K(p)$ is the *capacity* of a place, which is the maximum number allowed in $|\mu(p)|$;
$R(p)$ is the *range* of a place, which is the cardinality of the set of values that conforms $\angle\mu(p)$;
$L(p) = \log_2(R(p))$ is the *length* of a place.

For the sake of clarity, we illustrate these three limits using the example presented in Figure 4. The state space of the system is limited in the control domain to $K(p_i) = 3$ and in the data domain to $R(p_i) = 16$, and $L(p_i) = 4$.

Furthermore, having a marking function defined in terms of a periodic domain allows for a more natural representation of the embedded system hardware. For instance, an ALU performing an operation that exceeds the capacity of a register will produce a truncated result. The same effect can be observed in the DFN model, if the argument of a place is overflowed, because:

$$\cdots = \alpha - 2 \cdot R(p) = \alpha - R(p) = \alpha = \alpha + R(p) = \alpha + 2 \cdot R(p) = \cdots \qquad (1)$$

Definition 2.7 has expressed $\mu(p)$ by means of the constant $R(p)$. However, it is more useful to redefine $\mu(p)$ incorporating other parts of Definition 2.8 and, thus, linking it to the constant $L(p)$ instead. This result is shown below:

$$\mu(p) = \gamma\, \mathrm{e}^{\mathbf{i}\cdot 2\pi\cdot\frac{\alpha}{R(p)}} = \gamma\, \mathrm{e}^{\mathbf{i}\cdot 2\pi\cdot\frac{\alpha}{2^{L(p)}}} = \gamma\, \mathrm{e}^{\mathbf{i}\cdot\frac{\pi\cdot\alpha}{2^{L(p)-1}}} = \gamma\, \mathrm{e}^{\mathbf{i}\cdot(2^{1-L(p)}\cdot\pi)\alpha} \tag{2}$$

Thus far, Figure 4 exemplifies the principles hitherto introduced, by showing the representation of a marking function in a complex plane. The DFN model corresponds to a system that has been captured by means of three places p_1, p_2, and p_3. Since a marking m_i of a place p_i, i.e., $m_i = \mu(p_i)$, $1 \leq i \leq 3$, has modulus $\gamma \in \mathbf{N}$ and argument $\alpha \in \mathbb{Z}$, it is represented as a vector in a quantized complex plane. The *magnitude* of this vector is obtained from the number of tokens of the marked place and there is a direct relation between the vector *angle* and the number of data quanta in the marked place. Moreover, this can be visualized as moving radially through a line when performing changes in the control quantum γ of a marked place, while changes in data quantum α are equivalent to a move through concentric circles of constant radius –determined by the number of tokens γ.

In order to explain Definitions 2.7 and 2.8, and Eqs. (1) and (2), associated with the state space model, the example given in Figure 4 illustrates the state where place p_1 contains $\gamma = 2$ tokens and $\alpha = 1$ data quantum, place p_2 has $\gamma = 3$ tokens and $\alpha = 6$ data quanta and p_3 has $\gamma = 1$ token and $\alpha = -4$ data quanta. Likewise, each marked place is:

$$\mu(p_1) = 2\mathrm{e}^{\mathbf{i}\cdot\frac{\pi}{8}}; \quad \mu(p_2) = 3\mathrm{e}^{\mathbf{i}\cdot\frac{\pi}{8}\cdot 6}; \quad \mu(p_3) = 1\mathrm{e}^{-\mathbf{i}\cdot\frac{\pi}{8}\cdot 4} = 1\mathrm{e}^{-\mathbf{i}\cdot\frac{\pi}{2}}$$

2.3 DFN Behavioral Model

Having introduced both the structure and the state space of DFN models, it is subsequently possible to analyze the behavior of such models using the principles introduced in this section. This section introduces four definitions that state the behavior of DFN models. The behavior of a DFN model is described in terms of enabling and firing transitions, as in classic Petri nets, in addition to a synchronized data-flow operation scheme. The following two definitions introduce the rules that ensue from modifying the classic enabling and firing rules, in order to allow a marking function defined in the $\mathbb{Z}^*_{[\mathbf{i}]}$ domain.

Definition 2.9. A transition t is said to be *enabled*, for a given marking μ_k, if the following two conditions are met:

i. all places in preset $p_i \in {}^\bullet t$ contain at least $W(p_i, t)$ tokens, which is:

$$\bigwedge_{p_i \in {}^\bullet t} \left(|\mu(p_i)| \geq W(p_i, t)\right)$$

ii. the following atomic proposition holds: "the relation between the data quanta that affects all hulls in the preset $q_j \in {}^\circ t$, and 0, is given by the result of the guard function $G(t)$." Formally:

$$\bigwedge_{q_j \in {}^\circ t} AP\left(\sum_{\ell} \angle\mu(p_\ell)\cdot W(p_\ell, q_j) + H(q_j), G(t), 0\right)$$

Definition 2.9 states whether a transition is enabled or not, in the DFN models. The influence of both control and data flow aspects in this evaluation can be observed from the combined form of the enabling condition. Thus, from the control flow point of view, the enabling of a transition depends on the token distribution throughout the DFN model, i.e., subpart (*i*) of the definition. From the data flow point of view, the dependence is established in subpart (*ii*), by the conjunction of atomic propositions AP, which take data quanta as an argument. The summation (over ℓ) in the argument of AP is explained further in Definition 2.11.

Definition 2.10. The firing of an enabled transition t_j changes a marking μ_k into μ_{k+1} by means of the following rules:

i. A finite number of tokens are removed from $p_i \in {}^\bullet t_j$:
$$|\mu_{k+1}(p_i)| = |\mu_k(p_i)| - W(p_i, t_j),\ \forall p_i \in {}^\bullet t_j$$
ii. A finite number of tokens are added to $p_i \in t_j{}^\bullet$:
$$|\mu_{k+1}(p_i)| = |\mu_k(p_i)| + W(t_j, p_i),\ \forall p_i \in t_j{}^\bullet$$
iii. Each hull $q \in t_j{}^\circ$ is executed (c.f. Definition 2.11).

Hulls capture the data flow behavior of a DFN, as shown in Definition 2.11. In simple terms, the hull performs a summation of data quanta over the data domain. If the summation contains only one term, i.e., $|{}^\circ q| = 1$, it turns out to be a simple *move* operation. From the behavioral point of view, the execution of each hull q is synchronized with some transition t in the net. Therefore, no hull q can fire nondeterministically.

Definition 2.11. The firing of any transition $t \in {}^\bullet q$ produces the *execution* of the hull q, which changes a marking μ_k into μ_{k+1} as follows:

$$\angle \mu_{k+1}(p_j) = W(q, p_j) \cdot \left(\sum_i \angle \mu_k(p_i) \cdot W(p_i, q) + H(q) \right) \tag{3}$$

where $p_i \in {}^\circ q$ and $p_j \in q^\circ$.

Thus far, the definitions introduced in this section are based on classic Petri net notation and *linear algebra*. The control part of the DFN model is semantically equivalent to a Petri net, while the data part is modeled in terms of basic linear operators. By combining these two effects, we obtained a model of computation that allows representations to be very close to the final implementation of the system. However, this accuracy is achieved at the expense of additional aid for decisions at the system level.

2.4 Example

The DFN model of the multiplier described in Section 1.2 is shown in Figure 5. Places p_1 and p_2 are set to contain, in the argument of their markings, the multiplier operands a and b, respectively. One token is placed in each of these two places as an indication of the validity of such data quanta. This initial marking μ_0, captures the behavior of the en signal, in the sense that t_1 is only allowed to fire when *both* p_1 and p_2 have a token.

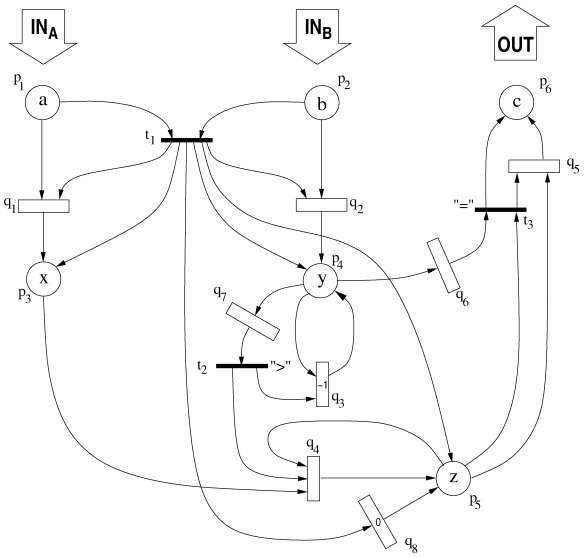


Fig. 5. DFN model of the multiplier.

In order to perform a multiplication, there is a sequence of events which sets the argument of p_6, i.e., $\angle\mu_k(p_6)$, into the product $\angle\mu_0(p_1) \cdot \angle\mu_0(p_2)$. This sequence is:

1. Transition t_1 fires. Therefore,
   (a) The two tokens are removed from p_1 and p_2 —Definition 2.10.1.
   (b) One token is placed in p_3, other in p_4, and another in p_5 — Definition 2.10.2.
   (c) The two hulls belonging to the set ${t_1}^\circ = \{q_1, q_2\}$ are executed accordingly with Eq. (3) —Definition 2.10.3.
2. Hulls q_1 and q_2 execute, according to Definition 2.11, copying the arguments from p_1 and p_2 into p_3 and p_4 respectively, i.e., the application of Eq. (3) when $|{}^\circ q| = |q^\circ| = 1$. Thus,

$$\begin{aligned}\angle\mu_1(p_3) &= W(q_1, p_3) \cdot (\angle\mu_0(p_1) \cdot W(p_1, q_1) + H(q_1)) \\ &= 1 \cdot (a \cdot 1 + 0) \\ &= a\end{aligned}$$

$$\begin{aligned}\angle\mu_1(p_4) &= W(q_2, p_4) \cdot (\angle\mu_0(p_2) \cdot W(p_2, q_2) + H(q_2)) \\ &= 1 \cdot (b \cdot 1 + 0) \\ &= b\end{aligned}$$

3. The new marking $\overrightarrow{\mu_1} = (0\mathrm{e}^{\mathbf{i}\cdot a}, 0\mathrm{e}^{\mathbf{i}\cdot b}, 1\mathrm{e}^{\mathbf{i}\cdot a}, 1\mathrm{e}^{\mathbf{i}\cdot b}, 1\mathrm{e}^{\mathbf{i}\cdot 0}, 0\mathrm{e}^{\mathbf{i}\cdot 0})^{\mathrm{T}}$ causes the transitions t_2 to fire repeatedly b times and, for each iteration, $\angle\mu(p_5)$ is incremented by a. This leads to obtaining $a \cdot b$ in the final value of $\angle\mu(p_5)$ as follows:
   (a) Since $\mu_1(p_4)$ has an argument greater than 0, t_2 is enabled according to Definition 2.9. Note the influence of $\angle\mu_1(p_4)$ in the enabling rule of the definition, through an atomic proposition AP based on ${}^\circ t_2 = \{q_7\}$ and $G(t_2) =$“$>$.”

(b) Firing the enabled transition t_2 produces the subsequent execution of two hulls: q_3 and q_4.

(c) The execution of q_3 always leads to an unitary decrement of the argument in p_4, since $|{}^{\circ}q_3| = 1$ and $H(q_3) = -1$. This is:

$$\begin{aligned} \angle\mu_{k+1}(p_4) &= W(q_3, p_4) \cdot (\angle\mu_k(p_4) \cdot W(p_4, q_3) + H(q_3)) \\ &= 1 \cdot (\angle\mu_k(p_4) - 1) \\ &= \angle\mu_k(p_4) - 1 \end{aligned}$$

For $k = 1 \cdots b + 1$.

(d) The execution of q_4 provides the system with iterative additions in the scope of p_5. This is:

$$\begin{aligned} \angle\mu_{k+1}(p_5) &= W(q_4, p_5) \cdot (\angle\mu_k(p_3) \cdot W(p_3, q_4) + \angle\mu_k(p_5) \cdot W(p_5, q_4) \\ &\quad + H(q_4)) \\ &= 1 \cdot (\angle\mu_k(p_3) + \angle\mu_k(p_5) + 0) \\ &= \angle\mu_k(p_3) + \angle\mu_k(p_5) \end{aligned}$$

For $k = 1 \cdots b + 1$.

(e) This is:

$$\begin{aligned} \overrightarrow{\mu_1} &= (0\mathrm{e}^{\mathbf{i}\cdot a}, 0\mathrm{e}^{\mathbf{i}\cdot b}, 1\mathrm{e}^{\mathbf{i}\cdot a}, 1\mathrm{e}^{\mathbf{i}\cdot b}, 1\mathrm{e}^{\mathbf{i}\cdot 0}, 0\mathrm{e}^{\mathbf{i}\cdot 0})^{\mathrm{T}} \\ \overrightarrow{\mu_2} &= (0\mathrm{e}^{\mathbf{i}\cdot a}, 0\mathrm{e}^{\mathbf{i}\cdot b}, 1\mathrm{e}^{\mathbf{i}\cdot a}, 1\mathrm{e}^{\mathbf{i}\cdot b-1}, 1\mathrm{e}^{\mathbf{i}\cdot a}, 0\mathrm{e}^{\mathbf{i}\cdot 0})^{\mathrm{T}} \\ \overrightarrow{\mu_3} &= (0\mathrm{e}^{\mathbf{i}\cdot a}, 0\mathrm{e}^{\mathbf{i}\cdot b}, 1\mathrm{e}^{\mathbf{i}\cdot a}, 1\mathrm{e}^{\mathbf{i}\cdot b-2}, 1\mathrm{e}^{\mathbf{i}\cdot 2\cdot a}, 0\mathrm{e}^{\mathbf{i}\cdot 0})^{\mathrm{T}} \\ &\vdots \\ \overrightarrow{\mu_{b-1}} &= (0\mathrm{e}^{\mathbf{i}\cdot a}, 0\mathrm{e}^{\mathbf{i}\cdot b}, 1\mathrm{e}^{\mathbf{i}\cdot a}, 1\mathrm{e}^{\mathbf{i}\cdot 2}, 1\mathrm{e}^{\mathbf{i}\cdot (b-2)\cdot a}, 0\mathrm{e}^{\mathbf{i}\cdot 0})^{\mathrm{T}} \\ \overrightarrow{\mu_b} &= (0\mathrm{e}^{\mathbf{i}\cdot a}, 0\mathrm{e}^{\mathbf{i}\cdot b}, 1\mathrm{e}^{\mathbf{i}\cdot a}, 1\mathrm{e}^{\mathbf{i}\cdot 1}, 1\mathrm{e}^{\mathbf{i}\cdot (b-1)\cdot a}, 0\mathrm{e}^{\mathbf{i}\cdot 0})^{\mathrm{T}} \\ \overrightarrow{\mu_{b+1}} &= (0\mathrm{e}^{\mathbf{i}\cdot a}, 0\mathrm{e}^{\mathbf{i}\cdot b}, 1\mathrm{e}^{\mathbf{i}\cdot a}, 1\mathrm{e}^{\mathbf{i}\cdot 0}, 1\mathrm{e}^{\mathbf{i}\cdot b\cdot a}, 0\mathrm{e}^{\mathbf{i}\cdot 0})^{\mathrm{T}} \end{aligned}$$

(f) When $\angle\mu(p_4)$ reaches the value of 0, only t_3 is allowed to fire next, deviating the control flow toward the end of the procedure.

4. The firing of t_3 does not only put a token in p_6, because of $p_6 \in t_3{}^{\bullet}$, but also sets $\angle\mu(p_6)$ to the value in $\angle\mu(p_5)$, because of q_5 executing.

2.4.1 *Trace for* $5 \times 3 = 15$. For the sake of comparison, a detailed trace of the $5 \times 3 = 15$ execution of the multiplier's DFN model is presented below. The initial marking $\overrightarrow{\mu_0} = (1\mathrm{e}^{\mathbf{i}\cdot 5}, 1\mathrm{e}^{\mathbf{i}\cdot 3}, 0\mathrm{e}^{\mathbf{i}\cdot 0}, 0\mathrm{e}^{\mathbf{i}\cdot 0}, 0\mathrm{e}^{\mathbf{i}\cdot 0}, 0\mathrm{e}^{\mathbf{i}\cdot 0})^{\mathrm{T}}$ implies that only t_1 is enabled. Thus:

t_1; $p_3 \leftarrow 1\mathrm{e}^{\mathbf{i}\cdot 5}$| $p_4 \leftarrow 1\mathrm{e}^{\mathbf{i}\cdot 3}$| $p_5 \leftarrow 1\mathrm{e}^{\mathbf{i}\cdot 0}$; $t_2 \leftarrow 1\mathrm{e}^{\mathbf{i}\cdot 3}$| CMP| t_2; SUB| $p_4 \leftarrow 1\mathrm{e}^{\mathbf{i}\cdot 2}$| SUM| $p_5 \leftarrow 1\mathrm{e}^{\mathbf{i}\cdot 5}$; $t_2 \leftarrow 1\mathrm{e}^{\mathbf{i}\cdot 2}$| CMP| t_2; SUB| $p_4 \leftarrow 1\mathrm{e}^{\mathbf{i}\cdot 1}$| SUM| $p_5 \leftarrow 1\mathrm{e}^{\mathbf{i}\cdot 10}$; $t_2 \leftarrow 1\mathrm{e}^{\mathbf{i}\cdot 1}$| CMP| t_2; SUB| $p_4 \leftarrow 1\mathrm{e}^{\mathbf{i}\cdot 0}$| SUM| $p_5 \leftarrow 1\mathrm{e}^{\mathbf{i}\cdot 15}$; $t_3 \leftarrow 1\mathrm{e}^{\mathbf{i}\cdot 0}$| CMP| t_3; $p_6 \leftarrow 1\mathrm{e}^{\mathbf{i}\cdot 15}$

It can be observed, by comparison with the trace obtained in Section 1.2, that the number of changes between control and data environments has been reduced, i.e., less ";" symbols than previously. In the former ETPN trace, there are 18 changes, while only 9 are counted in this DFN trace.

2.5 Remarks

The control/data-flow interaction problem is not new and has been previously addressed [Peng and Kuchcinski 1994; Strehl et al. 2001]. These approaches are useful for synthesis issues, such as scheduling and allocation, but have limited application in behavioral analysis. When the interaction between control and data flows is intensified, these approaches tend to increase their complexity as outlined in Section 1.2. Therefore, Section 2 has been promoting a modeling technique applicable to the design of embedded system, which exploits the interactions between control and data flow in order to reduce complexity and yet add expressiveness to the model. An increase on the expressive power of a model usually requires more resources and, therefore, verification complexity is magnified. What this paper exploits is the fact that by making the data domain periodic, the designer does not loose information, because that is the way the application is meant to behave, whereas it gains from the analysis point of view: the model has become data bounded. This contrasts traditional approaches, where data boundedness is achieved by limiting the size of the state space, with possible loss of information.

In order to broaden this point of view, the DFN model with other *nontripartite* approach should be compared. For instance, the PRES+ model is based on the same principles of Coloured Petri Net, i.e., it consists of an underlying *bipartite* graph and has an extended semantics to cope with dataflow manipulation. The data flow of the system is handled by associating two parameters to each token of the net: a value (v) and a time stamp (t). To illustrate what is meant by reducing complexity without compromising the model's expressiveness, Figure 6 shows a PRES+ representation of the multiplier example discussed in Section 1.2, and compares it with the DFN model in Section 2.4. The inputs to the multiplier are places A and B, while its output is bound to place C. The initialization of the multiplier is carried out by firing t_1, t_2, and t_3, which transfers the token in A and B to X and Y, respectively (for the first two transitions), and sets the value of the token in Z to 0 (for the firing of t_3). The main loop of the multiplier is captured by t_7, t_8, and t_9, where the guard function [y>0], and t_9's transition function y-1, indicate that the loop will be repeated Y times. The result is taken out to place C, when the guard function [y=0] allows t_{10} to fire.

Comparing Figures 5 and 6, it can be observed that having a tripartite instead of bipartite graph is advantageous. Clearly, modeling based on a bipartite graph (e.g., PRES+) requires intricate manipulations to cope with control/dataflow modeling, since this is not inherent in the model. By identifying which elements are of control nature and which ones belong to the data flow, DFN is capable of reducing unnecessary complexity and does not compromise the quality of the model in terms of expressiveness.

3. FORMAL VERIFICATION

Models are often validated by means of simulation, where the behavior of the system is checked against a certain set of stimuli and, therefore, only a limited part of the state space is searched. Formal verification, on the other hand, mathematically checks *for all possible behaviors* whether the model corresponds

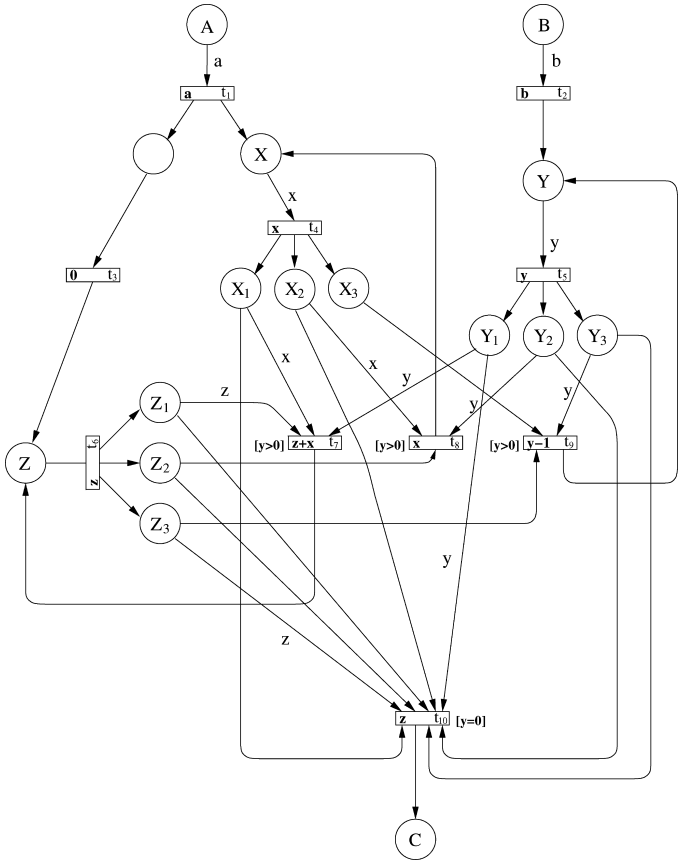


Fig. 6. PRES+ model of the multiplier.

to the given specification. A comprehensive overview of the state of the art of formal verification is given in Wang [2004].

The combination of atomic propositions (APs), such as the ones described in Section 2, is carried out using a finite set of logical operators, e.g., $\neg$, $\wedge$, and $\Longrightarrow$. This is known as Propositional Logics (PL). In addition, Modal Logics (ML) also introduce the concept of path quantification [Emerson 1990], through the universal ($\forall$) and existential ($\exists$) operators. However, since we are interested in state evolution analysis (which implies that the AP is allowed to change over the time), it is important to consider a type of logic that not only expresses several execution traces, as ML, but is also capable of capturing properties that vary along the time. Temporal Logic (TL) is a class of ML which provides a mechanism for analysis of assertions over time by means of an additional set of operators, called *temporal quantifiers* [Bellini et al. 2000], which describe the behavior of dynamic assertions along the time. For the rest of this paragraph, we assume that φ is either an AP or an already known property. Thus, a property, which holds at the **next** state, is represented by $\bigcirc\varphi_1$, a property that will **eventually** hold is symbolized by $\diamond\varphi_1$, if it **always** holds by $\Box\varphi_1$, and $\varphi_1 \mathcal{U} \varphi_2$ is used to represent that φ_1 holds **until** φ_2 holds.

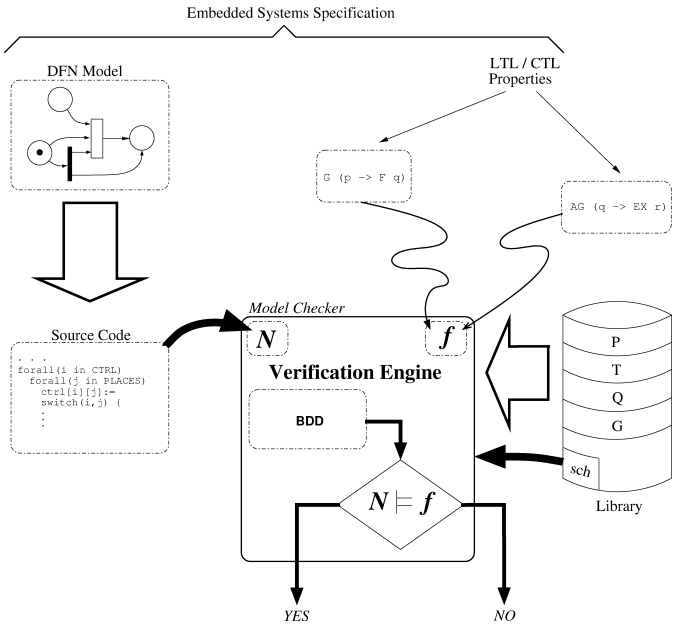


Fig. 7. Our model checking methodology.

There are two possible time classifications for TL properties: (a) assuming a state evolution where only one possible next state can be assigned for any given state, i.e., Linear Temporal Logics (LTL), or (b) taking into account possible splits in the state evolution, i.e., Branching Temporal Logics (BTL). Computational Tree Logic (CTL) is a BTL restricted by the constraint that each temporal operator *must* be preceded by a path quantifier. For further reference, the reader is referred to Emerson and Halpern [1986], Kupferman and Vardi [1995], Pnueli [1985], and Vardi [2001].

3.1 Model Checking of DFN models

Model Checking [McMillan 1993; Clarke et al. 1999; Burch et al. 1992] is a formal verification technique that has had a number of success stories, e.g., Burch et al. [1992], which certainly attracted the EDA industry. This section outlines our approach to formally verify DFN models via model checking. Figure 7 shows a *verification engine* as the core of the proposed methodology, which has been chosen to be the Cadence SMV model checker [Cadence 2001] for its robustness and expressiveness. However, it should not be inferred that our methodology is restricted to this particular tool. The methodology's input is an embedded system specification composed of both a DFN model and a set of properties expressed in a temporal logic. The DFN model is translated into a *source code* understood by the model checker while, on the other hand, a library written in the tool's language captures the DFN semantics according to the definitions introduced in the previous section. The verification is driven by a scheduler (`sch`), which determines a valid sequence of transition firings in order to analyze the resulting behavior. The outcome of the verification (*YES* or *NO* in Figure 7) is

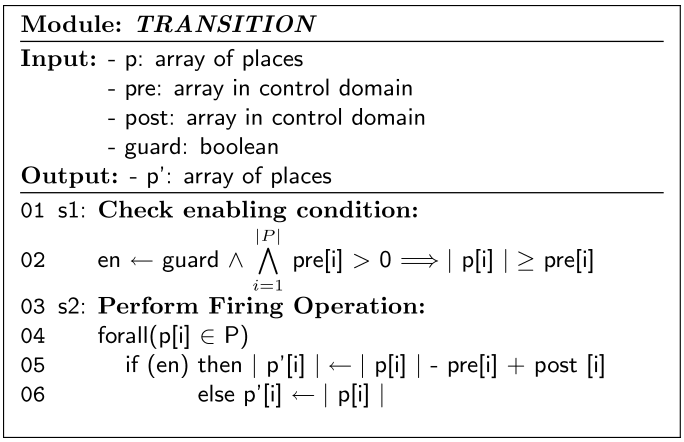


Fig. 8. Algorithm for a DFN transition.

the result of evaluating the correctness of the DFN model with regard to the temporal logic properties.

A library that captures the semantics of DFN has been implemented in order to perform the DFN model checking. This library consists of four modules (*place()*, *transition()*, *hull()* and *guard()*), and a scheduler (`sch`). The structure of the net is built by means of successive instantiations to these four modules. Since an enabled transition is not forced to fire, the scheduler has been defined in a way that restricts each step of the schedule to the set of *enabled* transitions, i.e., the scheduler nondeterministically chooses from the set of enabled transitions the next transition to fire. The code for such a scheduler is presented below, where `M`$= |T|$ is the number of control transitions in the net. A variable `sch` of order `M`$= |T|$ indicates which transition `t[sch]` is firing at time k. The last line defines the firing of the transition `t[sch]` by changing the marking μ of the system. Note that `pp` is the output of the *transition()* module, i.e., p' in Figure 8. As a consequence, there is a change in the enabled transitions, i.e., `t[i].en=1`, which influences the nondeterministic value assigned to `sch = {i : i = 1..(M), t[i].en}`.

```
typedef TRANSITIONS 1..(M);
...
sch : TRANSITIONS;
sch := { i : i = 1..(M), t[i].en };
...
forall (i in PLACES)
   next(p[i].modulus) := t[sch].pp[i];
```

In the following subsections, we introduce three modules of the library, where the *place()* module is a data structure, and the algorithms in the *transition()* and *hull()* modules are presented. The remaining *guard()* module is a `case` statement that returns the evaluation of a condition, based on the symbol used at its argument.

3.1.1 *Places.* Because of the underlying complex notation used in Definition 2.7, the set P cannot be constructed as an array of integers as in classic Petri nets. In the implementation of the DFN library, an array `p[i]` where each member comprise a two-member structure, is the basic data type that models the modulus and argument of a marked place. The extraction of each part of the complex tuple, i.e., the application of the $|\mu(p)|$ and $\angle\mu(p)$ operators in order to obtain modulus and argument, respectively, is performed through a direct access to the member of the structure stored within the array `p[i]`.

3.1.2 *Transitions.* The algorithm presented in Figure 8 consists of two main parts: checking if the transition is enabled (s1) and the firing operation (s2), which refer to Definitions 2.9 and 2.10, respectively. When a transition is selected by the scheduler, an assignment occurs in all places of the net. If the transition determined by the scheduler (`t[sch]`) is enabled, the assignment carried out in (s2) considers the effect of the firing rule presented above.

For the sake of clarity, assume a *transition()* module instantiated as follows:

```
t[1]: transition(p,[3,1,0,2],[0,1,2,0],1);
```

This means that:

$^\bullet t_1 = \{p_1, p_2, p_4\}$;
$W(p_1, t_1) = 3, W(p_2, t_1) = 1, W(p_4, t_1) = 2$;
$t_1{}^\bullet = \{p_2, p_3\}$;
$W(t_1, p_2) = 1, W(t_1, p_3) = 2$;
and
No guard function exists (since 1 is the default value on the result of a *guard()* module)

With regard to Definition 2.9, which is implemented through the enabling condition part (s1), the following logical condition:

$$\text{guard} \wedge \bigwedge_{i=1}^{|P|} \text{pre[i]} > 0 \Longrightarrow |\,\text{p[i]}| \geq \text{pre[i]}$$

is unfolded into:

$$\begin{aligned} \top \wedge |\,\text{p[1]}| \geq \text{pre[1]} \wedge |\,\text{p[2]}| \geq \text{pre[2]} \wedge |\,\text{p[4]}| \geq \text{pre[4]} \\ \equiv \top \wedge |\,\text{p[1]}| \geq 3 \wedge |\,\text{p[2]}| \geq 1 \wedge |\,\text{p[4]}| \geq 2 \end{aligned} \tag{4}$$

The boolean result of Eq. (4) is used in part s2 to see whether the next step of the argument of each marked place, or whether its former content, is assigned to the result of the firing rule. Note the connection between the scheduler's assignment and the output of the *transition()* module.

3.1.3 *Hulls.* Similarly, Figure 9 shows the algorithm for a hull $q \in Q$. The enabling part s1 checks whether the transition that is firing at a given step k (i.e., `t[sch]`) has any connection to the hull itself. This is because of Definition 2.10.3. On the other hand, the action described in Definition 2.11 is implemented in the firing part s2, which allows the next state of the argument

Module: ***HULL***

Input: - p: array of places
- pre: array in data domain
- post: array in data domain
- FA: set of arcs in $(T \times Q)$

Output: - p': array of places

```
01 s1: Check enabling condition:
02     if (output of the scheduler ∈ FA) &(there is no deadlock)
03     then
04         en ← ⊤
05     else en ← ⊥
06 s2: Perform Firing Operation:
07     forall(p[i] ∈ P)
08         s ← Σ_{i=1}^{|P|} ∠ p[i] · pre[i]
09     forall(p[i] ∈ P)
10         if (en) then ∠ p'[i] ← post[i] · s
11                 else p'[i] ← ∠ p[i]
```

Fig. 9. Algorithm for a DFN hull.

in p_i to be set to either (a) a sum of values coming from the preset, or (b) its previous content.

An implementation of the multiplier described in Section 2.4 is given below, in order to clarify the concepts hitherto introduced. This SMV code utilizes the `dfn.smv` library, which defines all elements of a DFN model in the way described in this section. The aforementioned library also includes the nondeterministic scheduler (described in Section 3.1) and a deadlock definition (used as last argument of `hull()`). The property to be verified is $\diamond(\angle\mu(p_6) = a \cdot b)$, which means that the multiplier's output will *eventually* reach the value of a times b. Since a and b inputs are defined as abstract variables in the range of `0..7`, all 256 combinations are taken into account for proving the design correct. This verification was performed in 5.23 sec, allocating 163007 BDD nodes.

```
#define Kc 3                /* capacity in the ctrl domain */
#define Kd 64               /* capacity in the data domain */

#define NN 6                /* |P| */
#define MM 3                /* |T| */
#define HH 8                /* |Q| */

#include "dfn.smv"

abstract a : 0..7;  next(a):=a;
abstract b : 0..7;  next(b):=b;

/* INITIAL MARKING */
ictrl := [1,1,0,0,0,0];
idata := [a,b,0,0,0,0];

/* NET */
g1 : guard(p[4],gt,0);
g2 : guard(p[4],eq,0);
```

```
t[1] : transition(p,[1,1,0,0,0,0],[0,0,1,1,1,0],1);
t[2] : transition(p,[0,0,0,0,0,0],[0,0,0,0,0,0],g1);
t[3] : transition(p,[0,0,0,0,1,0],[0,0,0,0,0,1],g2);

q[1] : hull(p,[1,0,0,0,0,0],[0,0,1,0,0,0],{1},0,sch,deadlock);
q[2] : hull(p,[0,1,0,0,0,0],[0,0,0,1,0,0],{1},0,sch,deadlock);
q[3] : hull(p,[0,0,0,1,0,0],[0,0,0,1,0,0],{2},-1,sch,deadlock);
q[4] : hull(p,[0,0,1,0,1,0],[0,0,0,0,1,0],{2},0,sch,deadlock);
q[5] : hull(p,[0,0,0,0,1,0],[0,0,0,0,0,1],{3},0,sch,deadlock);
q[6] : hull(p,[0,0,0,1,0,0],[0,0,0,0,0,0],{0},0,0,deadlock);
q[7] : hull(p,[0,0,0,1,0,0],[0,0,0,0,0,0],{0},0,0,deadlock);
q[8] : hull(p,[0,0,0,0,0,0],[0,0,0,0,1,0],{1},0,sch,deadlock);

/* SPEC */
--FirstStep : assert X (p[4].modulus = 1);
Result : assert F (p[6].phase = a * b);

END
```

3.2 Behavioral Properties

This section presents a brief description of the applicability of the methodology introduced in Section 3.1 to verify behavioral properties that are of interest for the verification of embedded systems, such as reachability, safety, and liveness. Furthermore, the relation of both LTL and CTL notation to such behavioral properties is discussed.

3.2.1 *Reachability.* A marking μ_k is said to be reachable from a marking μ_0 if there is a sequence of transition firings, which leads to μ_k. We use CTL formulas to express this condition as:

$$\varphi_r = \exists\Diamond \bigwedge_{i=1}^{|P|} \big(\mu(p_i) = c_i\big)$$

where $c_i = b_i\, \mathrm{e}^{\mathbf{i}\cdot a_i}$ is the desired final marking $\mu_k(p_i), \forall p_i \in P$ at the time step k. Therefore, if both $|\mu(p_i)| = b_i$ and $\angle\mu(p_i) = a_i$ holds, then the state of the system has *eventually* reached a marking of $\mu_k(p_i), \forall 1 \leq i \leq n$.

3.2.2 *Safety.* Safety properties are conditions that are verified along any execution path. These type of properties are usually associated with some critical behavior, and, thereby, should *always* hold. A particular type of safety property is known, in the context of Petri nets, as *safeness*. Classically, a *safe* PN allows, at most, one token in every place, for any reachable marking, which means that the following LTL formula holds:

$$\varphi_s = \Box \bigwedge_{i=1}^{|P|} \big(|\mu(p_i)| \leq 1\big)$$

3.2.3 *Liveness.* A DFN model that never changes its marking is likely to be of very little interest. Thus, liveness properties indicate that a certain DFN model would not get trapped into a single marking (or a particular cycle

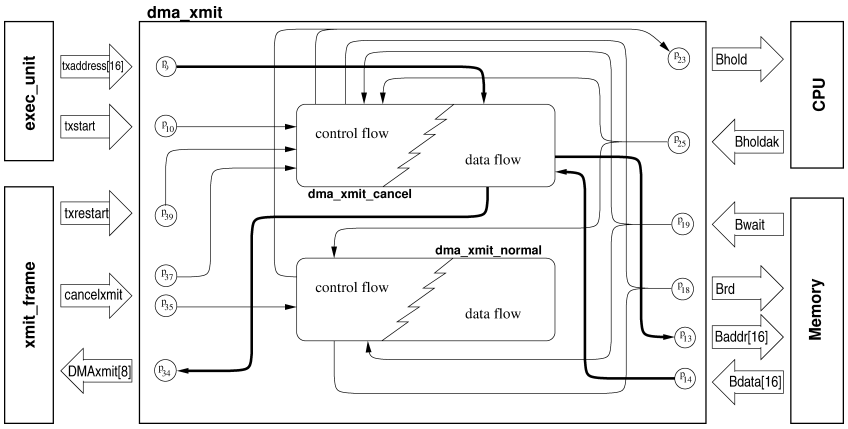


Fig. 10. DMA transmitter/receiver of the Ethernet coprocessor and signals involved.

defined by a limited set of markings). The absence of deadlocks is a fundamental liveness property in the theory of Petri nets. For the model checking of a DFN model, the deadlock condition is expressed as follows:

$$\varphi_d = \neg \bigvee_{i=1}^{|T|} \left(\bigwedge_{p_j \in {}^\bullet t_i} |\mu(p_j)| \geq W(p_j, t_i) \right)$$

This means that there is "at least one transition enabled." The condition φ_d is used within s1 in the hull module as a part of the enabling condition, i.e., the 'deadlock' condition affects the hull's member "en," as illustrated in Figure 9.

4. ETHERNET NETWORK COPROCESSOR

This section shows the applicability of our DFN modeling approach to the real-life Ethernet coprocessor. Empirical results have been obtained using the Cadence SMV tool [Cadence 2001] in a Sparc Sun-Ultra 10 / 440 MHz with 512 Mb RAM running Solaris 8.

The Ethernet network is an IEEE standard. The network coprocessor transmits and receives data frames over a network by means of the CSMA/CD protocol, which is defined in the IEEE 802.3 standard [ANSI/IEEE 1991]. There have been a number of attempts to perform a hardware/software codesign of the Ethernet coprocessor [Gupta and De Micheli 1992], as well as benchmarking [Narayan and Vahid 1992]. Furthermore, the formal verification of this coprocessor has been carried out using a predecessor of the tool used in our verification engine [Naik and Sistla 1994].

The operation of the coprocessor is controlled by the execution unit, which sends the starting memory address to the transmit unit and then enables a DMA unit to operate straight into memory. The DMA unit (dma_xmit) directly reads from the successive memory locations in order to obtain destination address, data length, and the actual data, which are then sent to the xmit_frame.

Figure 10 shows the top level diagram of the Ethernet coprocessor's DMA transmitter/receiver. Data flow has been marked with thick lines in order to

distinguish it from the control flow. This unit has two modes of operation: *dma_xmit_normal* and *dma_xmit_cancel*, so that it normally stays in the first mode but, if a failure occurs in the transmission to xmit_frame, the DMA unit switches to an alternative mode that sets the environment to restart the transmission process. It can be observed in the figure that only the first mode of operation deals with the data flow of the system.

The work in Narayan and Vahid [1992] describes the dma_xmit unit as a combination of eight mutual exclusive subprocesses: START, DEST1, DEST2, LENGTH, DATA, DATA2, END, and RESTART. Modeling this specification by the DFN model requires, to begin with, a set of eight places (p_1 to p_8) to indicate the beginning of each subprocess. These places have been highlighted in Figure 11 for further reference. For instance, the initialization of the START subprocess is captured by placing a token in p_1. The specification of dma_xmit also shows that once the START subprocess has been initialized, it will wait for an external signal (txstart), in order to request access from the CPU (Bhold). Such a behavior is conceptualized by the waiting of a token in p_{10} and, after firing a couple of transitions (t_1 and t_2), placing a token in p_{21}. This is more formally expressed in the φ_{acc} property.

Table I shows the list of properties that are necessary to prove, in order to assure the correctness of the Ethernet coprocessor model. We can identify two types of properties: those that describe the functionality of the system (φ) and those that assure that all subprocesses will eventually be used by such a functionality (ψ). As explained above, when a txstart signal is sent to the DMA unit, this will request access to the CPU (c.f. φ_{acc}). The system them reads from successive memory locations (c.f. φ_{s1}, φ_{s2} and φ_{s3}), starting from $\text{txaddress}_{[16]}$ (c.f. φ_{from}). At this point, the Ethernet coprocessor is ready to transmit $\text{Bdata}_{[16]}$ to xmit_frame, but this is an 8-bit unit. Properties φ_{high} and φ_{low} have been formulated in order to prove that both $\text{Bdata}_{[15..8]}$ and $\text{Bdata}_{[7..0]}$ are transferred to such unit. However, it is not sufficient to prove these nine functional properties. In order to complete the proof, *liveness* among the subprocesses has to be guaranteed. Therefore, by means of the ψ properties, we prove that each subprocess will eventually call another subprocess (i.e., the DFN model is *live*).

The complete correctness of the DFN model for the Ethernet coprocessor has been proved after 4809.2 sec (approx. 1 hour and 20 min) of verification time.

5. CONCLUSIONS

Control and data flow of embedded system specifications are usually kept separate on a structural point of view, for ease of its implementation phase, while its behavior is usually analyzed in a framework that combines both flows and makes them indistinguishable. Unlike most approaches in the realm of embedded-system modeling based on Petri nets, which extend the classical weighted, directed, bipartite graph, our DFN model is based on a *tripartite* graph. Thus, specifications consisting of both control and data flow are efficiently captured, keeping a tight link between control and data domains. Such a significant alteration to the classical Petri nets has shown, throughout the

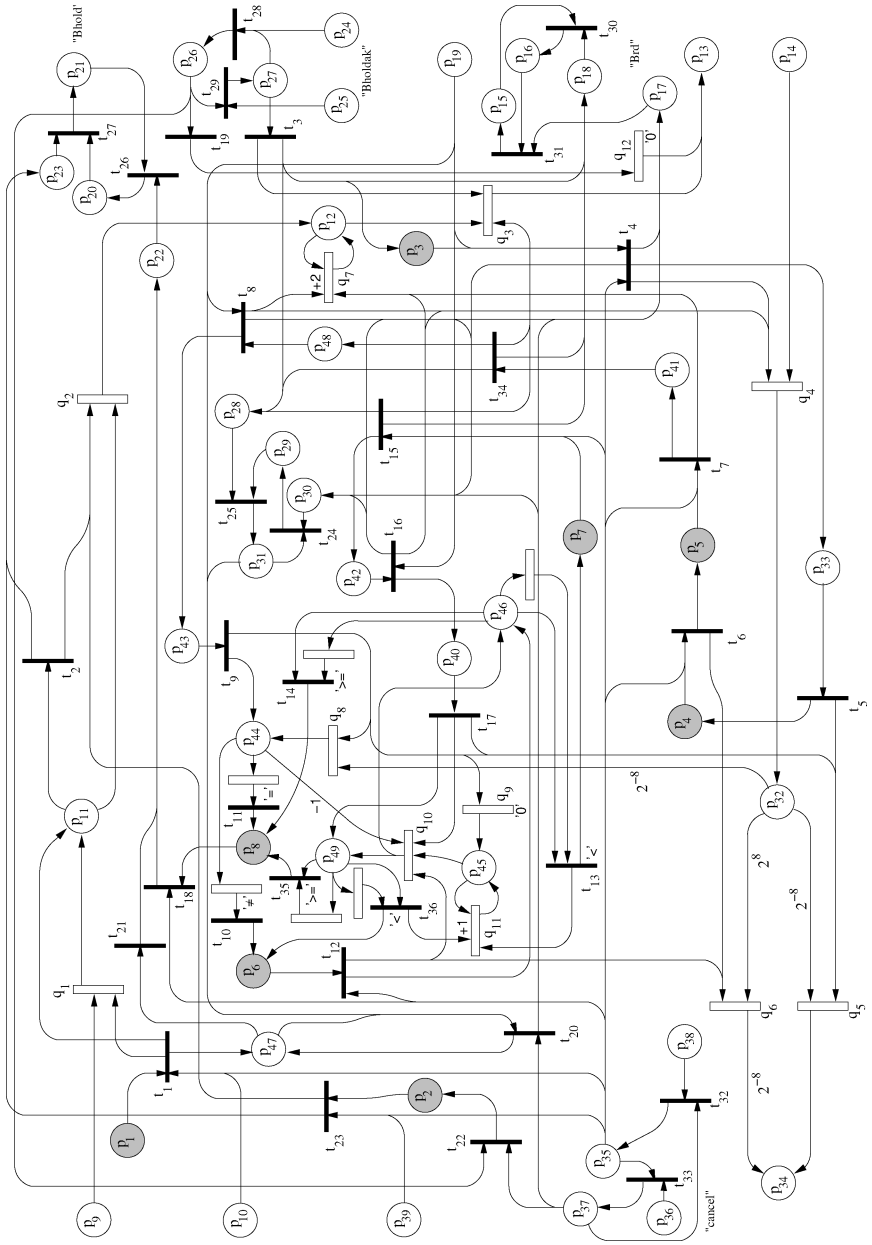


Fig. 11. DFN model of the Ethernet coprocessor.

paper, that issues concerning the linkage between control and data flow are of major concern, in order to accurately capture the behavior of embedded systems.

The formal verification of the DFN model has been also addressed in this paper, through a library that captures the semantics of the model. This library allows to mathematically prove the correctness of embedded system

Table I. LTL Properties of the Ethernet Coprocessor

Name	Property	BDD Nodes	Where?
φ acc	$\lvert\mu(p_{10})\rvert = 1 \Longrightarrow \Diamond\lvert\mu(p_{21})\rvert = 1$	10033	START (p_1)
φ from	$\lvert\mu(p_{10})\rvert = 1 \Longrightarrow \Diamond\angle\mu(p_{12}) = \angle\mu(p_9)$	10256	START (p_1)
φ s1	$\lvert\mu(p_{10})\rvert = 1 \Longrightarrow \Diamond\angle\mu(p_{13}) = \angle\mu(p_{12})$	10494	START (p_1)
φ s2	$\Diamond\angle\mu(p_{13}) = \angle\mu_0(p_{12}) + 2$	141160	LENGTH (p_5)
φ s3	$\Diamond\angle\mu(p_{13}) = \angle\mu_0(p_{12}) + 4$	140948	LENGTH (p_5)
φ high	$\Diamond\angle\mu(p_{34}) =$ high$(\angle\mu(p_{14}))$	5704	DEST1 (p_3)
φ low	$\Diamond\angle\mu(p_{34}) =$ low$(\angle\mu(p_{14}))$	3268	DEST2 (p_4)
φ rel	$\lvert\mu(p_8)\rvert = 1 \Longrightarrow \Diamond\lvert\mu(p_{26})\rvert = 1$	7015	END (p_8)
φ fail	$\lvert\mu(p_{36})\rvert = 1 \Longrightarrow \Diamond$sch $= 23$	9029	RESTART (p_2)
ψ start	$\lvert\mu(p_{10})\rvert = 1 \Longrightarrow \Diamond\lvert\mu(p_3)\rvert = 1$	10017	START (p_1)
ψ dest1	$\lvert\mu(p_3)\rvert = 1 \Longrightarrow \Diamond\lvert\mu(p_4)\rvert = 1$	4012	DEST1 (p_3)
ψ dest2	$\lvert\mu(p_4)\rvert = 1 \Longrightarrow \Diamond\lvert\mu(p_5)\rvert = 1$	2365	DEST2 (p_4)
ψ len-c	$\Diamond\lvert\mu(p_{43})\rvert = 1$	7628	LENGTH (p_5)
ψ len-d	$\Diamond\angle\mu(p_{32}) = \angle\mu_0(p_{14})$	69283	LENGTH (p_5)
ψ len-i	$\Diamond\angle\mu(p_{44}) =$ high$(\angle\mu(p_{32}))$	5506	LENGTH (p_5)
ψ len-n0	$\mu(p_{44}) = 1 \cdot e^{\mathbf{i}\cdot\alpha}, \alpha \neq 0 \Longrightarrow \Diamond\lvert\mu(p_6)\rvert = 1$	2629	LENGTH (p_5)
ψ len-e0	$\mu(p_{44}) = 1 \cdot e^{\mathbf{i}\cdot 0} \Longrightarrow \Diamond\lvert\mu(p_8)\rvert = 1$	3760	LENGTH (p_5)
ψ dt0	$\Diamond\lvert\mu(p_{46})\rvert = 1$	10013	DATA (p_6)

specifications represented in DFN, which does not mean that the DFN model acts as an intermediate step between another modeling paradigm and its verification. On the contrary, it is assumed that the designer develops the model based on the specifications of the design and produces a set of properties that will allow him to verify the correctness of that model. The DFN modeling and verification phase have been applied to a real-life example, i.e., the Ethernet coprocessor, in order to show the applicability of this methodology to complex designs.

ACKNOWLEDGMENTS

The authors wish to thank the anonymous reviewers for their constructive comments and suggestions, which improved the paper's presentation and clarity.

REFERENCES

ANSI/IEEE. 1991. *Information Processing Systems—Local Area Networks—Part 3: Carrier Sense Multiple Access with colision Detection (CSMA/CD) access method and physical Layer Specifications*. IEEE, New York.

BELLINI, P., MATTOLINI, R., AND NESI, P. 2000. Temporal Logics for Real-Time System Specification. *ACM Computing Surveys 32*, 1 (Mar.), 12–42.

BRAGE, J. P. 1993. Foundations of a high-level synthesis system. Ph.D. thesis, Department of Computer Science, Technical University of Denmark, Lyngby, Denmark.

BURCH, J. R., CLARKE, E. M., MCMILLAN, K. L., DILL, D. L., AND HWANG, L. J. 1992. Symbolic Model Checking: 10^{20} states and beyond. *Imperial College of Science, Technology and Medicine 98*, 2, 142–170.

CADENCE. 2001. The SMV Model Checker. http://www-cad.eecs.berkeley.edu/~kenmcmil/smv/.

CLARKE, E. M., GRUMBERG, O., AND PELED, D. A. 1999. *Model Checking*. MIT Press, Cambridge, MA.

CORTÉS, L. A., ELES, P., AND PENG, Z. 2000. Verification of Embedded Systems using a Petri Net based Representation. In *Proceedings of the* 13^{th}*International Symposium on System Level Synthesis (ISSS)*. Madrid, Spain. 149–155.

DIMIEV, S. AND MARKOV, K. 2002. Gauss Integers and Diophantine Figures. In *Proceedings of the* 31^{st} *Conference on Union of Bulgarian's mathematicians*. Borovetz, Bulgaria.

ELES, P., KUCHCINSKI, K., PENG, Z., DOBOLI, A., AND POP, P. 1998. Scheduling of conditional process graphs for the synthesis of embedded systems. In *Proceedings of the* 1^{st} *Design, Automation and Test in Europe (DATE)*. Paris, France.

EMERSON, E. A. 1990. Temporal and Modal Logic. In *Handbook of Theoretical Computer Science*, J. van Leeuwen, Ed. vol. B: Formal Models and Semantics. Elseiver Science, New York, Chapter 16, 995–1072.

EMERSON, E. A. AND HALPERN, J. Y. 1986. 'Sometimes' and 'Not Never' revisted: on Branching versus Linear Time Temporal Logic. *ACM 33*, 1, 151–178.

ESSER, R. 1996. An object oriented petri net approach to embedded system design. Ph.D. thesis, Computer Engineering and Networks Lab (TIK), Swiss Federal Institute of Technology (ETH) Zurich, Gloriastrasse 35, CH-8092 Zurich.

FINKEL, A. AND MEMMI, G. 1985. An Introduction to FIFO nets –Monogeneous nets: A subclass of FIFO nets. *Theoret. Computer Sci. 35*, 191–214.

GAJSKI, D. D., Ed. 1987. *Silicon Compilers*. Addison–Wesley, Reading, MA.

GAJSKI, D. D. 1997. *Principles of Digital Design*. Prentice-Hall, Englewood Cliff, NJ.

GENRICH, H. 1987. Predicate/Transition-Nets. In *Advances in Petri Nets 1986 Part I: Petri Nets, central models and their properties*, W. Brauer, W. Reisig, and G. Rozenberg, Eds. Lecture Notes in Computer Science, vol. 254. Springer-Verlag. 207–247.

GILL, A. 1962. *Introduction to the Theory of Finite-State Machines*. Electronic Sciences Series. McGraw-Hill, New York, 10020.

GRODE, J., MADSEN, J., AND JERRAYA, A.-A. 1998. Performance Estimation for Hardware/Software Codesign using Hierarchical Colored Petri Nets. In *International Symposium on High Performance Computing*.

GUPTA, R. K. AND DE MICHELI, G. 1992. System Synthesis via Hardware-Software Co-Design. Technical Report CSL-TR-92-548, Stanford University, Computer Systems Laboratory.

JENSEN, K. 1992. Coloured Petri Nets—Basic Concepts, Analysis Methods and Practical Use. *EATCS Monographs in Theoretical Computer Science 1*, Basic Concepts, 1–234.

JENSEN, K. 1994. Coloured Petri Nets—Basic Concepts, Analysis Methods and Practical Use. *EATCS Monographs in Theoretical Computer Science 2*, Analysis Methods.

JENSEN, K. 1997. Coloured Petri Nets—Basic Concepts, Analysis Methods and Practical Use. *EATCS Monographs in Theoretical Computer Science 3*, Practical Use.

KERN, C. AND GREENSTREET, M. R. 1999. Formal Verification in Hardware Design: A Survey. *ACM Transaction on Design Automation of Embedded Systems 4*, 2 (Apr.), 1–67.

KLEINJOHANN, B., TACKEN, J., AND TAHEDL, C. 1997. Towards a Complete Design Method for Embedded Systems Using Predicate/Transition-Nets. In *Proceedings of the* 13^{th} *IFIP WG 10.5 Conference on Computer Hardware Description Languages and Their Applications (CHDL)*. Toledo, Spain.

KUPFERMAN, O. AND VARDI, M. Y. 1995. On the Complexity of Branching Modular Model Checking. In 6^{th} *International Conference on Concurrency Theory (CONCUR)*. Philadelphia, Pennsylvania.

LEE, E. A. AND MESSERSCHMITT, D. G. 1987. Synchronous Dataflow. *Proceedings of the IEEE 75*, 9 (Sept.), 1235–1245.

MCMILLAN, K. L. 1993. *Symbolic Model Checking*. Kluwer Academic Pub, Boston, MA.

MERLIN, P. AND FABER, D. J. 1976. Recoverability of Communication Protocols. *IEEE Transaction on Communications 24*, 9, 1036–1043.

MURATA, T. 1989. Petri Nets: Properties, Analysis and Applications. *Proceedings of the IEEE 77*, 4 (Apr.), 541–580.

NAIK, V. G. AND SISTLA, A. P. 1994. Modeling and Verification of a Real Life Protocol Using Symbolic Model Checking. In *Proceedings of the* 6^{th} *International Conference on Computer Aided Verification*. Lecture Notes in Computer Science, vol. 818. Stanford, CA.

NARAYAN, S. AND VAHID, F. 1992. Modeling with SpecCharts. Technical Report ICS-TR-90-20, University of California, Irvine, Department of Information and Computer Science. Oct.

ÖSTERLING, A., BENNER, T., AND ERNST, R. 1997. Code Generation and Context Switching for Static Scheduling of Mixed Control and Data Oriented HW/SW Systems. In *Proceedings of the APCHDL'97*. Taiwan. 131–135.

PENG, Z. 1987. A formal methodology for automated synthesis of vlsi systems. Ph.D. thesis, Linköping University.

PENG, Z. AND KUCHCINSKI, K. 1994. Automated Transformation of Algorithms into Register-Transfer Level Implementations. *IEEE Transaction on Computer-Aided Design of Integrated Circuits and Systems 13*, 2 (Feb.), 150–166.

PNUELI, A. 1985. Linear and Branching Structures in the Semantics and Logics of Reactive Systems. In 12^{th}*International Colloquium on Automata, Languages, and Programming*. Springer-Verlag, Berlin, 15–32.

STAUNSTRUP, J. AND WOLF, W., Eds. 1997. *Hardware/Software Co-Design: Principles and Practice*. Kluwer Academic Publishers, Dordrecht, The Netherlands.

STREHL, K., THIELE, L., GRIES, M., ZIEGENBEIN, D., ERNST, R., AND TEICH, J. 2001. FunState—an internal design representation for codesign. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems 9*, 4 (Aug.), 524–544.

VARDI, M. Y. 2001. Branching vs. linear time: Final showdown. In *European Joint Conference on Theory and Practice of Software*. Genova, Italy.

WANG, F. 2004. Formal Verification of Timed Systems: A Survey and Perspective. *Proceedings of the IEEE 92*, 8 (Aug.), 1283–1305.

WOLF, W. H. 2003. A Decade of Hardware/Software Codesign. *IEEE Computer 36*, 4 (Apr.), 38–43.

ZIEGENBEIN, D., RICHTER, K., ERNST, R., THIELE, L., AND TEICH, J. 1999. SPI—An Internal Representation for Heterogeneously Specified Embedded Systems. In *Proceedings of the GI/ITG/GMM Workshop "Entwurfsmethoden & Beschreibungssprachen*." Braunschweig, Germany.

Received September 2003; revised March 2004 and November 2004; accepted April 2005