Symbolic Model Checking of Dual Transition Petri Nets

Mauricio Varea*, Bashir M. Al-Hashimi,
Department of Electronics and Computer Science
University of Southampton, SO17 1BJ, UK
{ m.varea , bmah }@ecs.soton.ac.uk

Luis A. Cortés, Petru Eles and Zebo Peng
Department of Computer and Information Science
Linköping University, S-581 83, Sweden
{ luico , petel , zpe }@ida.liu.se

ABSTRACT

This paper describes the formal verification of the recently introduced Dual Transition Petri Net (DTPN) models [12], using model checking techniques. The methodology presented addresses the symbolic model checking of embedded systems behavioural properties, expressed in either computation tree logics (CTL) or linear temporal logics (LTL). The embedded system specification is given in terms of DTPN models, where elements of the model are captured in a four-module library which implements the behaviour of the model. Key issues in the development of the methodology are the heterogeneity and the nondeterministic nature of the model. This is handled by introducing some modifications in both structure and behaviour of the model, thus reducing the points of nondeterminism. Several features of the methodology are discussed and two examples are given in order to show the validity of the model.

1. INTRODUCTION

Ensuring the correctness of embedded systems is becoming a key research area, since traditional methods of validation which involves simulation and testing are rapidly becoming infeasible. Formal verification, on the other hand, mathematically checks whether of not the functionality of an embedded system satisfies given properties. This form of validation is increasingly gaining popularity in hardware verification [6] and software development [5]. Formal verification techniques based on model checking have been widely used in the verification of finite-state concurrent systems. Model checking algorithms [1] decide whether or not a design satisfies some desired properties, which are expressed in a temporal logic such as computation tree logic (CTL) or linear temporal logic (LTL). If binary decision diagrams (BDD) are used, instead of an exhaustive search through the entire state space, this technique is known as *Symbolic Model Checking* [8].

Symbolic model checking has been applied to the verification of embedded systems internal design representation (IDR). An approach which reduces the reachability graph of a Petri net has been presented in [2]. This approach introduces a model checking algorithm which makes use of a potentially reachable state space, in order to reduce the size of the state space to be explored. Another approach based on functions driven by state machines (FunState) has been recently introduced in [10]. FunState formal verification strategy is based on the symbolic model checking of regular state machines (RSM) [11].

Petri net (PN) based models are a suitable internal design representation for hardware/software specifications of embedded systems, since they are capable of exploiting many desired features of the design, *e.g.* concurrency. PRES+ is a Petri net oriented model aimed to represent embedded systems, which has been applied to formal verification [3]. Verification of Timed CTL (TCTL) properties of PRES+ models is possible by means of their transformation into Timed Automata. A new IDR, which efficiently captures both control and data structure from a behavioural description of an embedded system, has been recently proposed [12]. This model is called *Dual Transition Petri Net* (DTPN), and one of its features is the combined representation of control and data flow.

This paper describes the formal verification, using symbolic model checking techniques, of the recently introduced DTPN models. The aim of this work is to exploit the features related to the coexistence of control and data flow in embedded systems specification. We capture the structure and behaviour of the embedded system by means of an IDR and further apply the Cadence SMV tool [14] in order to reason about its properties. This work is organised as follows. Section 2 introduces the framework of this paper, as well as proposes a methodology for symbolic model checking which takes into account the heterogeneous nature of embedded systems. Finally, two embedded systems of different complexity are analysed in Section 3, while some concluding remarks are given in Section 4.

2. VERIFICATION OF DUAL TRANSITION PETRI NETS

Dual Transition Petri Net (DTPN) is an extention of classical PNs which directly supports the separation of control and data flow that typically occurs in practical embedded systems design [12]. DTPN utilise a unified approach to model both control and data flow of an embedded system specification, using a complex number notation on the behavioural analysis of the system. A DTPN model is composed of: a set of places (P), two sets of transitions (T, Q), two sets of arcs (F_C, F_D), a weight function ($W = W_C \cup W_D$) and a guard function ($G = \bigcap_{j=1}^{m} G_j, m = |T|$). Two domains are identifiable: control domain, *i.e.* $\{P, T, F_C, W_C\}$, and data domain, *i.e.* $\{P, Q, F_D, W_D\}$. There is a link between both domains achieved by the guard function $G_j : {}^\circ t_j \rightarrow \{0,1\}$. Here, the *control* transition t_j fires according to the *data* contained in $p_i \in {}^\circ t_j$, $1 \leq i \leq n = |P|$.

*The first author performed part of this work as a research visitor at Linköping University.

A key issue in the behavioural DTPN model is the marking of its places $\mu(p) \in \mathbb{C}$, $\forall p \in P$, which allows a dual functionality of the system. As a consequence, PN models with $\mu(p) \in \mathbb{N}$ can be expressed as a subset of DTPN models which have $\mu(p) \in \mathbb{C}$.

Given $x \in \{P, T, Q\}$, two sets of presets ($^\bullet x$ for control and $^\circ x$ for data domain) and two sets of postsets ($x^\bullet$ for control and x° for data domain) have been defined. The behaviour of control transitions $t \in T$ are prompted by the classical enabling and firing rules, while data transitions $q \in Q$ have slightly different rules, which are formulated in (1) and (2) respectively.

$$\left[\exists p_i \in {}^\bullet q \,\middle|\, \angle\mu(p_i) \geqslant W_C(p_i, q)\right] \qquad (1)$$

$$|\mu_{k+1}(p_j)| = \sum_i |\mu_k(p_i)| \cdot W_D(p_i, q), \quad \forall p_i \in {}^\circ q, p_j \in q^\circ \qquad (2)$$

Further definition of DTPN modelling technique, as well as its motivation and introductory examples, can be found in [12].

Model checking algorithms check for completeness and uniqueness of a solution. Therefore, the amount of nondeterminism present in the transition relation of the underlying Kripke structure has to be controlled. In order to combine the DTPN modelling technique with symbolic model checking algorithms, it is important to perform a reduction in the points of nondeterminism of the model. To achieve this, Section 2.1 incorporates some further restrictions to the current definitions of DTPN.

2.1 Modified Dual Transition Petri Net

The original DTPN model [12] has been defined to support nondeterminism in both control and data domains, thus allowing a compact graphical representation. In order to reduce the level of nondeterminism in a DTPN model, we propose the use of an additional set of arcs in the structural model and a further rule in the behavioural model. There are two sets of arcs in DTPN: control flow arcs (F_C) and data flow arcs (F_D). The introduction of a third set of arcs, namely F_A, provides a further link between control and data flow. This set of arcs is defined by:

$$F_A \subseteq (T \times Q)$$

The link introduced reduces the amount of nondeterminism, since the new set of arcs F_A makes the execution of data transitions dependent on the firing of control transitions, as stated below in Definition 1. Therefore, only control transitions are allowed to fire nondeterministically (*i.e.* we transform the data domain into a deterministic set). As a consequence, the execution of the net is not only ruled by the control and data transitions firing rule formerly presented [12], but also the link between them.

DEFINITION 1. *Firing a control transition $t \in T$ produces the execution of* all *enabled data transitions $q \in Q$ which are linked to t by the set F_A, i.e. $\{q \in Q \,|\, (t,q) \in F_A\}$.*

Note that the new enabling condition $(t,q) \in F_A$ of a data transition $q \in Q$ depends on other active elements (*i.e.* control transitions $t \in T$) unlike [12], which uses places $p \in {}^\bullet q$ for this purpose. Therefore equation (1) has to be modified, leading to (3).

$$\left[\exists (t_j, q) \in F_A \,\middle|\, \angle\mu(p_i) \geqslant W_C(p_i, t_j), \forall p_i \in {}^\bullet t_j\right] \qquad (3)$$

To illustrate this modification, Figure 1 shows a simple DTPN model which will be explained in Section 3.1. The presence of the new set of arcs F_A can be observed, since data transitions $q_i \in Q$ are linked to a control transition $t_j \in T$, *e.g.* by arcs (t_1, q_2) and (t_2, q_1), as shown by dashed lines in Figure 1. From the behavioural point of view, the execution of each data transition q is synchronised with some control transition t in the net. The consequence of such a behaviour is that on the one hand no data transition q can fire in a nondeterministic way, but on the other hand the efficiency of dealing with the modelling complexity is slightly reduced.

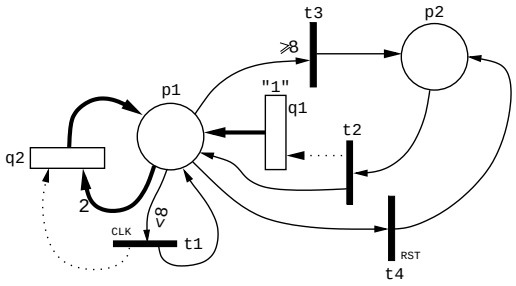

Figure 1: A modified DTPN model

2.2 Symbolic Model Checking Methodology

The proposed (symbolic) model checking methodology based on DTPN models is shown in Figure 2. The inputs to the methodology are the DTPN model of the embedded system and a set of properties expressed in temporal logics. A verification engine is used to generate the BDD space, using a translation of the DTPN model into a Kripke structure (N) and the given LTL or CTL properties to be verified (f). The outcome of the Model Checker, *i.e.* either YES, NO or $(unknown)$, provides information on the correctness of the DTPN model with regard to the LTL/CTL properties.

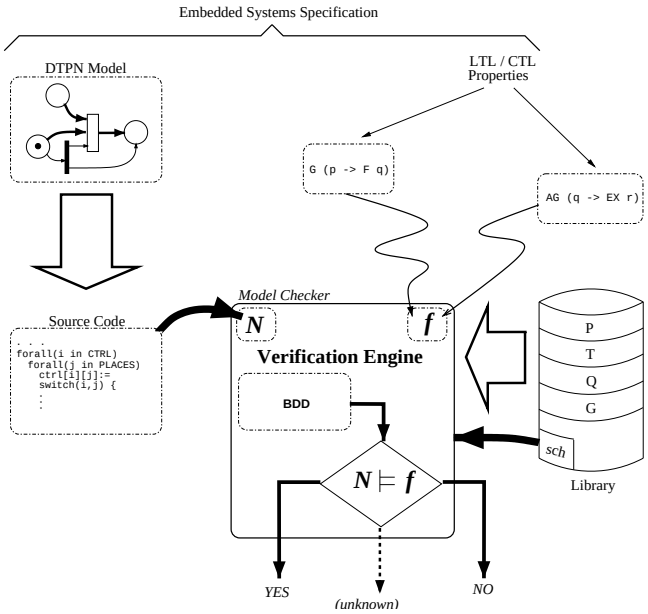

Figure 2: Our Model Checking Methodology

One form of heterogeneity present in embedded systems is due to the existence of two separate but related parts, *i.e.* control and data flow. These two parts are tightly linked in a DTPN model. In order to perform the symbolic model checking of a DTPN model, we have implemented a library which supports the underlying heterogeneity implicit in the model. This library consist of four modules which are instantiated for creating the structure of the net,

i.e. place(), *control_transition()*, *data_transition()* and *guard()*, and a scheduler `sch`. The communication among these modules is illustrated in Figure 3, where the effects of the new Definition 1 are highlighted. Here, it can be observed that the *control_transition()* module interacts with only half of the array structure, producing a result that uses F_A to select which of the elements of the array of places[1] *place()* $\equiv$ `p[i]` $\equiv p_i$ interacts with the *data_transition()* module. Figure 3 also shows that by means of a scheduler `sch`, which is global from the control transitions point of view but locally defined as to the data transitions, the two modules are perfectly synchronised. Since an enabled control transition may or may not fire, the scheduler has been defined in a nondeterministic way such that the schedule generated for each step is restricted to the set of control transitions which are enabled, *i.e.* the scheduler chooses nondeterministically (from the set of enabled control transitions) the next transition to fire. Because only *enabled* control transitions are used in the nondeterministic points, the generation of BDD nodes is optimised.

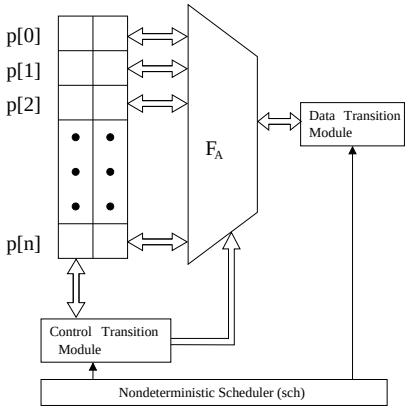

Figure 3: Communication among library modules

2.2.1 Places

Places in a DTPN are the only elements in the net with storage capabilities. As a consequence, in classical PN the set P is represented by an array of integer elements. However, a DTPN marking of a place $\mu(p_i)$ is composed of two disjointed parts, *i.e.* modulus and phase [12]. Therefore, each element in the array (`p[i]`) is a structure composed of two members, one for the number of tokens (control domain) and another for the value (data domain), as shown in Figure 3. The extraction of each part, *i.e.* the application of the $|\mu(p)|$ and $\angle\mu(p)$ operators in order to obtain modulus and phase respectively, is performed through a direct access to the corresponding member of the structure.

2.2.2 Transitions

Both control and data transitions have the same mechanism which involves (s1) checking whether they are enabled or not, and (s2) executing an action if s1 holds. The evolution of the state on a DTPN model depends on which control transitions fire, as they modify the marking of the net. Derived from PNs, the enabling condition for such transitions can be generalised into the following expression, where $1 \leqslant i \leqslant |T|$:

$$\psi_e(i) = \bigwedge_{\{j|p_j \in {}^\bullet t_i\}} \left(\angle\mu(p_j) \geqslant W_C(p_j,t_i)\right)$$

[1] the *place()* module and array `p[i]` are terms that we use interchangeably, since this *module* is only used to reserve some space in memory (*array*) for the allocation of control and data information.

Note that $\psi_e(i)$ is expressed in modal logics [4], which is not sufficient to express behavioural properties along the time.

Module: ***CONTROL_TRANSITION*** (T)
Input: - p: array of places
- pre: array in control domain
- post: array in control domain
- guard: boolean

Output: - p': array of places

s1: **Check enabling condition:**
en $\leftarrow$ guard $\wedge \bigwedge_{i=1}^{|P|}$ pre[i] $> 0 \implies \angle$ p[i] $\geqslant$ pre[i]

s2: **Perform Firing Operation:**
forall(p[i] $\in$ P)
if (en) then $\angle$ p'[i] $\leftarrow \angle$ p[i] - pre[i] + post [i]
else p'[i] $\leftarrow \angle$ p[i]

Figure 4: Algorithm for a DTPN Control Transition

The algorithm presented in Figure 4 consists of two main parts, checking for an enabling condition (s1) and the firing operation (s2). When a control transition is selected by the scheduler, an assignment occurs in all places of the net. Then, according to the structure formed by the instantiation of the *control_transition()* modules, only places affected will change the number of tokens.

Example 1 For the sake of clarity, consider a *control_transition()* module instantiated as follows:

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

Where:

- the first argument `p` is transparent to the user.
- from the second argument, `[3,1,0,2]`, the following information can be obtained: ${}^\bullet t_1 = \{p_1, p_2, p_4\}$ and $W_C(p_1,t_1) = 3$, $W_C(p_2,t_1) = 2$, $W_C(p_4,t_1) = 1$.
- analogously, the third argument `[0,1,2,0]` means that $t_1{}^\bullet = \{p_2, p_3\}$ and that $W_C(t_1,p_2) = 1$, $W_C(t_1,p_3) = 2$.
- finally, t_1 is not guarded by any guard function (since `1` is the default value on the result of a *guard()* module).

According to these considerations, Figure 5 illustrates the graphical representation of the control transition t_1.

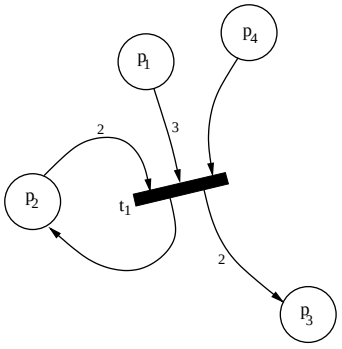

Figure 5: Graphical representation of the *control_transition()* module instantiation, for the example given

The enabling condition part (s1) of the *control_transition()* module introduced in Example 1 is:

$$\text{guard} \wedge \bigwedge_{i=1}^{|P|} \text{pre[i]} > 0 \implies \angle\text{p[i]} \geqslant \text{pre[i]}$$

Which can be unfolded into:

$$\top \wedge \angle \mathrm{p}[1] \geqslant \mathrm{pre}[1] \wedge \angle \mathrm{p}[2] \geqslant \mathrm{pre}[2] \wedge \angle \mathrm{p}[4] \geqslant \mathrm{pre}[4] = \top \wedge \angle \mathrm{p}[1] \geqslant 3 \wedge \angle \mathrm{p}[2] \geqslant 2 \wedge \angle \mathrm{p}[4] \geqslant 1 \quad (4)$$

The boolean result of (4) is used in part s2 to see whether the next step of the phase of each marked place is assigned to the result of the firing rule or its former content.

Similarly, Figure 6 shows the algorithm for a data transition $q \in Q$. The enabling part s1 uses a condition which is twofold: Firstly, it checks whether or not the control transition which fires at a given step k, *i.e.* `t[sch]`, has any connection to the data transition itself. This is due to Definition 1 —see (3). Secondly, it confirms that there is no *deadlock* in the control domain. Furthermore, the firing part s2 implements equation (2).

Module: ***DATA_TRANSITION*** (Q)

Input: - p: array of places
- pre: array in control domain
- post: array in control domain
- FA: set of arcs in F_A

Output: - p': array of places

s1: **Check enabling condition:**
if (output of the scheduler ∈ FA)&(there is no deadlock)
then
en ← ⊤
else en ← ⊥

s2: **Perform Firing Operation:**
forall(p[i] ∈ P)
$s \leftarrow \sum_{i=1}^{|P|} | p[i] | \cdot pre[i]$
forall(p[i] ∈ P)
if (en) then | p'[i] | ← post[i] · s
else p'[i] ← | p[i] |

Figure 6: Algorithm for a DTPN Data Transition

The deadlock condition is another modal logics formula, which can be expressed as:

$$\psi_d = \neg \bigvee_{i=1}^{|T|} \psi_e(i)$$

This means that "at least one control transition is enabled". The condition ψ_d is used to reason about the availability of a module in the execution path. Thereby, part s1 of Figure 6 shows that the deadlock condition, which has been globally defined, affects the en parameter.

2.2.3 Guard Function

The guard function G, defined in [12], maps information from the data domain into the control domain by comparing the *value* in a place with the label val of the guard function. This boolean evaluation is taken into account by a control transition $t \in T$.

2.3 Analysis of Properties in DTPN Models

The analysis of behavioural properties is of much interest in PN theory (thus, in DTPN). Since the evolution of the state of an embedded system is a time function, assuring that a certain temporal logics formula holds throughout the entire evolution leads to gain knowledge of the system's behaviour. In this section we analyse three behavioural properties, *i.e.* TL formula, namely *reachability*, *safety* and *liveness*, and present the CTL/LTL formulae which describes them.

To investigate about the dynamics of a system modelled in terms of a PN extension, it is necessary to perform the *reachability analysis* of the system's model. A marking μ_k is said to be reachable from a marking μ_0 if there is a sequence of firings which can turn μ_0 into μ_k. Using CTL formulae, it is possible to express this property as follows:

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

Where $c_i = b_i \, \mathbf{e}^{\mathbf{i} \cdot a_i}$ 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 general state of the system *eventually* reaches a marking of $\mu_k(p_i), \forall 1 \leqslant i \leqslant n$.

Safety properties are conditions that are verified along any execution path. These type of properties are usually associated with some critical behaviour, thereby they should *always* hold. Classically, a *safe* PN only allows a boolean marking function, which means that the following LTL formula holds:

$$\varphi_S = \Box \bigwedge_{i=1}^{|P|} (\angle \mu(p_i) \leqslant 1)$$

Analogously, liveness properties are useful to express that "interesting things eventually happen". Like in classical PNs, a control transition t_i is said to be *live* if it can eventually fire, which implies that it is eventually enabled $\Diamond \psi_e(i)$, $\forall 1 \leqslant i \leqslant m$, where $m = |T|$. Thus, a live DTPN model satisfies the following LTL property:

$$\varphi_L = \bigwedge_{i=1}^{|T|} \Diamond \psi_e(i)$$

3. EXPERIMENTAL RESULTS

The proposed methodology has been applied to a number of examples in order to demonstrate its validity. Without loosing generality, the implementation of the algorithms has been done using the Cadence SMV tool [14] as verification engine for the methodology, for the following reasons:

- It is robust and well known within the community.
- It is able to analyse both CTL and LTL properties.
- It can potentially reduce the BDD space by means of symmetry.
- It supports data type reduction.

3.1 Verification of State Machines

In this section we present a very basic FSM which will aid to the understanding of the DTPN model itself and the property encoding process. The FSM analysed has a cyclic behaviour, unless a reset (RST) signal holds [13], and its number of states is directly proportional to the number of places in the net. The simple sequence for this four-bit state machine is: `0001` → `0010` → `0100` → `1000`. When RST holds, the state register is unconditionally assigned a value of `0001`.

The DTPN model for such a behaviour has been introduced in Section 2.1. With reference to Figure 1, there are two input signals: clock (CLK) and reset (RST). The first signal is bound into transition t_1 while the latter into transition t_4. This means that the firing

of t_1 represents the rising edge of CLK (*i.e.* when the signal CLK is active) and the firing of t_4 takes place when RST holds.

Once the DTPN model is encoded in the way described in Section 2.2, and used as input to the verification engine, some properties of the embedded system can be analysed. Since there are two signals present in this FSM, *i.e.* CLK and RST, it is important to check how the model responds to each of them, independently. First, we propose four LTL formulas to check if the sequential behaviour produced by the CLK signal is the desired one.

$$\varphi_1 = \Box(|\mu(p_1)| = 1 \implies \bigcirc|\mu(p_1)| = 2)$$
$$\varphi_2 = \Box(|\mu(p_1)| = 2 \implies \bigcirc|\mu(p_1)| = 4)$$
$$\varphi_3 = \Box(|\mu(p_1)| = 4 \implies \bigcirc|\mu(p_1)| = 8)$$
$$\varphi_4 = \Box(|\mu(p_1)| = 8 \implies \Diamond|\mu(p_1)| = 1)$$

Properties φ_1, φ_2, φ_3 and φ_4 show the natural evolution of the state for this FSM assuming no *reset* action, *i.e.* t_4 never fires, which is itself another property to verify:

$$\varphi_e = \Box(\texttt{sch}^* \neq t_4)$$

Property φ_e is an *assumption*, which means that it is assumed to be true while verifying properties $\varphi_i, \forall 1 \leq i \leq 4$. This assumption can be read as *the scheduler never points to* t_4, where $\texttt{sch}^*$ is the value pointed by sch. Since there exist an interdependability among φ_i, it can be concluded that if all four properties hold (we already know that the BDD space generated will only consider the cases in which φ_e holds), the embedded system will remain in the cyclic behaviour described at the beginning of this section.

To verify that firing t_4 will reset the system, regardless of the previous state, the following LTL property should be assessed:

$$\varphi_r = \Box(\angle\mu(p_2) = 1 \implies \bigcirc|\mu(p_1)| = 1)$$

This means that if, at any time p_2 is marked with a token, then the next state has $|\mu(p_1)| = 1$ unavoidably.

The four-state example presented in this section has allocated 1105 BDD nodes in memory. If the same example is encoded as a classical PN, 4353 BDD nodes are necessary in order to verify equivalent properties. Moreover, it is not difficult to extend these results to a larger number of states in the FSM. Since the SMV tool used here is potentially designed to cope with symmetry, having to formulate more φ_i properties in order to verify the correctness of the FSM for the CLK signal will not necessarily increase the BDD space proportionally.

3.2 Verification of a VME-bus controller

An interesting asynchronous circuit which has been widely studied is the VME-bus controller [9]. This bus is of much interest in industrial applications (*e.g.* avionics), since it provides a flexible multi-master bus arbitration scheme which is both simple and not vulnerable to noise. The VME design technology has evolved since it became an ANSI approved standard. In [7], the controller was modelled by means of Signal Transition Graphs (STG), a Petri net oriented formalisation of timing diagrams for asynchronous design, and then synthesised into a gate netlist. Since STGs are a form of PNs and the DTPN model is a generalisation of PNs (*c.f.* Section 2), we can apply the proposed methodology to this controller.

Figure 7 shows the interface of a generic device connected to a VME bus. The functionality of the controller is to regulate the reading and writing cycles of the device connected to the bus through

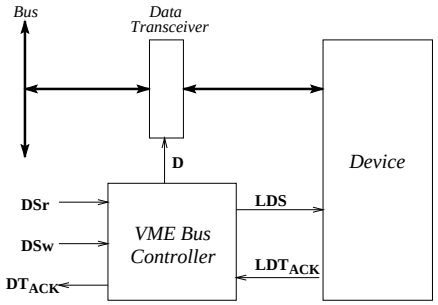

Figure 7: VME bus controller

a data transceiver. The controller opens the transceiver to transfer data from/to the device/bus, through the signal D In the read cycle, a request-to-read signal DSr is propagated to LDS. Then, the device acknowledges with a $\mathsf{LDT_{ACK}}$ when the data is ready, which is taken into account to activate signal D. A falling edge in signal DSr, *i.e.* the end of the read cycle, causes D and all other outputs of the controller to go low. In the write cycle (started by signal DSw), the controller places the data in the device's port and sends a request-to-read signal to the device through LDS signal. When the device is ready a $\mathsf{LDT_{ACK}}$ signal is produced, which closes the transceiver in order to isolate the device from the bus. Further information about the VME-bus signals can be found in [7].

Our goal is to verify if the synthesis results shown in [7] correspond to the behaviour inherent to the net description. To achieve this, we have encoded each signal of the original specification as an abstract type signal, which indicates that these signals are only used as part of the proof but do not belong to the actual system implementation.

Four next-states functions are derived from [7], which are:

$$D = LDT_{ACK} \cdot csc_0 \quad (5)$$
$$LDS = D + csc_0 \quad (6)$$
$$DT_{ACK} = D \quad (7)$$
$$csc_0 = DSr \cdot (csc_0 + \overline{LDT_{ACK}}) \quad (8)$$

We know that a next-state function in the synthesis of FSM represents the behaviour of the system described in terms of its previous state. This is analogous to the $\bigcirc$ operator in temporal logics. Therefore, we use the notation $\implies \bigcirc$ to express a "next-state" function, *i.e.* if the cause of the implication holds then the consequence holds in the *next* time step. Therefore, we propose three LTL formulas (φ_1, φ_2 and φ_3) to request the *infinitely often* completion of (5), (6) and (7).

$$\varphi_1 = \Box\Diamond\left(LDT_{ACK} \wedge csc_0 \implies \bigcirc D\right)$$
$$\varphi_2 = \Box\Diamond\left(D \implies \bigcirc DT_{ACK}\right)$$
$$\varphi_3 = \Box\Diamond\left(D \vee csc_0 \implies \bigcirc LDS\right)$$

Since (8) is recursive, it cannot be expressed as an LTL formula in SMV. However, SMV supports data type reduction and, thus, we can still cope with this limitation by means of an abstract type auxiliary signal (csc0). This is:

```
abstract csc0 : boolean;
next(csc0):= DSr & ( csc0 | ~LDTACK );
```

Altogether, the next-state functions derived in [7] are correct if these three properties hold when using the definition of the abstract signal for the fourth signal, hence the synthesis process is verified.

For the $n = 11$ places and the $m = 10$ control transitions that form part of the DTPN model used for the verification of the VME-bus controller, the methodology has allocated 10957 BDD nodes in memory (which represents an increase of only 2.52 times w.r.t. the $n = 2, m = 4$ example given in Section 3.1). Thus, the validation of DTPN through this methodology is more suitable for control dominated applications than for data dominated systems.

3.3 Scalability Analysis

This section analyses the results obtained when the method is scaled up to realistic problems. Sections 3.1 and 3.2 have shown the applicability of the proposed methodology to rather simple control dominated examples. However, the methodology can be applied to the formal verification of more complex problems, mainly in the area of reactive embedded systems. In order to give an insight of the methodology's applicability in dealing with such complexity, this section investigates the effects of variations performed on two early indicators of the complexity in control and data domains. These are: the capacity in the control domain (K_C) and the capacity in the data domain (K_D).

As in classical PN, the notion of capacity of a place in the control domain (K_C) is associated to the maximum number of tokens allowed. Similarly, K_D defines the capacity in the data domain as the maximum value allowed in a place. Variations on the parameter K_D of the underlying DTPN model are reflected in variations on the overall bus width, since there is a direct relation between the maximum *value* allowed in places and the size of the physical registers in the final implementation.

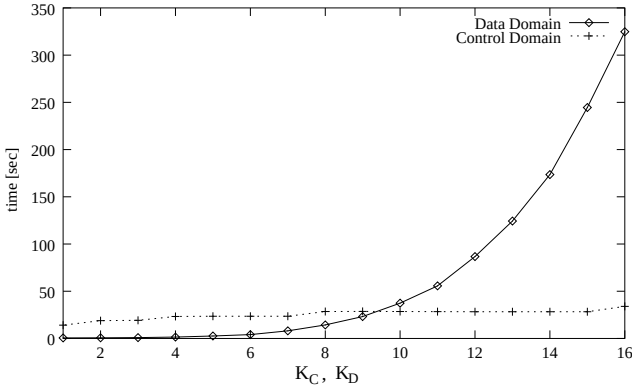

Figure 8: Verification time vs. capacities K_C, K_D

To illustrate the scalability of the proposed methodology, we have chosen the multiplier example presented in [12], since it contains a well balanced mixture of control and data flow. Figure 8 shows a search through the state space of this DTPN model using our proposed methodology. The solid line shows the state space exploration of the multiplier when $K_C = 2$ and K_D is variable, while the dotted line represents a $K_D = 8$ and a variable K_C. As it can be observed, increasing K_D results in an exponential growth of verification time, while increasing K_C results in a linear growth. Therefore, it can be said that larger ranges of the *value*s have a stronger impact on the complexity of the state space search, rather than increased number of tokens.

4. CONCLUSIONS

This paper has proposed a methodology for formal verification of the recently introduced dual transition Petri net (DTPN) model, using symbolic model checking techniques. This has involved the development of a four-module library which underpins the key features of DTPN, including heterogeneity. A new set of arcs and a rule for firing both types of transitions has been introduced in order to cut down the amount of nondeterminism. Both CTL and LTL formulas have been treated and a generalised analysis of properties has been introduced. The proposed methodology has been successfully applied to a number of examples, confirming its validity. Furthermore, this paper concludes with a preliminary indication on the scalability of the methodology, showing more affinity for control dominated than data dominated applications.

5. ACKNOWLEDGMENTS

The authors wish to thank Gethin Norman (University of Birmingham, UK), for many fruitful discussions related to this investigation.

6. REFERENCES

[1] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 1999.

[2] J. Cortadella. Combining structural and symbolic methods for the verification of concurrent systems. In *Int. Conf. on Application of Concurrency to System Design*, pages 2–7, Mar. 1998.

[3] L. A. Cortés, P. Eles, and Z. Peng. Verification of Embedded Systems using a Petri Net based Representation. In *Proceedings of the* 13^{th}*International Symposium on System Level Synthesis (ISSS)*, pages 149–155, Madrid, Spain, 20-22 Sept. 2000.

[4] E. A. Emerson. Temporal and Modal Logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, chapter 16, pages 995–1072. Elseiver Science Publishers, B.V., 1990.

[5] J. D. Gannon, J. M. Purtillo, and M. V. Zelkowitz. *Software Specification: A Comparison of Formal Methods*. Ablex, Norwood, NJ, 1994.

[6] C. Kern and M. R. Greenstreet. Formal Verification in Hardware Design: A Survey. *ACM Transaction on Design Automation of Embedded Systems*, 4(2):1–67, Apr. 1999.

[7] M. Kishinevsky, J. Cortadella, and A. Kondratyev. Embedded tutorial: Asynchronous interface specification, analysis and synthesis. In *Proceedings of the* 35^{th}*Design Automation Conference (DAC)*, pages 2–7, June 15–18 1998.

[8] K. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[9] W. D. Peterson. *The VMEbus Handbook*. VITA publications, Scottsdale, AZ 85260-3415, USA, 4^{th} edition, 1998.

[10] K. Strehl, L. Thiele, M. Gries, D. Ziegenbein, R. Ernst, and J. Teich. *FunState*—an internal design representation for codesign. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 9(4):524–544, Aug. 2001.

[11] L. Thiele, J. Teich, and K. Strehl. Regular state machines. *Journal of Parallel Algorithms and Applications*, 15:265–300, 2000. Special Issue on Advanced Regular Array Design.

[12] M. Varea and B. Al-Hashimi. Dual Transitions Petri Net based Modelling Technique for Embedded Systems Specification. In *Proceedings of the* 4^{th}*Proc. Design, Automation and Test in Europe (DATE)*, pages 566–71, Munich, Germany, Mar. 2001. IEEE/ACM.

[13] J. F. Wakerly. *Digital Design: Principles & Practices*. Prentice-Hall, Englewood Cliffs, New Jersey, 2^{nd} edition, 1994.

[14] The SMV Model Checker. http://www-cad.eecs.berkeley.edu/∼kenmcmil/smv/.