UDC 004.4
DOI: 10.20535/SRIT.2308-8893.2020.3.01

METHOD OF SEMANTIC APPLICATION VERIFICATION IN GPGPU TECHNOLOGY

S.L. KRYVYI, S.D. POGORILYY, M.S. SLYNKO, A.A. KRAMOV

Abstract. An application development and verification method for massively parallel systems using NVIDIA GPUs is proposed. The method allows creating models at different levels of abstraction using the apparatus of marked transition systems. The compositions (product) of such systems are transformed into a Petri net, which are then analyzed by appropriate means. The proposed method allows specifying model properties by temporal logic formulas. This allows studying the properties of massively parallel systems which is almost impossible to analyze manually, since the number of execution threads in the latest NVIDIA video adapter architectures (Pascal, Volta, Turing, Ampere) is measured in hundreds of thousands or millions.

Keywords: CUDA, graphical processing units (GPU), General Purpose Graphics Computing (GPGPU), transition system, Petri net, model design.

INTRODUCTION

The main feature of the latest graphical processing units (GPU) is the availability of a set of streaming multiprocessors (SM) that were used previously in image processing algorithms and tasks only. General Purpose Graphics Computing (GPGPU) technology is based on the use of a combination of GPUs working in parallel to process data using general-purpose algorithms (scientific or other, but not necessarily related to image processing). The latest graphics architectures from NVIDIA include Pascal, Volta, Amper, Thuring [1]. For this day, Volta is one of the most powerful GPU architectures, which is an indicator of achievements in the field of high-performance artificial intelligence calculations (the GTX TITAN Z graphics adapter, built on the top of two powerful GK110 cores, can provide peak performance up to 8 teraflops, and each core can implement 2880 stream processors, which in total gives 5760 stream processors).

NVIDIA, in addition, develops a series of video adapters focused on scientific applications and use in high-performance (cluster) computing. These GPUs lack some graphic-specific features and are widely used in the scientific field. This led to a significant increase in the number of supercomputers included in the TOP500 world most efficient computers [2] that utilize NVIDIA video adapters.

Nowadays high-performance computing (HPC) trends get shifted from using clusters consisting of general-purpose modules to more specialized accelerated components (in other words, from universal CPUs to other units — GPU, FPGA, etc.), that is, to less functional and less power consuming modules. Accelerators, unlike universal CPUs, can not run an operating system, and rely on external systems for I/O operations or task scheduling. Their advantage in productivity lies solely in the fact that these elements are used in large groups simultaneously. The article proposes methods of modelling the properties of GPU accelerator architectures at different abstraction levels.

In terms of programming, success of the most common GPGPU technologies (CUDA in particular) lies in the fact that they encapsulate the SIMD nature of GPU hardware. In most cases, the developer deals with individual streams that work with scalar data instead of warps [1] working with vectors.

Development of the complicated, massively parallel systems that utilize video adapters requires new scientific methods to justify both the architecture of the system and applications for them. Since modern multi-threaded applications have hundreds of thousands and millions of threads, solving parallelization tasks for such systems makes impossible to use traditional engineering design approaches and requires the use of a mathematical apparatus and formalization methods to substantiate the decisions made. Model justification is an effective way of algorithmic study of the parallel algorithms properties. One of the options for implementing such a justification is the use of the apparatus of algorithmic algebras [3], which allow formulating schemes of algorithms in the form of algebraic expressions that depend on various parameters, including software and hardware platforms, paradigms of parallel programming, etc. This paper focuses on using the transition systems (TS) [4] and their compositions as the main mathematical model, which allow creating models at different abstraction levels. Their properties can be investigated by translating in Petri networks (PN) [5] and can be specified by temporal logic formulas [6].

MODERN NVIDIA GPU ARCHITECTURES

CUDA (Computational Unified Device Architecture) is a parallel computing architecture developed by NVIDIA to facilitate the GPGPU programming by using high-level APIs. Since the pilot CUDA platform had been introduced more than 10 years ago, each new generation of NVIDIA GPUs provided better application performance (for example, in floating point operations), increased energy efficiency, added important new computing capabilities, and streamlined graphics processor programming. Today, NVIDIA GPUs are leading computing devices that have, in some way, defined the artificial intelligence revolution (AI). Nowadays GPGPU technology accelerates deep learning; high-precision text, voice, and other media data recognition systems; are used in the areas of molecular modeming, modelling of medical products, medical diagnosis, financial modelling, and others.

Application instructions that are executed in a GPU-based heterogeneous environment can be logically decomposed into the following parts:

- “host” instructions — blocks which are executed on the CPU;
- “kernel” functions — instruction blocks which are executed on the GPUs.

Host blocks define the context of the kernel functions execution, transferring data between the computer’s RAM and GPU memory.

All NVIDIA GPU architectures execute instructions in groups of 32 thread (known as warps) using the SIMT model (Single Instruction, Multiple Thread), that is, one instruction is executed by many threads simultaneously, although behaviour of each individual thread is not limited by anything. However, architectures prior to Pascal include, among others, a software counter and a mask common to all threads of the warp that determines which threads are active at any given moment. This means that in the case of a execution ow branching, each execution path uses only a subset of all threads, while the rest are deactivated. Once execution paths are converged, threads of a warp start being executed simultaneously again.

Such an implementation model gets rid of the necessity to track each individual thread state separately. However, tracing only a warp in general means reducing the level of parallelism if branching is present, as described above. In turn, this prevents the data exchange between the threads within a single warp, if those threads are at different execution stages, or if they execute the instructions in different branches. That means that threads of different warps may execute instructions in parallel, but threads of a single warp sometimes have to execute instructions sequentially. Thus, algorithms that require data sharing at a high level of detail or that utilize synchronization tools (such as mutexes) can suffer from deadlocks. Therefore, developers have to rely on algorithms with minimal blocking support while using the NVIDIA GPUs of Pascal architecture or earlier (that is, more than 60% of devices, because Volta architecture was released only in 2017 and Turing — in 2018).

More modern architectures, starting from Volta, include independent flow planning, which stores instruction counters and call stacks for each thread separately, which can be utilized for optimal resource usage or to allow one thread to wait for data from another. To increase the level of parallelism, Volta includes an optimizer that defines how to group active threads within the warp within the SIMT module. This maintains the high performance of the SIMT approach, as in previous NVIDIA GPUs, but with much greater flexibility: the threads can now perform various branching paths within a single warp. The verification methods proposed are illustrated on a simple system model analysis example.

MODEL JUSTIFICATION AND VERIFICATION

Testing process has always been the main method of increasing the reliability of programs, developed using traditional methods. Edsger Dijkstra once said: “Program testing can be used to show the presence of bugs, but never to show their absence!”. In addition, testing can not detect typical synchronization errors of parallel programs. Parallel programs may for years retain errors that manifest themselves after a long usage period as a reaction to a specific combination of numerous factors that have arisen (for example, due to the unpredictable rates of individual threads/processes execution in parallel programs). However, if any of the system properties can be expressed formally, for example, in the form of a mathematical logic formula, then analysis of this property can be performed by verification methods. Normally process of the system verification consists of the following parts:

1. Construction of a mathematical model of the system under analysis.

2. Definition of the properties to be checked in the form of a formal text (also known as specification).

3. Building a formal proof of the presence or absence of the property being verified.

Usually, mathematical model of a system is a graph whose vertices are called states and represent situations (or situation classes), in which the system may be present at different times; whose edges can have labels depicting the actions system can perform. The functioning of the system in this model is represented by transitions along the edges of the graph from one state to another. If the passable edge has a label, then this label represents the action of the system, executed when passing from the state at the beginning of the edge to the state at its end.

The choice of the abstraction level for the system modelling depends on many factors (algorithmic solvability, astronomical dimensions of the model, the absence of effective methods of formal analysis of properties etc.). In this regard, the informal rules of this choice are reduced to the following: system model should not be over-specified, because the excessive model complexity may cause significant computational problems during its formal analysis. On the other hand, system model should not be oversimplified: it should reflect those aspects of the system that are relevant to the properties being verified, and preserve all the properties of the simulated system which are of interest for analysis.

Model checking (MC, [7]) approach is used to find a formal proof that the model does not meet its specification. In this paper we propose a new method of justifying that the model satisfies the specification. This paper focuses on the method of checking the conformity of a model an its specification, which uses the apparatus of TS and PN. Definition of a simple and labeled TS is given below, while definition of PN is well known and can be found in [7] if necessary.

Definition 1. A simple transition system is $A=(S,R,\alpha,\beta)$, where

S — finite or infinite set of states;

R — finite or infinite set of transitions;

α,β — two mappings from S to R, which make a correspondence between a transition $t\in R$ and two states $\alpha(t)$ and $\beta(t)$, which are called respectively the beginning and end of the transition t.

The transition t with beginning s and end s' is written as follows: $s\to s'$. Sometimes transitions may have a common beginning or end or both. This means that the pair $\alpha,\beta:R\to S$ is not necessarily an injective function. TS A is called finite if sets S,R are finite. If the set of states defines an initial state, such TS is labeled as $A=(S,R,\alpha,\beta,s_0)$ and is called initial TS. Such model as a simple TS may be enough to study properties of a model at a certain level of abstraction, but when it is necessary to carry out a more detailed analysis, labeled TS is more appropriate.

Definition 2. Let X be an alphabet. The labeled transition system (LTS) is an ordered six $A=(S,T,\alpha,\beta,s_0,h)$ where (S,T,α,β,s_0) represent a TS, and h is a mapping from T to X, which makes a correspondence between each transition t and its label $h(t)\in X$. LTS is finite if sets S, T, X are finite. Transition label $h(t)$ may also be called an action, and the transition itself is written as follows:

$(s, h(t), s')$ or $s \xrightarrow{h(t)} s'$. Transition label set is often accompanied by a special label τ, which represents an internal action of the system that is not visible at a given level of modelling.

Using LTS as a model of a real system allows analysing properties of actions associated with transitions, which is impossible by using a simple TS. Analysis and verification of the applications for graphic video adapters (NVIDIA in particular) was chosen as a subject of the study, because this area perfectly illustrates the impossibility of manual verification, as the number of threads allocated for solving the problem is measured by hundreds of thousands (in Pascal/Volta architectures). The use of LTS to construct a high-level model for substantiating the properties of a CUDA application was described in [8].

CUDA APPLICATION EXECUTION MODEL

A generalized execution model in the NVIDIA CUDA architecture, based on LTS and Petri nets was presented in [9]. The main emphasis was put on obtaining a high-level model without a detailed examination of the labels semantics of each transition system. The following briefly recalls the main details: three LTS were emerged as a result of CUDA application decomposition:

LTS $A = (S_1 = \{a_0, a_1, a_2, a_3\}, R_1, \alpha_1, \beta_1, a_0, h_1)$ — represents the warp that contains a set of instructions and sequentially executes them, where $X_1 = \{r_1, r_2, r_3, r_4\}$;

LTS $B = (S_2 = \{b_0, b_1, b_2, b_3, b_4, b_5\}, R_2, \alpha_2, \beta_2, b_0, h_2)$ — represents a generalized information instruction, where $X_2 = \{p_1, p_2, p_3, p_4, p_5, p_6\}$;

LTS $C = (S_3 = \{c_0, c_1, c_2, c_3\}, R_3, \alpha_3, \beta_3, c_0, h_3)$ — represents a thread block execution process (warp scheduling) on the SM, where $X_3 = \{q_1, q_2, q_3, q_4\}$.

Transition label functions h_1, h_2, h_3 are shown below in the charts (Fig. 1, 2, 3). The main activities modelled at this level of abstractions include planning, choosing warp and instruction for it and providing exclusive access to computational resources for the execution time of each warp-instruction tuple. This paper shows creation of a model of the lower level of abstraction, which will allow us to analyze the properties of actions associated with transitions, and the properties of the model in particular states.

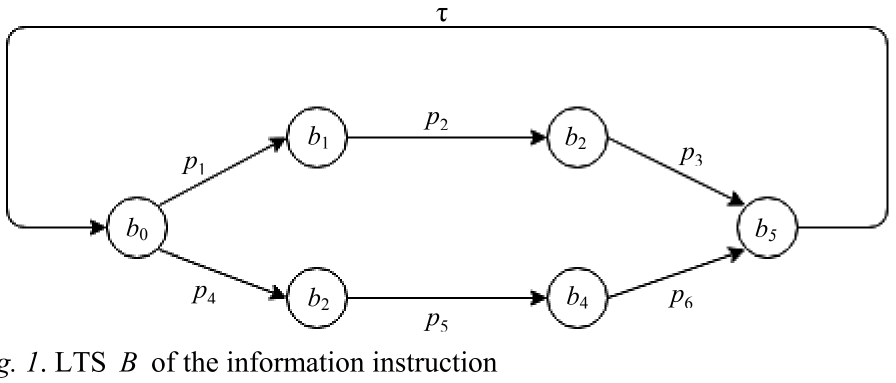


Fig. 1. LTS B of the information instruction

Definition 1. General TS (GTS) is a TS $A=(S,X,R,s_0,AP,L)$, where

S — set of states;

X — set of actions associated with transitions;

$R\subseteq S\times X\times S$ — transition relation;

s_0 — initial state from S;

AP — set of propositional formulas associated with states;

$L:S\to B(AP)$ — state label function, where $B(AP)$ is the power set of AP.

We introduce a set of Boolean variables:

$V_1=\{instrAr, instrMem, instrArExec, instrMemExec, instrArFin, instrMemFin\}$,

where

$h_1(\tau)=\{instrAr=0, instrArExec=0, instrArFin=0, instrMem=0,$
$instrMemExec=0, instrMemFin=0\}$;

$h_1(p_1)=\{instrAr=1\}$ — arithmetical instruction selection;

$h_1(p_4)=\{instrMem=1\}$ — global memory access selection;

$h_1(p_2)=\{instrArExec=1\}$ — arithmetical instruction execution;

$h_1(p_5)=\{instrMemExec=1\}$ — memory access execution;

$h_1(p_3)=\{instrArFin=1\}$ — retrieving results of an arithmetical operation;

$h_1(p_6)=\{instrMemFin=1\}$ — retrieving results of memory access operation.

We define the state label function L_1:

$L_1(b_0)=\{instrAr\vee instrMem=0\}$;

$L_1(b_1)=\{instrAr=1\}$;

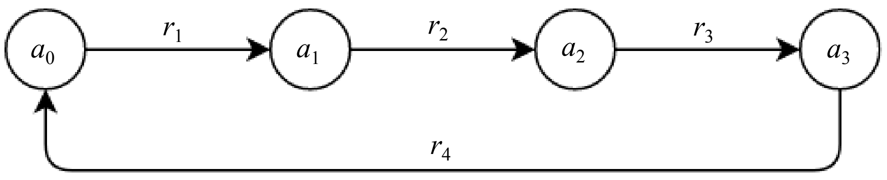
$L_1(b_2)=\{instrArExec=1\}$;

$L_1(b_3)=\{instrMem=1\}$;

$L_1(b_4)=\{instrMemExec=1\}$;

$L_1(b_5)=\{instrMemFin\vee instrArFin=1\}$.

We introduce a set of Boolean variables $V_2=\{warpActive, warpBusy,$ $warpFin\}$ that are set by the following transition labels (Fig. 2):



$$\begin{tikzcd} a_0 \arrow[r, "r_1"] & a_1 \arrow[r, "r_2"] & a_2 \arrow[r, "r_3"] & a_3 \arrow[lll, bend left, "r_4"] \end{tikzcd}$$

Fig. 2. LTS A of the instruction execution within a warp

$h_2(r_1)=\{warpActive=1\}$ — warp activation — given warp gains access to SM resources to execute a single instruction;

$h_2(r_2)=\{warpBusy=1\}$ — instruction is being executed by a warp;

$h_2(r_3)=\{warpFin=1\}$ — instruction execution is finished;

$h_2(r_4) = \{warpActive = 0, warpBusy = 0, warpFin = 0\}$ — warp deactivation.

State label function L_2 is defined as following:

$L_2(a_0) = \{warpActive \vee warpBusy \vee warpFin = 0\}$;

$L_2(a_1) = \{warpActive = 1\}$;

$L_2(a_2) = \{warpBusy = 1\}$;

$L_2(a_3) = \{warpFin = 1\}$.

We introduce a set of Boolean variables $V_3 = \{warpInstrSel, warpInstrExec, warpInstrFin\}$ that are set by the following transition labels (Fig. 3):

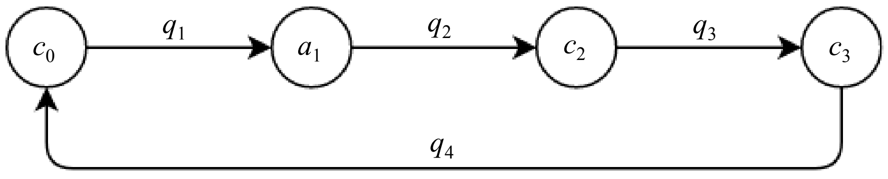


Fig. 3. LTS C of the warp scheduler

$h_3(q_1) = \{warpInstrSel = 1\}$ — warp and instruction selection;

$h_3(q_2) = \{warpInstrExec = 1\}$ — execution of the selected instruction by a selected warp;

$h_3(q_3) = \{warpInstrFin = 1\}$ — confirmation of instruction execution finish;

$h_3(q_4) = \{warpInstrSel = 0, warpInstrExec = 1, warpInstrFin = 1\}$ — transition to the next iteration.

State label function L_3 is defined as following:

$L_3(c_0) = \{warpInstrSel \vee warpInstrExec \vee warpInstrFin = 0\}$;

$L_3(c_1) = \{warpInstrSel = 1\}$;

$L_3(c_2) = \{warpInstrExec = 1\}$;

$L_3(c_3) = \{warpInstrFin = 1\}$.

Integration of multiple TS into a holistic system that orchestrates the joint work of all subsystems is performed depending on the requirements of the component interaction model (synchronous, asynchronous, parallel, sequential). These interaction methods are introduced using different TS composition types and the general notion of TS. To analyze the model properties, we consider the concept of parallel composition of the TS.

There are several options of TS composition that model parallel functioning of multiple TS. The simplest one is the composition in which TSs work in parallel, but do not interact with each other. Such a composition is based on the concept of alternating actions (interleaving), which are performed by different subsystems of the composition. In this case, the order of actions performed by each TS is preserved. This is a common way of modelling parallel interactions, which is based on the assumption that the result of a parallel execution of operations coincides with the result of their sequential execution. The formal definition of this composition is as follows.

Let $A_i = (S_i, X_i, R_i, s^i{}_0, AP_i, L_i)$ — transition systems, where $i = 1,2,...,n$. Parallel composition of TS $A_1, A_2,..., A_n$ with interleaving is TS

$A = A_1 \| A_2 \| ... \| A_n = (S, X, R, s_0, AP, L)$, where $S = S_1 \times S_2 \times ... \times S_n$, $s_0 = (s^1{}_0, s^2{}_0, ..., s^n{}_0)$, $X = X_1 \cup X_2 \cup ... \cup X_n$, and transitions from R are defined as follows: transition $((s_1,...,s_n), x, (s'_1,...,s'_n)) \in R$ if and only if state $(s'_1,...,s'_n)$ differs from $(s_1,...,s_n)$ by the value of not more than one component, that is there exists $i \in \{1,2,...,n\}$ such that $(s_i, x, s'_i) \in R_i$, where $x \in X_i \cup \{\tau\}$.

Note that in this composition TS can be repeated, that is, a given A_i may be included in TS A more than once. In addition, some of the TS A_i may be compositions of other TS: $A_i = A_{i1} \| A_{i2} \| .. \| A_{in}$. Therefore one can describe the whole system in a structured from by using a parallel composition. However, there may be achievable states in the resulting TS which are not desirable. Therefore a parallel composition with alternating actions does not always reflect the real situation when the TSs should interact with each other. Handshaking composition concept is a more adequate approach to describe parallel interaction. Such option describes a situation when different TS are synchronized by the actions in which multiple TSs participate simultaneously (data exchange etc.). Those actions are indicated by the same symbol in alphabets of all TS which take part in the interaction.

Definition 2. Let $A_i = (S_i, X_i, R_i, s^i{}_0, AP_i, L_i)$ — transition systems, where $i = 1,2$. Parallel composition of TS A_1, A_2 with handshaking is a TS $A = A_1 ||| A_2 = (S, X, R, s_0, AP, L)$, where $S = S_1 \times S_2$, $s_0 = (s^1{}_0, s^2{}_0)$, $X = X_1 \cup X_2$. Transitions that belong to T are defined as following:

if $x \in X_1 \cap X_2$ and $(s_1, x, s'_1) \in R_1$, $(s_2, x, s'_2) \in R_2$, then $((s_1, s_2), x, (s'_1, s'_2)) \in R$;

if $x \in X_1 \setminus X_2$ and $(s_1, x, s'_1) \in R_1$, then $((s_1, s_2), x, (s'_1, s_2)) \in R$;

if $x \in X_2 \setminus X_1$ and $(s_2, x, s'_2) \in R_2$, then $((s_1, s_2), x, (s_1, s'_2)) \in R$.

The following composition summarizes both previous concepts and is called a synchronized parallel composition or a synchronous product of TS (Fig. 4).

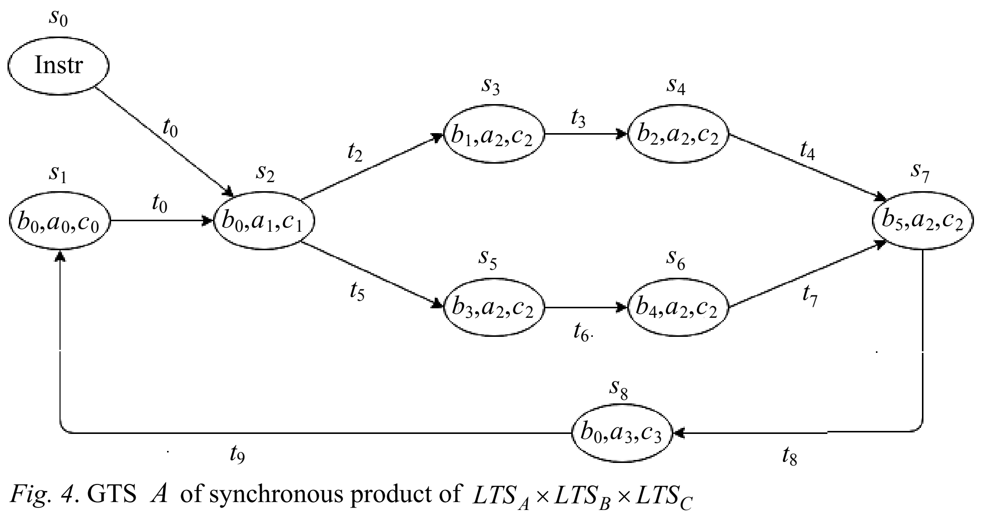


Fig. 4. GTS A of synchronous product of $LTS_A \times LTS_B \times LTS_C$

Definition 3. $A=(S_s,X_s,R_s,s_0,AP_s,L_s)=A_1\times\ldots\times A_n$, where $S_s=S_1\times\ldots$ $\ldots\times S_n$, $s_0=(s^1{}_0,\ldots,s^n{}_0)$ is called a synchronous product of TS $A_1,A_2,\ldots,A_n$ where $A_i=(S_i,A_i,R_i,s^i{}_0,AP_i,L_i)$. Set of transitions is divided into two classes: asynchronous and synchronous. When synchronous transition from state $s_1,\ldots,s_n$ occurs, some of its components change simultaneously while the rest remains unchanged. To describe this the symbol ε is added to labels of each transition set R_i (i.e. $X_i\cup\{\varepsilon\}$) along with a corresponding transition (s_i,ε,s_i). Such a symbol shows that state does not change during a particular transition. Subset R of $R_1\times\ldots\times R_n$ is called the synchronization constraints set. If TS is labeled, then the set of global transitions R corresponds to the set of transition labels X. In other words, $X\subseteq X_1\times\ldots\times X_n$, where X_i is an alphabet of LTS A_i, $i=1,\ldots,n$. An arbitrary element $A=(A_1,\ldots,A_n,R)$ of the set X is called the LTS synchronization vector.

We build a model of application execution in the CUDA architecture in the form of synchronous product with the following global transitions (initial and final states of the transitions are omitted, since they are present in the model):

$$R=\{t_1=(\varepsilon,r_1,q_1),t_2=(p_1,r_2,q_2),t_3=(p_2,\varepsilon,\varepsilon),t_4=(p_3,\varepsilon,\varepsilon)\ ;$$

$$t_5=(p_4,r_2,q_2),t_6=(p_5,\varepsilon,\varepsilon),t_7=(p_6,\varepsilon,\varepsilon),t_8=(\tau,r_3,q_3),t_9=(\varepsilon,r_4,q_4)\}.$$

DETERMINATION OF SEMANTIC CORRECTNESS OF THE MODEL ACTIONS

Definition function of L_i allow to find the context of correct system functioning at each transition in each state. For example, if LTS_B is in state b_0, values of both *instrAr* and *instrMem* should be 0. If system was moved to a state b_0 after a number of allowed transitions and *instrMem* flag is set, this will mean that system is not semantically correct, even if the corresponding PN meets all reliability criteria.

To analyze the semantic correctness of a model, it is necessary to determine the context of each global state and consider its conformity to the system under analysis. As shown above, transition system state context is specified by the state label function L_i, and the global state context can be retrieved by combining local state contexts of the components of the synchronous products. Each of the global states can be described as follows:

$$L(s_1)=L_1(b_0)\wedge L_2(a_0)\wedge L_3(c_0)=$$

$$=\{instrAr\vee instrMem=0\}\wedge\{warpActive=0\}\wedge\{warpInstrSel=0\}\ \text{—}$$

instruction is absent, warp is not active, scheduler is ready for planning;

$$L(s_2)=L_1(b_0)\wedge L_2(a_1)\wedge L_3(c_1)=$$

$$=\{instrAr\vee instrMem=0\}\wedge\{warpActive=1\}\wedge\{warpInstrSel=1\}\ \text{—}$$

warp and instruction are selected, but instruction type is not yet determined;

$$L(s_3)=L_1(b_1)\wedge L_2(a_2)\wedge L_3(c_2)=$$

$$=\{instrAr=1\}\wedge\{warpBusy=1\}\wedge\{warpInstrExec=1\}\ \text{—}$$

arithmetic instruction is selected for a given warp;

$$L(s_4) = L_1(b_2) \wedge L_2(a_2) \wedge L_3(c_2) =$$

$$= \{instrArExec = 1\} \wedge \{warpBusy = 1\} \wedge \{warpInstrExec = 1\} \text{ —}$$

given warp executes arithmetic operation;

$$L(s_5) = L_1(b_3) \wedge L_2(a_2) \wedge L_3(c_2) =$$

$$= \{instrMem = 1\} \wedge \{warpBusy = 1\} \wedge \{warpInstrExec = 1\} \text{ —}$$

memory access instruction is selected for a given warp;

$$L(s_6) = L_1(b_4) \wedge L_2(a_2) \wedge L_3(c_2) =$$

$$= \{instrMemExec = 1\} \wedge \{warpBusy = 1\} \wedge \{warpInstrExec = 1\} \text{ —}$$

given warp executes memory access instruction;

$$L(s_7) = L_1(b_5) \wedge L_2(a_2) \wedge L_3(c_2) =$$

$$= \{instrArFin \vee instrMemFin = 1\} \wedge \{warpBusy = 1\} \wedge \{warpInstrFin = 1\} \text{ —}$$

current instruction is executed regardless of its type;

$$L(s_8) = L_1(b_0) \wedge L_2(a_3) \wedge L_3(c_3) =$$

$$= \{instrAr \vee instrMem = 0\} \wedge \{warpFin = 1\} \wedge \{warpInstrFin = 1\} \text{ —}$$

instruction is absent, warp and scheduler finished their work.

All received contexts are semantically correct in accordance with the materials [1], thus, the set of global transitions of synchronous products is defined correctly.

In addition to the above, the semantic correctness of the model is provided by the following properties:

- *mutual exclusion*: a single instruction can be of one type only — either arithmetic or memory access;
- *fairness*: if the warp is active, scheduler must provide an instruction to be executed;
- *liveliness*: if the warp is active, one of the available types of instruction should be executed;
- *deactivation*: warp scheduler does not activate the warp if there are no instructions for execution.

Sequence $\sigma = s_0 \alpha_1 s_1 \alpha_2 ... \alpha_n s_n$ is called a finite execution in GTS, where $(s_i, \alpha_i, s_{i+1}) \in T$ for $i = 0,1,...,n-1$.

The following finite execution paths exist in the received LTS:

$$Path_1 = (s_1 t_1 s_2 t_2 s_3 t_3 s_4 t_4 s_7 t_8 s_8 t_9 s_1)^*;$$

$$Path_2 = (s_1 t_1 s_2 t_5 s_5 t_6 s_6 t_7 s_7 t_8 s_8 t_9 s_1)^*,$$

where denotes the iterative operation of regular language. The language that corresponds to the paths above is:

$$L(Path) = (L(Path_1) \vee L(Path_2))^* = (t_1(t_2 t_3 t_4 \vee t_5 t_6 t_7) t_8 t_9)^*.$$

And this language must correspond to the formula:

$$L(s_1)L(s_2)L(s_3)L(s_4) \vee L(s_5)L(s_6)L(s_7)L(s_8).$$

Let's check the semantic correctness properties of the model:

- *mutual exclusion*: the following expression is always true: $L(s_3) \vee L(s_5) = L_2(a_2)L_3(c_2)(L_1(b_1) \vee L_1(b_3))$;
- *fairness*: always $L_2(a_1) \to L_3(c_3)$;
- *liveliness*: always $L_2(a_1) \to L_1(b_1)L_2(a_2)L_3(c_2) \vee L_1(b_3)L_2(a_2)L_3(c_2)$;
- *deactivation*: $L_3(c_0) \to L_1(b_0)L_2(a_0)$.

FORMALIZATION OF THE GTS ANALYSIS PROCESS

Analysis process presented in the previous section can be generalized and reused for any GTS.

Definition 1. The subset of a set $B(AP)^*$, where $B(AP)$ is the power set of the set AP, is called the linear-temporal property P over the set of atomic propositional formulas AP. Consequently, $P \subseteq (B(AP))^*$.

In our case set $(B(AP))^*$ is a set of words of finite length constructed from concatenated formulas of $B(AP)$. Assume there is $A = (S, X, R, I, AP, L)$ and $\pi = s_0 s_1 ... s_n$ is a sequence of states. Such sequence is called a path fragment if $s_{i+1} \in Post(s_i)$ where $Post(s_i) = \cup Post(s_i, x)$, and $Post(s_i, x) = \{s' \in S :$ $(s_i, x, s') \in R\}$, $x \in X$. Sequence $\pi = s_0 s_1 ... s_n$ for which $Post(s_n) = \varnothing$ is called the maximal path fragment.

Definition 2. Word $L(s_0)L(s_1)...L(s_n)$ is called a trace $(trace(\pi))$ of a finite sequence π. Consequently, the set of traces is a set of finite words over the alphabet of the propositional formulas $B(AP)$ which are executed in states of this sequence. Denote $trace(\Pi) = \{trace(\pi) | \pi \in \Pi\}$, $trace(s) = trace(Path(s))$ and $Traces(A) = \cup_{s \in I} trace(s)$ where $Path(s)$ is maximal fragment of path π that begins in state s.

In our case, the set of AP propositional formulas includes the following items:

$$AP = \{instrAr, instrMem, instrArExec, instrMemExec, instrArFin, instrMemFin, warpActive, warpInstrSel, warpBusy, warpInstrExec, warpFin, warpInstrFin\}.$$

The following words are traces of GTS A:

$$p_1 = L(s_2)L(s_3)L(s_4)L(s_7)L(s_8)L(s_1);$$

$$p_2 = L(s_2)L(s_5)L(s_6)L(s_7)L(s_8)L(s_1).$$

Let's define bad prefixes in these words as prefixes that violate the truth of $L(s_i)$, which mean the following words:

$$p'_1 = \overline{L(s_2)}L(s_3)...L(s_1);$$

$$p'_2 = L(s_2)\overline{L(s_3)}...L(s_1);$$

$$p'_3 = L(s_2)\overline{L(s_5)}...L(s_1);$$

$$\dots\dots\dots\dots\dots\dots\dots\dots\dots\dots$$

$$p'_k = L(s_2)\overline{L(s_1)}.$$

Therefore language $BadPref(A) = (p_1 \vee p_2)^*(p'_1 \vee ... \vee p'_k)$ is regular and is accepted by a finite automaton $B = (Q, AP, f, Q_0, F)$ where $Q_0 \cap F = \varnothing$ that is shown at Fig. 5.

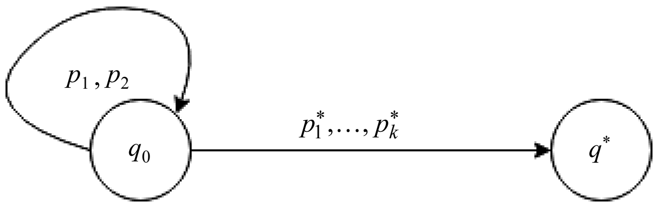


Fig. 5. Automaton B that accepts bad prefixes, $q_0 \in Q_0$, $q_0 \notin F$, $q^* \in F$

We will connect the GTS from Fig. 4 and automaton B with such a product $A \times B$ that produces GTS $\overline{A}$ as result so that:

$$S' = S \times Q,$$

R' — the smallest relation defined by the rule

$$\frac{s \xrightarrow{x} s' \wedge q \xrightarrow{L(s')} q'}{(s,q) \xrightarrow{x} (s',q')},$$

$$I' = \{(s_0, q_0) | s_0 \in I \wedge \exists q_0 \in Q_0 : q_0 \xrightarrow{L(s_0)} q\},$$

$$AP' = Q,$$

$$LS \times Q \rightarrow B(Q) \text{ where } L'(s,q) = \{q\}.$$

Then the correctness of GTS A functioning is expressed as a condition

$$Traces(A) \cap BadPref(A) = Traces(A) \cap L(B) = \varnothing,$$

where $L(B)$ is the language accepted by the automaton B. Consequently, if P is a property whose execution guarantees the correct functioning of GTS A, then $Traces(A) \cap BadPref(P) = \varnothing$. Thus, GTS $\overline{A}$ will look as shown on Fig. 6. There is a transition from each vertex, except for (s_0, q_0), to the state (s_2, q^*). Such transitions represent one of the bad prefix traces. For the sake of clarity only 4 examples of bad prefixes are left in Fig. 6.

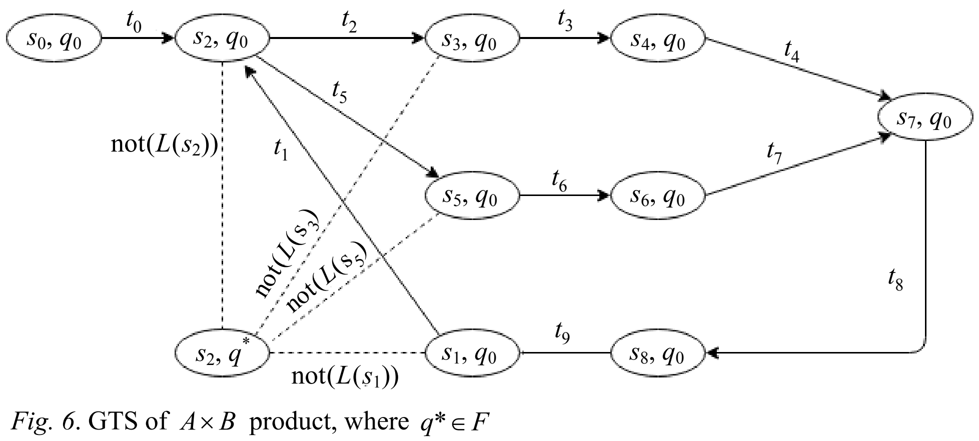


Fig. 6. GTS of $A \times B$ product, where $q^* \in F$

DETERMINATION OF MODEL CORRECTNESS ON THE HIGH LEVEL OF ABSTRACTION

In addition to semantic verification, it is important to check the redundancy of the system, deadlock/trap balance etc. To do this, we use verification at the highest level of abstraction, without need of transition label semantics analysis.

The translation of the received synchronous product into the PN gives the network shown in Fig. 7. It is described in [10], [11] that the TS product semantics and the semantics of the corresponding PN are consistent in the sense that a sequence of global transitions represents the global history of the TS product A if and only if it is an admissible sequence of transitions in the PN. Accordingly, elements of the set R become transitions of the PN, and the global states of the TS product (the set of states of each TS involved in the synchronous product before or after the global transition) become the places of the received network. We build a PN by using the synchronization constraints set, and such a network simulates the interoperability of all subsystems. Recall that NVIDIA video adapters operate with multiple warps at the same time, so there is a situation of synchronous and asynchronous execution, since different instructions may have different execution times and will not reach synchronization location at the same time. Therefore using the Petri net apparatus is expedient.

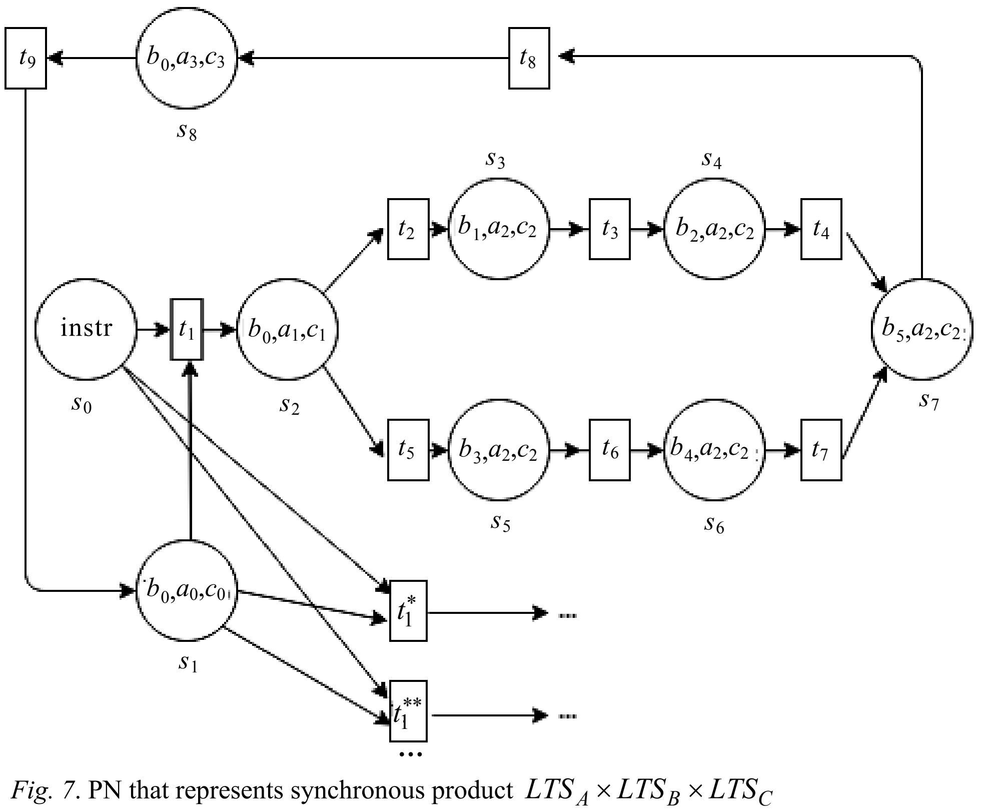


Fig. 7. PN that represents synchronous product $LTS_A \times LTS_B \times LTS_C$

Let's analyze the presence and number of dead transitions in PN for one warp case. To do this, we form a state equation $A \cdot x + M_0 - M_k =$

$= A \cdot x + d = 0$ where $M_0 = (1,1,0,0,0,0,0,0,0)$, $M_k = (0,1,0,0,0,0,0,0,0)$, $d = -(M_k - M_0)$ (see Table 1).

Table 1. Petri net state equation matrix

s	t									$-(M_k - M_0)$
	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	
s_0	-1	0	0	0	0	0	0	0	0	1
s_1	-1	0	0	0	0	0	0	0	1	0
s_2	1	-1	0	0	-1	0	0	0	0	0
s_3	0	1	-1	0	0	0	0	0	0	0
s_4	0	0	1	-1	0	0	0	0	0	0
s_5	0	0	0	0	1	-1	0	0	0	0
s_6	0	0	0	0	0	1	-1	0	0	0
s_7	0	0	0	1	0	0	1	-1	0	0
s_8	0	0	0	0	0	0	0	1	-1	0

Applying the TSS-algorithm [9] to solve the state equation with the above matrix, we obtain the following solutions (Table 2).

Table 2. Petri net state equation solutions

t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9
1	1	1	1	0	0	0	1	1
1	0	0	0	1	1	1	1	1

As can be seen from the set of solutions, all transitions in the PN with the initial and final markings given above are alive (the value corresponding to each transition is positive at least in one of the solutions). In addition, the property of *mutual exclusion* has been verified: transitions that correspond only to one of the possible types of instructions are performed at each point of time.

The analysis of the properties of the received PN also showed the absence of deadlocks, verified limitation and controllability [10]. As described above, this analysis did not take into account the properties of actions in transitions and properties associated with the states of the model.

CONCLUSION

This paper proposes the use of the TS mathematical apparatus to obtain a formalized system specification in the GPGPU technology. The advantages of the existing model include the ability to reduce synchronous product to a PN, which allows for further verification by automated means. The ability to study the characteristics of the model created by the combination of LTS and PN apparatuses is shown. The model was analyzed to verify there are no dead transitions and places (without taking into account the semantics of the transition labels), and a separate analysis was performed to verify the semantic correctness of the model actions.

As a result of these actions, the proof of the correct construction of the model was obtained. The developed approach allows to simplify and reduce the processes of verification and testing of multi-threaded applications in computer systems that utilize video adapters.

REFERENCES

1. Nvidia Data Center – Nvidia, 2018. [Online]. Available: https://www.nvidia.com/en-us/data-center/. Accessed on: 2019, March 14.
2. TOP500 Lists - TOP500 Supercomputer Sites, 2018. [Online]. Available: https://www.top500.org/lists. Accessed on: 2019, March 14.
3. A.V. Anisimov, S.D. Pogorilyy, and D.Yu. Vitel, “About the Issue of Algorithms formalized Design for Parallel Computer Architectures”, *Applied and Computational Mathematics*, vol. 12, no. 2, pp.140–151, 2013.
4. A. Arnold, *Finite Transition Systems: Semantics of Communicating Systems*. Paris, France: Prentice Hall, 1994, 177 p.
5. T. Murata, “Petri nets: properties, analysis and applications”, in *Proc. of the IEEE*, 77:541.80, 1989.
6. M. Ben-Ari, *Mathematical Logic for Computer Science*. UK: Prentice Hall International Ltd, 1993, 305 p.
7. E.M. Clarke, Jr., O. Grumberg, and D.A. Peled, *Model Checking*. USA: MIT Press, 1999.
8. S.D. Pogorilyy, S.L. Kryvyi, and M.S. Slynko, “Model justification of GPU-based applications”, *Control Systems and Computers*, vol. 4, pp. 46–56, 2018.
9. S.L. Kryvyi, *Linear Diophantine constraints and their applications*. Chernivtsi: Bukrek Publishing House, 2015.
10. S.L. Kryvyi, S.D. Pogorilyy, and M.S. Slynko, “Transition systems as method of designing applications in GPGPU technology”, in *Proc. 11-th international scientific and practical conference on programming UkrPROG’2018*.
11. S.L. Kryvyi et al., “Design of Grid Structures on the Basis of Transition Systems with the Substantiation of the Correctness of Their Operation”, *Cybernetics and Systems Analysis*, vol. 53, no. 1, pp.105–114, New York, USA: Springer Science + Business Media, January 2017.

Received 15.10.2020

INFORMATION ON THE ARTICLE

Serhii L. Kryvyi, ORCID: 0000-0003-4231-0691, Taras Shevchenko National University of Kyiv, Ukraine, e-mail: sl.krivoi@gmail.com

Sergiy D. Pogorilyy, ORCID: 0000-0002-6497-5056, Taras Shevchenko National University of Kyiv, Ukraine, e-mail: sdp77@i.ua, sdp@univ.net.ua

Maksym S. Slynko, ORCID: 0000-0001-9667-8729, Taras Shevchenko National University of Kyiv, Ukraine, e-mail: maxim.slinko@gmail.com

Artem A. Kramov, ORCID: 0000-0003-3631-1268, Taras Shevchenko National University of Kyiv, Ukraine, e-mail: artemkramov@gmail.com

МЕТОД СЕМАНТИЧНОЇ ВЕРИФІКАЦІЇ ЗАСТОСУВАНЬ У ТЕХНОЛОГІЇ GPGPU / С.Л. Кривий, С.Д. Погорілий, М.С. Слинько, А.А. Крамов

Анотація. Запропоновано метод розроблення та верифікації застосувань для систем з масовим паралелізмом на основі відеоадаптерів від компанії NVIDIA, який дозволяє створювати абстракції різних рівнів за допомогою апарата розмічених транзиційних систем. Композиції таких систем трансформуються в мережі Петрі, які далі аналізуються відповідними засобами. Метод також дає змогу створювати моделі на різних рівнях абстракції, а їх властивості можуть специфікуватися формулами темпоральної логіки. Це дозволяє досліджувати властивості систем з масовим паралелізмом, які майже неможливо аналізувати вручну, оскільки кількість потоків у новітніх архітектурах відеоадаптерів (Pascal, Volta, Amper, Тюрінг), виділених для виконання коду, вимірюється сотнями тисяч або мільйонами.

Ключові слова: CUDA, графічні процесори (GPU), графічні обчислення загального призначення (GPGPU), транзиційна система, мережа Петрі, побудова моделі.

МЕТОД СЕМАНТИЧЕСКОЙ ВЕРИФИКАЦИИ ПРИЛОЖЕНИЙ В ТЕХНОЛОГИИ GPGPU / С.Л. Крывый, С.Д. Погорелый, М.С. Слинько, А.А. Крамов

Аннотация. Предложен метод разработки и верификации приложений для систем с массовым параллелизмом на основе видеоадаптеров от компании NVIDIA, который позволяет создавать абстракции различных уровней с помощью аппарата размеченных транзиционных систем. Композиции таких систем трансформируются в сети Петри, которые далее анализируются соответствующими средствами. Метод позволяет создавать модели на различных уровнях абстракции, а их свойства могут специфицироваться формулами темпоральной логики. Это позволяет исследовать свойства систем с массовым параллелизмом, которые практически невозможно анализировать вручную, так как количество потоков в новейших архитектурах видеоадаптеров (Pascal, Volta, Amper, Тьюринг), выделенных для выполнения кода, измеряется сотнями тысяч или миллионами.

Ключевые слова: CUDA, графические процессоры (GPU), графические вычисления общего назначения (GPGPU), транзиционная система, сеть Петри, построение модели.