Formal Approach for GPU Architecture Schedulability

ZOUANEB Imane, BELARBI Mostefa
University of Ibn Khaldoun
Computer Sciences Department
Tiaret, Algeria
LIM Research Laboratory
i_zouaneb@yahoo.fr, belarbimostefa@yahoo.fr
CHOUARFIA Abdellah
University of Sciences and Technology Oran Mohamed Boudiaf
Computer Sciences Department
Oran, Algeria
LIM Research Laboratory
chouarfia@univ-usto.dz

Abstract— **Parallel application modelling and specifying is not an easy task to do because it treats tasks scheduling and time evolution. Graphics processing Unit is one of the main architectures that guaranties parallel execution. Event B is a skilled formal language based on sets theories. Our goal is to model and to specify the parallel execution of programs on GPU using Event B & RODIN platform. We are interesting to timing and scheduling of tasks on GPU.**

Key-Words : Parallel application, GPU, Formal specification, Timing, Scheduling, Event B.

I. INTRODUCTION

Parallel applications are the applications that can be divided into parts that can be executed in the same time. These parts do not depend on each other so they can be run simultaneously. Many-cores architectures permit to execute parallel applications thanks to its multiple processors. The Graphics Processing Unit (GPU) is one of these architectures and it is a puissant SIMD coprocessor (Single Instruction Multiple Data). The parallelism processing is granted by the big number of processing units on GPU. GPUs are used to improve applications execution such as multimedia applications and huge calculation applications. We call an application that is launched on GPU a kernel. This kernel is transformed into a grid of blocs. These blocs are divided into groups of 32 threads. When executing an application on GPU, we cannot see the different stages and the scheduling details. Modeling and specifying parallel applications is not a simple task to do. There are many tools to model this type of applications; one of them is formal methods. These latter are based on mathematic notions which make it sure and proved specification. Our goal is to model scheduling of kernel, blocs and threads and to propose a temporal model of tasks execution on GPU using Event B. The temporal model permits to show the time evolution when executing the tasks on GPU. Event B is a formal tool that allows us to create models and to validate it using automatic provers.

Event B does not support timing and scheduling of GPU tasks representation. Its mathematic bases permit to represent time evolution and scheduling process on GPU. Several works have dealt with time representation with Event B. Joris Rehm [1] has used Event B to model time constraints of the final step (root contention) of the distributed algorithm of the leader election protocol from IEEE 1394[2]. The proposed work consists of representing time and timers as additional variables of the system. They proposed to separate between the application model and the time constraints model so they refined the application model in a new model containing time evolution events. These events can be observed only when the system reaches a specific time which was named active time. This method was applied on several applications and it was also validated by Rodin in [3][4]. Another approach [5] was proposed to represent and to refine discrete time properties in Event B. They dealt with three main categories of discrete timing for trigger-response pattern: deadline, delay and expiry. These three kinds of timing constraints are used in many categories of time critical systems. For scheduling representation a set of works have treated it in Event B in different filed. The work of [6] proposed an approach to model concurrent scheduling. They presented an Event B model that covers the different interactions and concurrence of the famous problem of philosophers dinning through successive refinements. Another work [7] has dealt with modeling of event driven interaction in multi-agent systems. They have specified and proved interaction and scheduling between events using

Event B. In this paper, we propose a new approach to model timing and scheduling of tasks execution on GPU using Event B.

The present paper carries on in Section 2 by presenting Event B. Then, in Section 3 we introduce GPU architecture and scheduling on GPU. Section 4 shows the proposed formal specification of task execution on GPU and its scheduling and timing. Finally we conclude our work and we propose some perspectives.

II. EVENT B

Event B is an enriched extension of the formal method B created by J. R Abrial [8] for system specification, design and coding. It is based on Set theory and it specifies the system by abstract machines, operations and successive refinements which permit to prove, to verify and to validate the specified system.

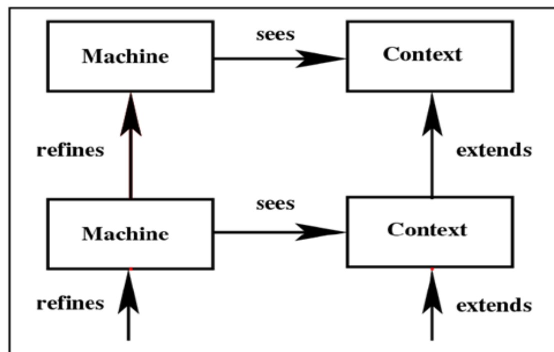

Fig.1 Refinements of models and contexts

Event B is based on MODEL notion which describes the labeled transaction of the system, named also machine in B method. A MODEL is composed of a static part which contains the states, its invariants and its properties and a dynamic part containing transitions (events). A MODEL has a name, variants, invariants and Events. A MODEL is completed by a formalism called the CONTEXT. It plays an important role in MODEL parameterization and instantiation. A CONTEXT has also a name, Sets, Invariants. [9][10] Each MODEL can reference a CONTEXT and many refinements which concrete models and contexts as it is shown in the figure 2. The Event B method is efficient because it uses tools like Atelier B[1] and the platform **RODIN** (Rigorous Open Development Environment for Complex Systems). This platform is a tool to develop and to prove Event B specification under Eclipse environment. [9] The main objective of RODIN is to create a methodology and supporting open tool platform for cost-effective, rigorous development of complex dependable software systems and services. [11]

III. GRAPHIC PROCESSING UNIT (GPU)

Graphic Processing Unit (GPU) is a puissant many core processor. GPU have a high performance processors dedicated to graphics processing. Originally, GPUs were oriented to accelerating graphics rendering functionality. Lately they are used to perform different kinds of general purpose computations in a parallel way to minimize application's runtime. [12]

A. *GPU Architecture*

GPU is a multi-core architecture used to enhance intensive computing and to discharge the CPU. A GPU is composed of a global memory and a set of Streaming Multiprocessor (SM). Each streaming multiprocessor is constituted of a set of Streaming Processor (SP) and each streaming processor is linked to a local memory (Register memory). And the SPs of a SM are linked to a shared memory. [13]

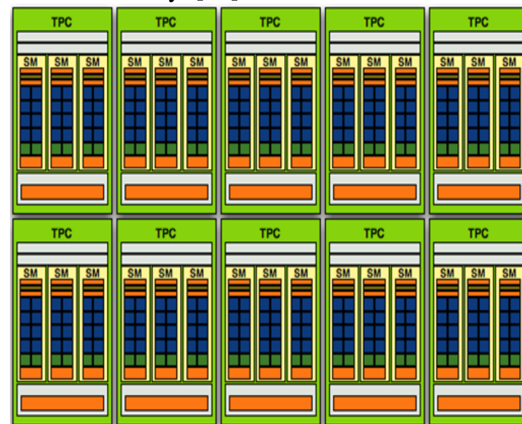

Fig.2 Nvidia GPU architecture

In Nvidia architecture, tasks are executed using SIMD (Single Instruction Multiple data) blocs written in CUDA. [14] CUDA (Compute Unified Device Architecture) provides a set of software libraries, an execution environment and a multitude drivers for different languages of programming (C,C++,…). CUDA is an extension of C language for programming on NVIDIA GPU. The computations on a GPU are programmed as *kernel* functions. A kernel program describes the execution of a serial *thread* on a GPU. The kernel is launched by the host CPU with specified numbers of blocs and threads, where a *bloc* represents a set of a certain number of threads, and all blocs in that kernel launch have the same numbers of threads. [13][14] The figure 3 shows the architecture of CUDA.

[1] Atelier B is a tool that permits operational use of the method B : http://www.atelierb.eu

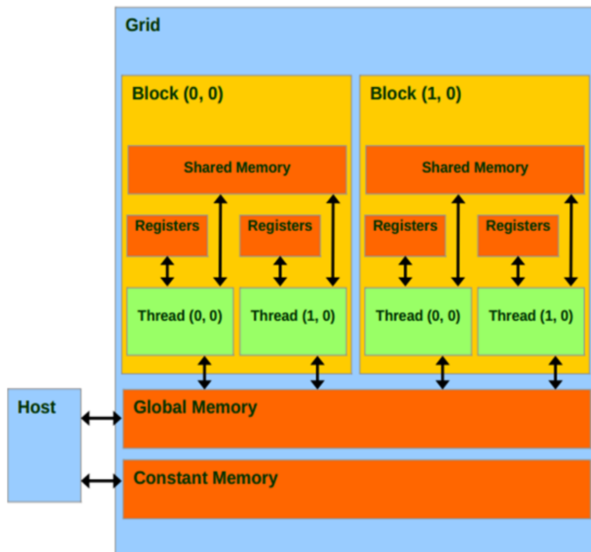

Fig.3 CUDA Architecture

B. *Scheduling on GPU*

Programs launched on GPU are called kernels. One kernel can be executed on a GPU in an instant. When a kernel is launched, it will be affected to a GPU and input data will be transferred from CPU Memory into GPU global memory. The kernel is represented by a grid composed of a set of blocs. Each bloc is constituted of a group of 32 threads. A bloc is executed on a SM of the GPU. If the number of available SMs on GPU is insufficient to execute all blocs in parallel, the blocs will be affected to free SMs and the reminding blocs will be added into a FIFO (First In First Out) waitlist. When a SM is liberated, the first bloc in the waitlist will be affected to this SM. In a bloc, the threads are executed in a parallel way in groups of 32 threads. The concurrence between the running threads of a bloc impact coherence memory (shared memory, global memory). In a bloc threads can communicate with each other using memory and synchronization barriers but threads of different blocs cannot be synchronized. When the grid finishes its execution, the result (output data) will be transferred to CPU.

IV. Proposed Event B Specification of Tasks Execution on GPU

In order to specify execution tasks on GPU, we propose an Event B model of kernel. This model is successively refined to show execution details. We have four levels: kernel execution, bloc scheduling, bloc execution and thread execution. A GPU context is added to define machines variables.

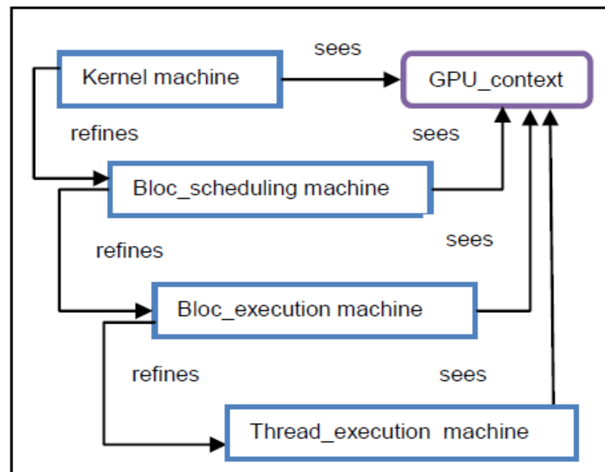

Fig.4 Elements of GPU executionspecification

A. *Basic model structure (kernel machine)*

The GPU kernel is defined by the variables:

- **nb_SM_GPU:** represents the number of SMs in the GPU of execution.
- **nb_kernel_threads:** represents the total number of kernel's threads.
- **Time_start:** represents the time of execution starting.
- **Time_end:** represents the time of execution end.
- **T_ev:** represents the time evolution.
- **GPU_OCC:** Boolean variable used to check if the GPU is free or taken.
- **K_state:** represents the state of kernel.
- **affect:** number of blocs.
- **nbreiter:** number of blocs according to the number of SMs on the GPU.
- **blocsArray:** a table that represents the blocs states.
- **blocs_start_time:** a table that represents the time of execution beginning of blocs.
- **blocs_end_time:** a table that represents the time of execution end of blocs.

The kernel machine has three events: waiting, execution and Endexecution. While the GPU is not free, the kernel waits.

```
WAIT ≙
  WHEN
 grd1 : GPU_OCC=TRUE
  THEN
 act1 : k_state:=wa ting
  END
```

If the GPU_OCC variable is equal to false, the kernel will starts its execution. The value GPU_OCC will be changed to True and the kernel state will be "**executing**". To devise the kernel on blocs, the total number of threads is divided on 32. Then, the result is divided on the number of SMs on the GPU. This value defines the number of blocs that can be executed in parallel on available SMs of execution architecture (GPU). The T_ev

variable is incremented by c value, the time of division and affectation of blocs to SMs.

```
    EXECUTION  ≙
      WHEN
    grd1  :  k_state=waiting
    grd2  :  GPU_OCC=FALSE
      THEN
    act1  :  GPU_OCC:=TRUE
    act2  :  k_state:=executing
    act3  :  affect:=nb_kernel_threads÷32
    act4  :  nbreiter:=affect÷nb_SM_GPU
    act   :  T_ev:=Time_start+c
      END
```

The kernel finishes its execution when all the elements of bloc states are equal to end. So it liberates the GPU and save the time of execution end.

```
    ENDEXECUTION  ≙
ANY
        m
WHERE
        grd3  :  m∈(0··nb_kernel_threads)
        grd1  :  k_state=executing
        grd2  :  blocsArray(m)=end
THEN
        act1  :  GPU_OCC:=FALSE
        act2  :  Time_end:=T_ev
        act3  :  k_state:=ending
END
```

B. *Scheduling modeling*

In the kernel machine, the kernel is divided into blocs of threads. These blocs must be scheduled to be executed on the available SMs of the executing GPU. To represent scheduling interaction in the kernel we proposed to use bloc state array (blocsarray). This array is modified in each stage of execution. Its dimension is the number of blocs calculated in the kernel machine. The values of the array's elements are initialized with "wait" in the beginning of execution. When the kernel is launched and divided into blocs the first 16 blocs will start there execution and there values in blocsarray is modified into "run". A table of 32 elements is created representing the threads states of the bloc, called threadsArray. When a thread is executing, it will change its state in the threadsArray table.

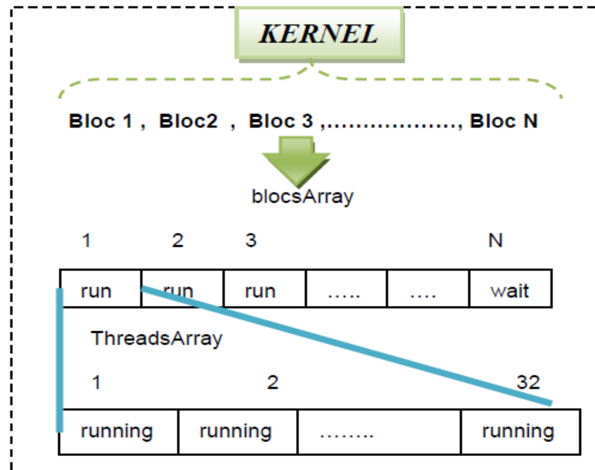

Fig.5 Arrays of states of blocs and threads

A bloc cannot liberate a SM until the 32 threads states are all equal to "finishing". So the bloc state will be changed to "end" and liberate the SM. These arrays permit the control and the evolution of parallel execution process. The kernel ends its execution when all the blocstates' elements are equal to "end", so it will liberate the GPU.

These arrays permit the control and the check of parallel execution process of the kernel, the blocs and the threads.

C. *Timing modeling*

To model time evolution, we proposed to use a variable (T_ev) that will be initialized by 0, then it will be incremented. To calculate blocs and threads timing we used tables for saving starting time execution and ending time execution. T_ev is incremented in kernel machine by the duration of kernel decomposition (c). When the kernel is launched the elements of blocs_start_time array will be initialized by T-ev (T_ev=c, in the beginning of bloc execution). blocs_time_end array also is initialized by T_ev. If a bloc starts its execution, two tables of 32 elements will be created called (threads_start_time, threads_end_time).

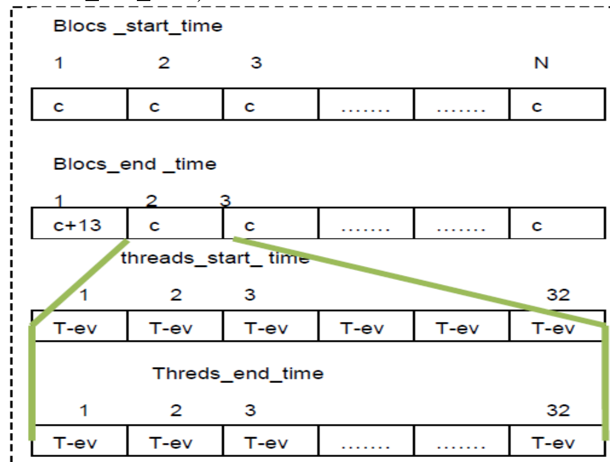

Fig.6 Arrays of timing of blocs and threads

The two arrays threads_start_time, threads_end_time are initialized by the value of

bloc_start_time. When a thread starts its execution, the duration of this latter will be added to strat_time_thread. If the thread needs access to global memory and this latter is not accessible, the time of waiting is added to its runtime until getting access to memory. When all threads of a bloc finish there execution, the maximum of the threads_end_time of the 32 threads is affected to bloc_time_end of this bloc. The maximum of end_time_blocs is affected to T_ev and this latter is the runtime of the kernel.

D. *Refinements of the basic model*

D.1 *Bloc_scheduling machine*

When the kernel execution is launched, the first 16 blocs will be affected to the available SMs (16 in our GPU). So there states will be changed into "run". There states and there starting times are initialized in the Affectation event.

```
      Affectation  ≙
WHEN
        grd2  :  j<k
THEN
        act4  :  current_bloc≔j−(nb_SM_GPU+1)
        act1  :  blocsArray(j)≔run
        act2  :  blocs_start_time(j)≔T_ev
        act3  :  j≔j+1
END
```

After having launched the sixteen blocs execution, the rest of blocs are all waiting for an SM to liberate.

When a bloc is in state run, a table of 32 elements is created, it is called threadsArray.

D.2 *Bloc_execution machine*

When a bloc is affected to a SM and its state is changed to "run", it creates the threads-state and the timing arrays. These arrays are initialized in the event bloc_executing. The threadsArray elements are initialized with "ready".

```
      bloc_executing  ≙
ANY
        M
WHERE
        grd1  :  m∈ℕ
        grd2  :  blocsArray(m)=run
        grd3  :  pointeur1≤32
THEN
        act1  :  threadsArray(pointeur1)≔ready
        act2  :  pointeur1≔pointeur1+1
END
```

When a bloc is in execution, the threads are running in parallel. Some threads can finish there execution and the others can't. So there is a verification event that verifies the execution end and saves the runtimes of the threads in a Time Set.

```
      bloc_ending_verification  ≙
ANY
        m
WHERE
        grd1  :  m∈ℕ
        grd2  :  blocsArray(m)=run
        grd3  :  pointeur2≤32
        grd4  :  threadsArray(pointeur2)=finishin
THEN
        act2  TimeSet≔TimeSet∪{threads_end_t
        :     (pointeur2)}
        act3  :  pointeur2≔pointeur2+1
END
```

When all the threads of a bloc finish there execution, its blocsArray value is changed to "end" and the maximum of threads execution is affected to its bloc_end_time.

```
      bloc_ending  ≙
REFINES
        verifying_execution_end
ANY
        m
WHERE
        grd2  :  m∈ℕ
        grd1  :  blocsArray(m)=run
THEN
        act1  :  blocs_end_time(m)≔max(TimeS
        act2  :  T_ev≔max(TimeSet)
        act3  :  blocsArray(m)≔end
END
```

D.3 *Thread_execution machine*

When a thread is created, it is initialized with the state "asleep". If its dominant bloc is activated, the thread state is modified to "ready" state and its starting time is initialized. This step is represented by the election event.

```
      Thread_election  ≙
ANY
        m
WHERE
        grd2  :  blocsArray(m)=run
        grd1  :  threadsArray(pos)=asleep
THEN
        act1  :  threadsArray(pos)≔ready
        act2  :  threads_start_time(pos)≔T_ev
        act3  threads_end_time(pos)≔threads_
        :     start_time(pos)
END
```

When the threads are running, they need access to global memory. This access could be happen in the same time, so we propose to use a variable that controls the access memory. The thread is waiting while the global memory is inaccessible. This thread's execution time is incremented by the time of waiting.

Thread_waiting ≙

WHEN
grd1 : threadsArray(pos)=ready
grd2 : MGA=notaccessible
THEN
act1 : threads_end_time(pos):=threads_s s)+1
END

When the memory becomes accessible, the thread passes to the "running" state and it will be executed. The duration of its execution is added to its thread_end_time value.

Thread_running ≙
WHEN
grd1 : threadsArray(pos)=ready
grd2 : MGA=accessible
THEN
act3 : MGA:=notaccessible
act1 : threadsArray(pos):=running
act2 : threads_end_time(pos):=threads_end_time(pos)+duration
END

When the thread finishes its execution, it liberates the global memory and modifies its state to "finishing".

Thread_finishing ≙
WHEN
grd1 : threadsArray(pos)=running
THEN
act1 : threadsArray(pos):=finishing
act2 : MGA:=accessible
END

V. Conclusion

In this paper we proposed a formal specification of GPU tasks execution using Event B language. The proposed specification models the Nvidia GPU's programming model. The programming model of a GPU consists of executing kernels in the form of grids composed of blocs and these blocs are composed of threads. This organization has been specified using successive refinements of the basic model which is the kernel in Event B using Rodin platform. In our specification, we tried to model the scheduling on the GPU and the timing of each component (kernel, bloc, thread). Another aspect was treated which is the access memory concurrence. The complexity of our specification is measured by the number of proof obligations which are automatically/manually is charged (see table 1).

Model	Total	Auto	Manual
Kernel	25	15	0
blocks_scheduling	22	12	0
Block_execution	38	18	0
Thread_execution	17	10	0
Total	102	55	0

Tab .1 Summary of proof obligations

We remark that automatic proofs changes from a model to the other, in the bloc_execution model there are more proofs that are not handled by Rodin provers. We didn't used the manual proofs or import hypothesis to discharge obligations proof to see the correctness of our specification.

As a part of our future works, we aspire to model specific parallel applications such as matrix multiplication and image processing on different GPU's architectures by refining our proposed basic Event B specification. Another perspective is to generate a valid executable code in CUDA and OpenCL from the Event B specification of parallel applications

References

[1] Joris Rehm, "A method to refine time constraints in event B framework," 6th International workshop on Automated Verification of Critical Systems (AVoCS 2006), Nancy,France, pp. 173-177, 2006.
[2] J-R. Abrial, D. Cansell and D Méry, "A mechanically proved and incremental development of IEEE 1394 tree identify protocol," Formal Asp. Comput. 14, pp. 215-227, 2003.
[3] Dominique Cansell, Dominique Merey, and Joris Rehm, "Time constraint patterns for event B development," in 7th International Conference of B Users, Besançon , France, pp. 140-154, 2007.
[4] Joris Rehm, "A duration pattern for event-B method," in 2nd Junior Researcher Workshop on Real-Time Computing (JRWRTC 2008), Rennes, France,2008.
[5] Mohammad Reza Sarshogh, Michael Butler, "Specification and refinement of discrete timing properties in Event-B," in the 46th *ECEASST*, 2011.
[6] Pontus Boström, Fredrik Degerlund, Kaisa Sere and Marina Waldén, "Concurrent Scheduling of Event-B Models," in *EPTCS,Vol.55, pp.166-182, 2011.*
[7] Lorina Negreanu and Matei Popovici, "Modeling and proof of event-driven interaction in multi agent systems in Event-B," 19th International Conference on Control Systems and Computer Science, Bucharest, Romania, pp. 180 – 183, 2013.
[8] J.R Abrial, "The B-book: Assigning programs to meanings", 1996.
[9] C. Métayer, J.-R. Abrial, L. Voisin, "Event-B Language", May 2005.
[10] Yamine AIT-Ait-Ameur & all. "Vérification et validation formelles de systèmes interactifs fondées sur la preuve : application aux systèmes multi-modaux," IN Journal d'Interaction Personne-Système, Vol. 1, No. 1, Art. 3, Septembre 2010.
[11] Joey Coleman, Cliff Jones, Ian Oliver, Alexander Romanovsky, and Elena Troubitsyna, "RODIN (Rigorous Open Development Environment for Complex Systems),"In 5th European Dependable Computing Conference: EDCC-5 supplementary volume, April 2005.
[12] Sylvain Collange, Yoginder S. Dandass, Marc Daumas, and David Defour, "Using Graphics Processors for Parallelizing Hash-Based Data Carving,"In *HICCS*, Hawaii International Conference on System Sciences, pp 1-10, 2009.

[13]] Peter N. Glaskowsky. "NVIDIA's Fermi: The First Complete GPU Computing Architecture", 2009. [Online]. Available:http://www.nvidia.com/content/PDF/fermi_white_papers/P.Glaskowsky_NVIDIA'sFermi-he_First_Complete_GPU_Architecture.pdf

[14] Sylvain Collange, Marc Daumas, David Defour & Régis Olivés, "Fonctions élémentaires sur GPU exploitant la localité de valeurs", In SYMPosium en Architectures nouvelles de machines, Fribourg, Switzerland, pp 1-11, 2008.