Introduction to the HAPI© Timed Discrete-Event Simulator

M.J.G. Bekooij

October 28, 2015

Chapter 1

HAPI

1.1 Overview of HAPI

HAPI is a timed discrete-event simulator build on top of the SystemC library. HAPI is intended to obtain the schedules of self-timed executed dataflow graphs. HAPI has build-in support for modeling shared resources. Worst-case as well as typical schedules can be obtained with HAPI. Therefore the HAPI results can be used to validate upper and lower bounds that are obtained with analytical techniques. With HAPI the schedules of parallel applications that are executed on multiprocessor systems can be obtained.

HAPI is implemented as C++-classes. The user should describe the dataflow graph in C++ and compile the source code with the g++ compiler which produces an executable of the program. By executing the program on a personal-computer the simulation results are obtained. The schedules obtained with HAPI are stored in a so-called vcd file and can be viewed with the open-source Gtkwave tool.

HAPI has build-in support to model the effects of shared resources. Such a shared resource can be used to model processors on which multiple tasks are scheduled. Several arbitration policies for shared resources are supported. Examples of these policies are Time-Division Multiplex (TDM), Round-Robin (RR) and Fixed Priority Preemptive (FPP).

The dataflow graphs that HAPI supports can be static such as HSDF, SDF, and CSDF as-well-as dynamic such as VRDF, VPDF, and FDDF. These dataflow graphs may contain auto-concurrent actors and unbounded queues.

HAPI is suitable for simulation of functional deterministic dataflow graph as well as functional non-deterministic dataflow graphs.

The firing durations of the actors in HAPI can be constants, can depend on the input data, or can be generated by a random number generator.

As stated above HAPI is intended to obtain schedules of self-timed executed dataflow graphs. However with HAPI also the schedules can be obtained of dataflow graphs that contain time triggered actors.

With HAPI the combined functional and temporal behavior of dataflow graphs can be obtained because functionality can be implemented inside the dataflow actors.

With HAPI it can be validated whether an application that is modeled as

a dataflow graph respects its throughput and latency constraints. To this end periodic source and sink actors are supported that report an error in case they cannot run run-strictly periodically because they are blocked due to lack of space or data. Furthermore facilities for measuring the average and worst-case latency are provided.

HAPI is build on top of the SystemC library. After compilation of the HAPI program a single threaded program is create. Therefore the functional code in each actor of an HAPI program is executed without being preempted. The internal notion of time of the simulator is called the simulation time. The simulation time is advanced by the execution of the systemC wait-statements inside the functional code of the actor. Each execution of a wait statement places a new event with a time-stamp in the ordered event queue of the simulator. The arrival of events can cause an enabling of an actor. The actors will fire in the order they are enabled according to the time-stamps, i.e. actors that are enabled by time-stamps with a lower value are fired first. In case of simultaneous enabling the firing order of the actors is chosen arbitrarily. Actor firings will produce new events that are stored in the order of their time-stamps in the event queue of the simulator.

HAPI provides an API for describing dataflow graphs and modeling task graphs. This API provides a number of options to make the description of often occurring cases more concise. Because HAPI is build on top of the SystemC library also fixed point data types are supported. The SystemC documentation should be consulted for more information about these types.

HAPI is provided as open-source code under the GPL license. HAPI, or more precisely HAPILight, has been created by Maarten Wiggers and has been extended by Stefan Geuns, and Joost Hausmans during their PhD/Postdoc at the University of Twente while being located at Philips/NXP in Eindhoven.

1.2 Source code examples

A number of HAPI source code examples are provided that illustrate the use of specific features of HAPI. The examples can be found in the directory HAPILight/examples. These examples are:

1. Simple HSDF graph modeling a producer, and a consumer that communicate via a FIFO buffer. The HSDF graph does not contain auto-concurrency and the actors do not contain functionality. The schedule becomes periodic because the actors have constant firing durations.

(a) HSDF graph

(b) Corresponding task graph

Figure 1.1: HSDF and task graph corresponding to example 1.

2. A simple HSDF graph with a producer and a consumer of which the producer can be executed auto-concurrently. This results in a slightly confusing representation of the schedule in gtkwave

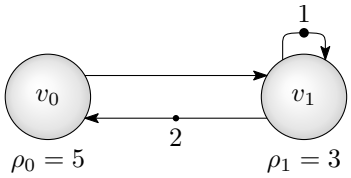


Figure 1.2: HSDF graph corresponding to example 2.

3. A simple HSDF graph with a producer and a periodic sink actor. The producer has a random firing duration. Larger FIFO capacities will cause that a violation of the periodic execution constraint will become much less likely.

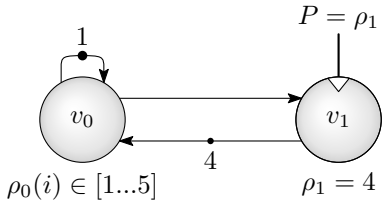


Figure 1.3: HSDF graph corresponding to example 3.

4. (a) Simple HSDF graph of a producer and a consumer of which the producer is scheduled by a TDM arbiter on a shared resource. This example illustrates that a larger FIFO buffer capacity results in a higher throughput because multiple firings can happen during a time-slice.

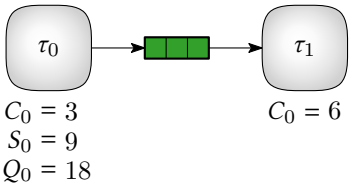


Figure 1.4: Task graph corresponding to example 4a.

   (b) Same as 4a but the TDM arbitration modeled with a latency-rate actor.

5. An HSDF graph with a periodic source and two actors scheduled under fixed priority preemptive scheduling. This example illustrates that a larger difference between the best-case firing duration and the worst-case firing duration reduce throughput and can cause that the source cannot run strictly periodically.

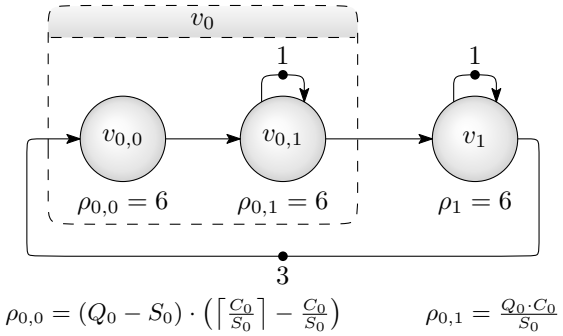


Figure 1.5: HSDF graph corresponding to example 4b and corresponding to the task graph of example 4a.

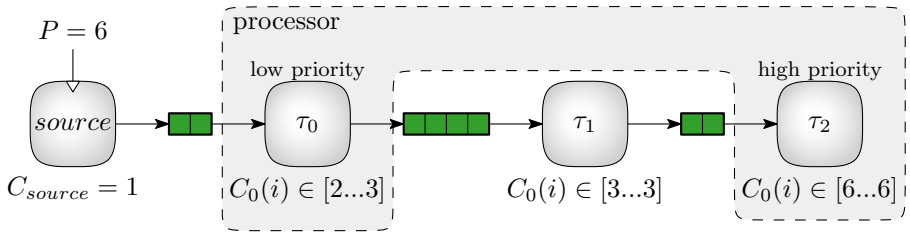
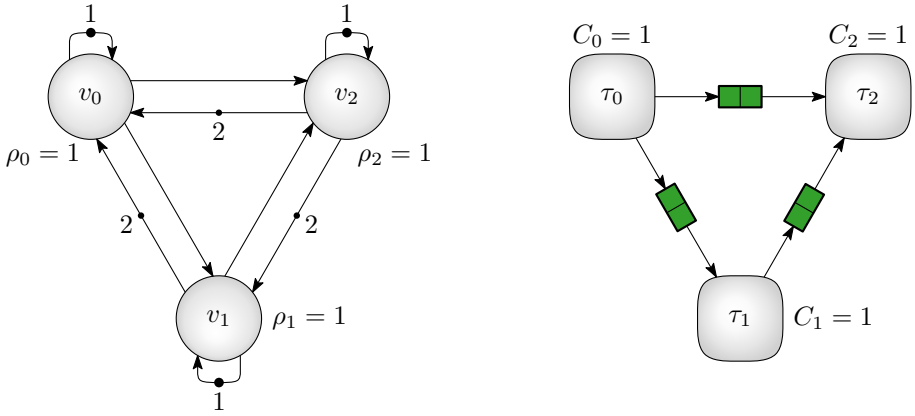


Figure 1.6: Task graph corresponding to example 5. Tasks τ_0 and τ_2 share a processor that uses a fixed priority preemptive scheduler. Task τ_0 has the lowest priority and task τ_1 the highest.

6. An HSDF graph consisting of 3 actors. This examples shows that the maximum throughput of one firing every ms is not obtained if all buffers have a capacity of two tokens. Which buffer should be increased can be found with MCM analysis.

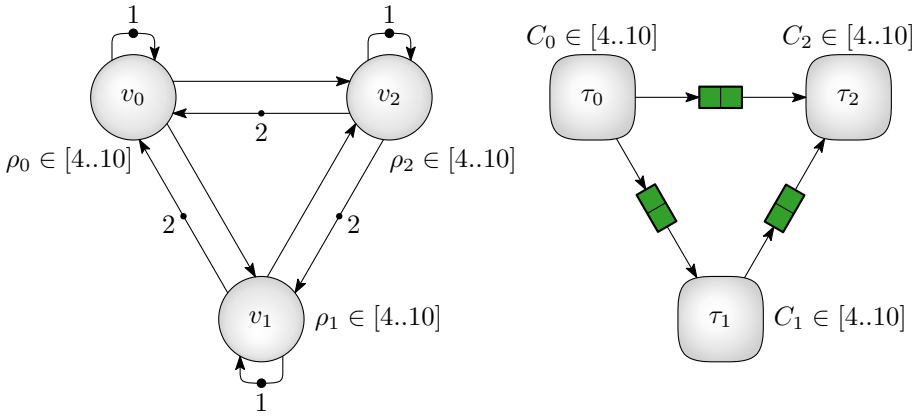


(a) HSDF graph

(b) Corresponding task graph

Figure 1.7: HSDF and task graph corresponding to example 6.

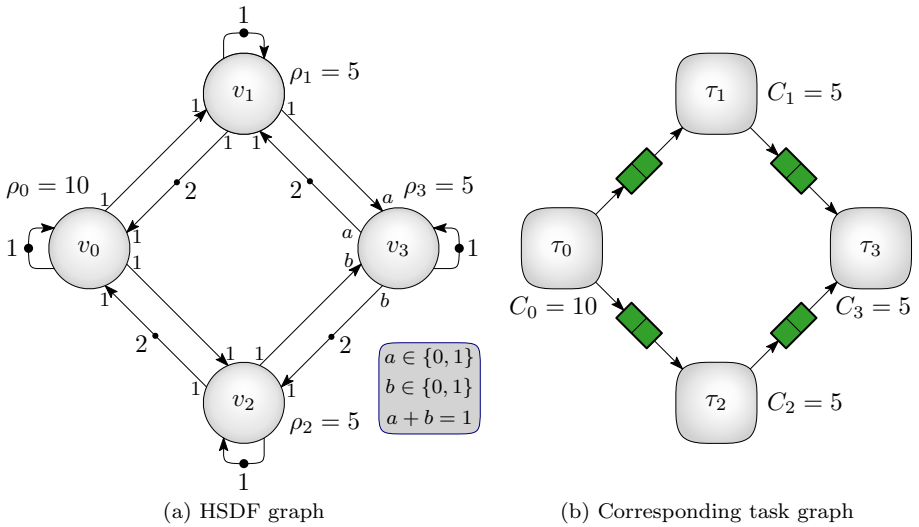
7. An HSDF graph with actors that have a functional behavior. The same sequence of output values is always produced because the dataflow graph is functionally deterministic despite that it is temporally non-deterministic as a result of random firing durations.



(a) HSDF graph (b) Corresponding task graph

Figure 1.8: HSDF and task graph corresponding to example 7.

8. A dataflow graph exhibiting functionally non-deterministic behavior by using the peek function. As a result the functional behavior is non-deterministic.



(a) HSDF graph (b) Corresponding task graph

Figure 1.9: HSDF and task graph corresponding to example 8.

9. Simple SDF graph containing 3 actors that communicate via a FIFO buffer. The graph is inconsistent and as a consequence deadlocks.

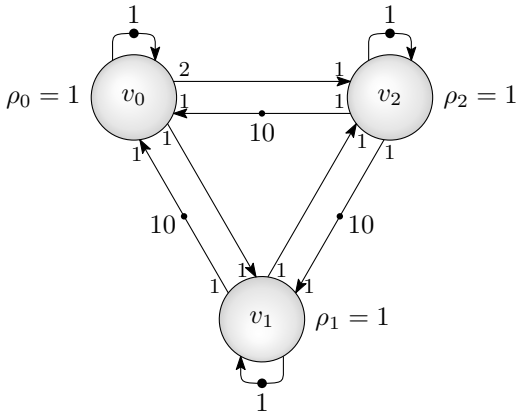


Figure 1.10: HSDF graph corresponding to example 9.

10. A functionally deterministic dataflow graph that closely resembles a Kahn process network. Deadlock can occur because the number of tokens consumed and produced changes dynamically.

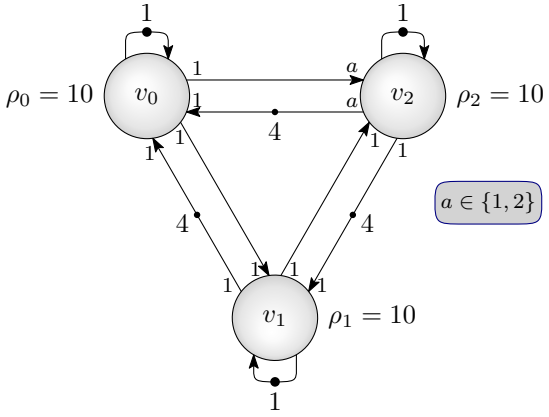


Figure 1.11: HSDF graph corresponding to example 10.

An example can be build and run as follows:

```
cd doc
make ex1 && ./ex1
```

The resulting simulation traces can be viewed with:

```
gtkwave ex1.vcd&
```

The values in the traces are activity counters. For actors that can run auto-concurrently in total three instead of one trace is created per actor. One trace (*ActorName_active*) indicates the number of active firings of the actor and the other traces indicate the consumption times (*ActorName_cons*) and production times (*ActorName_prod*) of tokens.

1.3 Related approaches

A number of related discrete event simulation approaches exist. A number of them are describe below to position the HAPI simulator.

Transaction Level Modeling (TLM) [2] is approach to build simulation models of multiprocessor systems at various levels of abstraction. The SystemC kernel is used for simulation of the system. Higher level of abstractions improve the simulation speed but result in an approximation of the temporal behavior. Building the required TLM models takes usually a significant amount of time. A key difference between HAPI and the typical TLM models are that HAPI simulates dataflow graphs and that the simulation results correspond with the results of analytical models. Such a correspondence between analytical results and simulation results is in general not present for TLM models. Another important difference is that HAPI models are usually abstract in the sense that they do not directly reflect the structure of the system. There is for example no notion of a communication bus in HAPI. As a result of this high level of abstract a relatively low number of events need to be handled by the simulation kernel and simulation is typically fast.

Platform architect is a commercial simulation tool of Synopsys for the evaluation of the performance of several multiprocessor system architecture configurations performance in early design phase. Platform architect is based on the SystemC simulation kernel and has a graphical user interface. Platform architect is a commercial tool for transaction level simulation. Models of communication buses and memory controllers are provided and describe what happens during transactions instead of during each clock cycle. The use of these transactions level models improves the simulation speed. Usually the processors are replaced by traffic generators because the application software is not yet available in the early design phases. Even in case that the application software is available the execution of this software on an instruction set simulator in platform studio is often not practically feasible due to the low simulation speed. HAPI models can describe functional and temporal behavior. For HAPI models it can be guaranteed that the functional behavior is functionally deterministic, i.e., independently of the temporal behavior. A key difference with Platform Studio is that HAPI requires that worst-case execution times of the task can be determined independently of the other tasks in the system. Therefore, HAPI is not suitable to derive the traffic and contention on memory ports as a result of read and write accesses of the processors during the execution the tasks.

Parallel Object Oriented Specification Language (POOSL) [5] is a system-level modeling language based on a small set of language primitives. POOSL enables a precise representation of the system based on a mathematically defined semantics. It consists of a process part and a data part. The process part is based on a real-time probabilistic extension of the process algebra Calculus of Communicating Systems (CCS). The data part is based upon the concepts of traditional sequential object-oriented programming languages like C++. POOSL descriptions are simulated using the discrete event simulator Rotalumis or executed in an interpretative way by the SHESim tool. The underlying formal model of POOSL is the Timed Probabilistic Labeled Transition System (TPLTS). A TPLTS can be transformed into a Markov chain. The equilibrium distribution of a Markov chain can be computed analytically. However the Markov chains obtained after transformation have typically a very large

number of states which results in a prohibitive run-time of the algorithms to compute the equilibrium distribution. As a consequence it is typically only practical to simulate a POOSL descriptions to obtain an estimate the equilibrium distribution. The equilibrium distribution corresponds with long running average performance metrics. A key difference between HAPI and POOSL is that HAPI has dataflow graphs as underlying formal model instead of TPLTS. With dataflow models performance metrics that hold for bounded intervals of time can be computed analytically with computational efficient algorithms. Furthermore worst-case production times can be obtained by simulating a HAPI model. A practical difference is that HAPI has build in support for a number of schedulers while POOSL does not have this.

Y-chart Application Programmers Interface (YAPI) [4] is an event driven simulator for Kahn Process Network (KPN) [3]. A key difference with the HAPI simulator is that the YAPI simulator does not contain a notion of time because KPNs do not have a notion of time. Furthermore the YAPI simulator does not have a way to describe firing conditions because KPNs do not have firing rules. The YAPI simulator does also not support shared resources.

In [1] a simulation approach is described for multiprocessor systems that execute task graphs. The approach assumes that useful worst-case execution times of tasks can be determined in isolation. It should therefore be possible to include the effects of the sharing of memory ports, caches, buses, and SDRAM in the worst-case execution times of the tasks. The task graphs are modeled as functional deterministic dataflow graphs. These dataflow graphs are simulated by the simulator and the worst-case effects of scheduling are computed during simulation and included in the firing durations. With the simulation approach conservative arrival times of the data given the input data stream can be found. This simulation approach requires that only so-called budget schedulers are applied in the system. These budget schedulers guarantee a minimum budget during each replenishment interval. This enables that the worst-case effects of scheduling of tasks can be determined independently of the execution rate and execution times of other tasks. These conservative production times can not be found if for example Fixed Priority Preemptive (FPP) schedulers are applied because then derivation of the worst-case interference of other tasks requires that the critical instance is triggered. The HAPI simulator supports a similar approach as described in [1], however it also supports schedulers that do not belong to the class of budget schedules such as FPP and it supports the evaluation of the actual instead of the worst-case behavior. This is achieved by making schedule decisions during the simulation of the dataflow graph based on the availability of resources and the selected arbitration policy. The HAPI simulator produces reproducible traces, i.e., runing the simulation again results in the same trace.

1.4 List of Symbols

v_i	identifier for actor i
ρ_i	firing duration of actor v_i
τ_i	identifier for task i
C_i	Worst-Case Execution Time (WCET) of task τ_i
S_i	budget of task τ_i when it is scheduled using TDM
Q_i	interval in which the TDM budget of task τ_i is replenished

Bibliography

[1] M. Bekooij, S. Parma, and J. van Meerbergen. Performance guarantees by simulation of process networks. 2005.

[2] A. Donlin. Transaction level modeling: Flows and use models. pages 75–80, 2004.

[3] G. Kahn. The semantics of a simple language for parallel programming. In *Proceedings IFIP Congress*, pages 471–475, 1974.

[4] E.A. Kock, G. Essink, W.J.M. Smits, P. van der Wolf, J.-Y. Brunel, W.M. Kruijtzer, P. Lieverse, and K.A. Vissers. YAPI: Application modeling for signal processing systems. pages 402–405, Los Angeles, June 2000.

[5] B. D. Theelen, O. Florescu, M.C.W. Geilen, J. Huang, P.H.A. van der Putten, and J.P.M. Voeten. Software/hardware engineering with the parallel object-oriented specification language. In *Proceedings of the 5th IEEE/ACM International Conference on Formal Methods and Models for Codesign*, pages 139–148. IEEE Computer Society, 2007.