MULTI-SCHEDULER AND MULTI-THREAD: POSSIBLE WITH SMP2?

Workshop on Simulation for European Space Programmes (SESP)
24-26 March 2015

ESA-ESTEC, Noordwijk, The Netherlands

Charles Lumet[(1)], Nadie Rousse[(2)], Pierre Verhoyen[(1)]

[(1)] SPACEBEL SAS
Technoparc 8 – Rue Jean Bart
31 670 Labège, France)
Email: first.last@spacebel.fr

[(2)] CNES
18 Avenue Édouard Belin
31400 Toulouse, France
Email: nadie.rousse@cnes.fr

ABSTRACT

Discrete event simulators are excellent and efficient for simulating timelines. A problem arises however when several satellite subsystems must be simulated, each of which having for instance its own drifting clock(s). Two fundamental issues then need to be addressed.
Firstly, how can multiple timelines be introduced in discrete event simulators? Obviously, maintaining the advantages of the SMP (Simulation Model Portability) standard is critical as far as portability is concerned. However, these issues being not really addressed by the standard, clever solutions are needed to implement multiple timelines so that no changes are required in the models and that they stay multiple-timeline-agnostic and SMP compliant.
Secondly, one must find out whether these multiple timelines can be taken advantage of in order to better exploit the multi-core processing capability provided by modern mainstream computers, so as to maintain acceptable performance. Again, model agnosticism remains here a major concern.
Multi-scheduler and multi-thread solutions are a subject that is currently in R&D phase at CNES and SPACEBEL. Several interesting trails are already under investigation: the challenge to maintain portable and reusable models is challenging but not impossible.

INTRODUCTION

The SMP2 standard [1] provides a solution for portable models. Some study simulators, such as Argos, need a throughput capacity of several hundreds of thousands of events per second. With modern satellites, simultaneous emulation of several processors and more precise and sophisticated modeling significantly increase the need of processing capability; for instance, a Pléiades satellite has a Central Software, a GPS, and three Star Trackers, meaning 5 emulators to be dealt with in parallel during a simulation.
Such issues can be addressed using multiple timelines and multi-thread. This paper presents solutions implemented by SPACEBEL and CNES to provide multi-thread and multi-scheduler mechanisms guaranteeing genericity and reusability.
This paper is divided as follow: the first section briefly describes the BASILES infrastructure, focusing on key specificities when multiple timelines are involved; the second section explains how multiple schedulers can be used and develops the major advantages of such an approach; finally the third section shows potential solutions as well as promising leads for multi-threaded execution of simulations.

BASILES INFRASTRUCTURE

BASILES is the CNES infrastructure used to design and operate discrete events simulators. BASILES has its own model interface. An adaptation layer allows operating SMP2 models. However, BASILES models play a major part in the execution of a simulation: the BASILES' kernel is entirely made of BASILES models. Since this has major consequences on multi-scheduler and multi-thread mechanisms, we develop in this section the main reasons for that choice.

Configurability through Models

Configuration needs are the main reason why BASILES' kernel is designed with models. Using models means that the configuration becomes as rich as specifying models instances to be created and links between them: BASILES' kernel is thus highly configurable.
This configuration is set via two mechanisms. Firstly, as in SMP2 with the *Catalog*, *Configuration* and *Assembly* files, some dedicated BASILES files are used to instantiate and configure the kernel models. But there is another way of setting the configuration (and more generally interacting with the simulation): BASILES comes with a command interpreter. Establishing a connection between two models is then as simple as sending a command to the interpreter. From this comes the ability to easily reconfigure the kernel at initialization or during a simulation according to the simulation needs. The added advantage is that it facilitates significantly fault injection, test procedures and introspection.

Data Propagation

In SMP2, for an input Field to be updated with the connected output Field value, one needs to operate some sort of data propagation. Interfaces could be used to overcome this issue, but they have the drawback of highly restricting the introspection possibilities and error injection.
In BASILES models, on the contrary, an input and a connected output, automatically share one unique value: no specific data propagation is required. Consequently, inputs can be simply implemented as pointers, whose values are set by the infrastructure upon connection. Such a mechanism allows for high performance, even when numerous models and thus numerous connections are involved in the simulation. BASILES models are therefore a perfect fit for a use in the infrastructure kernel, where performance and introspection capabilities are major concerns.

Events in BASILES

In BASILES, each event has to be "published" once by its model to the infrastructure before being executed for the first time. With such a mechanism, comes the ability to store contextual data for each event. This somehow acts as an SMP2 *Schedule* file, but it extends the concept to all events and broadens the spectrum of event-related data that can be stored. Among this data, key information can be found, among which the priority defined for the event, the scheduler in which it should be posted, and the drift of the event (if any).

INTRODUCING MULTIPLE SCHEDULERS

One main issue arises when one has to simulate complex satellites: several processors, operating at the same time, have to be simulated together. This means being able to firstly integrate one or more processor emulators and thus their local schedulers, if any, into BASILES, and secondly execute them together. Let us first explain how BASILES is capable of operating different types of scheduler.

Types of Schedulers in BASILES

Different types of simulation may require different types of schedulers. This is achieved in BASILES by the interface BslSchedulerBase that any scheduler must implement as in Fig. 1. This interface includes typical scheduling functions, and allows the implementations to be declared as schedulers to the BASILES' kernel. Note that this interface is naturally also available in an SMP2 context (see Fig. 1).
Each type of scheduler (and possibly several at the same time, see below) can be instantiated by the kernel and used for a simulation.

Scheduler Types and Performance

This possibility to design and use different types of schedulers is critical as far as performance is concerned, and also because no single scheduler implementation can be optimal for all scheduling schemes. The complexity for using – inserting and removing events from – a scheduler mainly depends on the used data structure to store events. BASILES comes natively with two types of scheduler whose design is based on that observation: the first one is built on a **map** data structure based on a weight-balanced binary tree) and the second one on a **list** (data structure based on a doubly linked list).
In BASILES, events are ranked by increasing execution date; two events sharing the same date but with different priorities will be ranked according to their priority; finally, if two events have the same date and priority, they will be executed in the same order they have been posted. In the case of the "list" scheduler, all events are stored in a doubly linked list, according to their execution order. In the case of the "map" scheduler, the events belong to the leaves of the weight-balanced tree; if two events share the same date and same priority, they belong to the same leaf, which is itself a list (see Fig 2).
The complexity of insertion and removal of an element from these data structures implies the following: the "list" scheduler is to be preferred most of the time; however, when the number of events in the scheduler become increasingly large throughout a simulation, the "map" scheduler offers better performance, which is moreover independent of the event characteristics (cyclic/acyclic, etc.).
This "map" scheduler is used with great benefit on ARGOS simulator. ARGOS is a worldwide beacon-based tracking and environmental monitoring system. A lot of beacons have to be simulated simultaneously, which leads to an unusually high number of events (data emissions, receptions, etc.) in the scheduler at any moment. A typical case involves around 40 000 beacons, and leads to roughly as many events in the scheduler at any given time; in such a use case, the "map" scheduler processes around 60 times faster than the "list" scheduler.

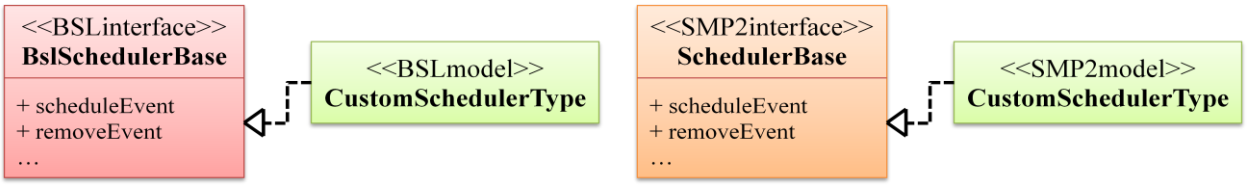

Fig. 1. Multi-scheduler UML diagram

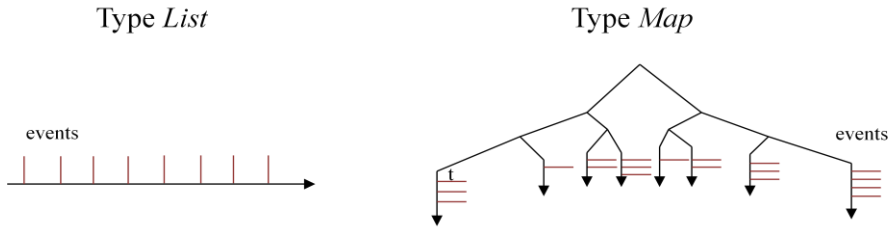

Fig. 2. Storage of events in a list-type scheduler

Multi-scheduler mechanism (see following sections) allows complex configurations by enabling the simultaneous use of different schedulers, and each one can be optimized for the relevant posting scheme.
Therefore, the “list” and “map” schedulers can be used together for some simulator to obtain optimal performance. For instance, in a simulation where model A only posts cyclic events of the same frequency, whereas model B posts acyclic events at random dates, then it would be efficient to instantiate one scheduler of each type, posting events of model A to the “list” scheduler and events of model B to the “map” one.

Integrating Flight Software

The execution of a processor is simulated via an emulator, which usually comes as a proprietary software library: it usually also includes its own scheduler. The events belonging to models including flight software are intended to be posted either in the main simulation scheduler or in the emulator scheduler: the choice is usually made when designing the simulator, depending on the event interactions and how representative the simulation needs to be.
The integration of an emulator in BASILES is a simple encapsulation process (see Fig. 3): the emulator library is represented as two SMP models. The first one encapsulates the scheduler part of the emulator, the second the emulator other main functionalities. The scheduler model depends on a kernel service “RegisterScheduler” so that it can register itself to the kernel (more precisely the Monitor module) as a scheduler. This is possible since it is also implementing the BslSchedulerBase interface; this implementation finally allows the scheduler to be operated by the monitor through the BslBslSchedulerBase interface and the generic ExternalScheduler SMP2 model.

Multiple schedulers in action

In BASILES, event management is centralized: all events are posted to the Monitor (a kernel module), which then acts as a switch to direct each event to the appropriate scheduler. The information regarding the scheduler an event should be posted in is set from the simulation configuration script (which acts here as an extended SMP2 *Schedule* file). One key feature of this mechanism is that the models stay entirely scheduler-agnostic: they do not have to depend nor rely in any way on the number nor types of schedulers used by the simulation (see Fig. 4).
As exposed in the previous sections, all schedulers implementing BslSchedulerBase interface can be registered in the BASILES’ kernel, and each scheduler is a BASILES model (both BASILES default schedulers and external types of scheduler via an encapsulation process).
The synchronization of the execution of several schedulers is then simply done by adding connections between those “scheduler” models. For instance, Fig. 5 shows a possible configuration with two schedulers. Such a configuration synchronizes the execution of both schedulers as long as there is no event cross-posting (i.e. events post other events in their own scheduler only). More complex connections allows for complete synchronization, with no hypothesis whatsoever on event posting.

TOWARD MULTI-THREADED SIMULATIONS

Parallelizing the execution of a simulator can take numerous forms. We propose to distinguish here between three different approaches, based on the “level” at which the parallelization occurs.

Function-level Multi-threading

This is the lowest level of parallelization and this might be the most obvious and direct way of using multi-thread in a simulation. When one designs a model, one can stumble upon a tedious time-consuming computation. It may be the case that such a computation is separable in some way, and several threads could be used to increase performance, for instance through the use of parallel libraries and tools. This use of multi-thread is entirely localized inside the model and should be transparent for the model user (see Fig. 6).
We do not go into more detail about this approach since it does not raise any major challenges as far as the simulation infrastructure is concerned.

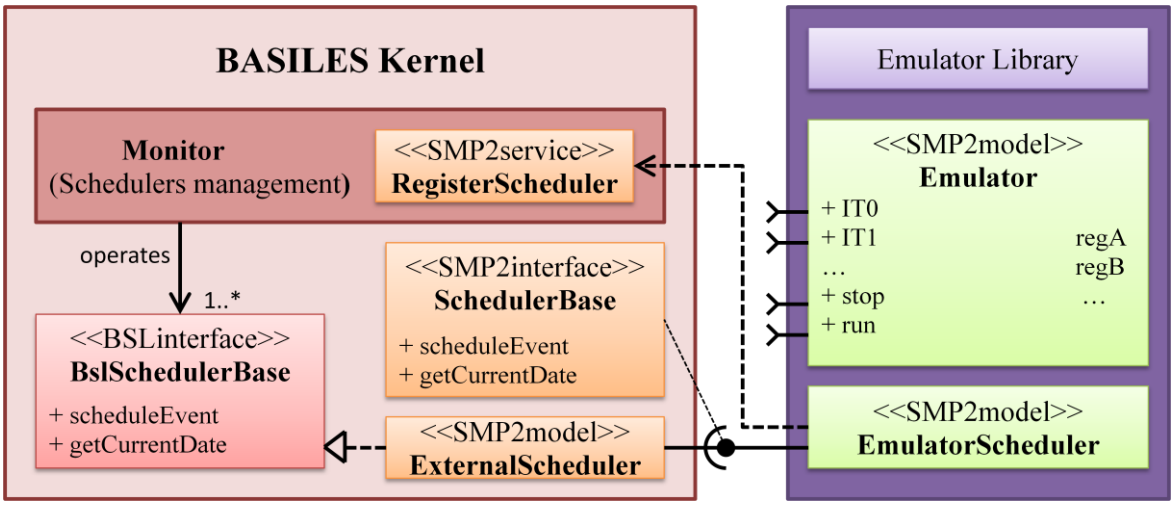

Fig. 3. Integration of an emulator in BASILES

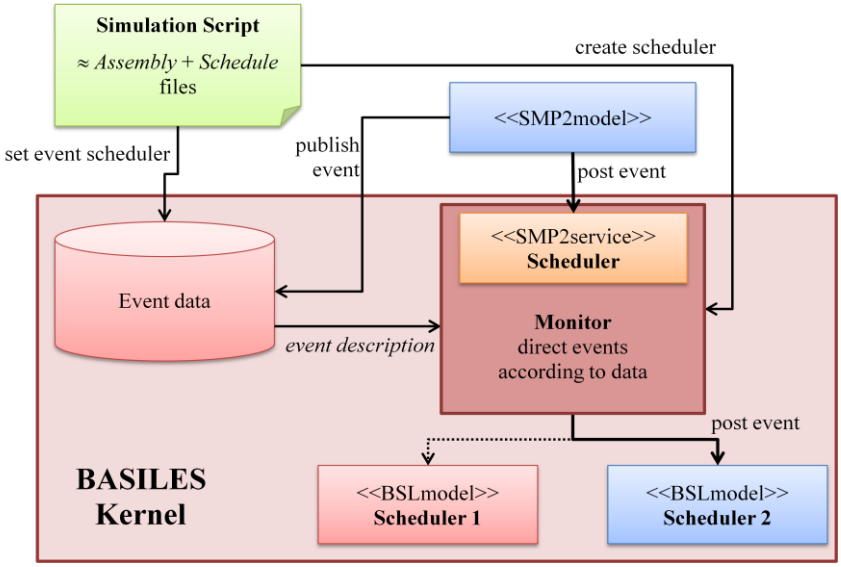

Fig. 4. Functional diagram of BASILES' kernel event management in a multi-scheduler context

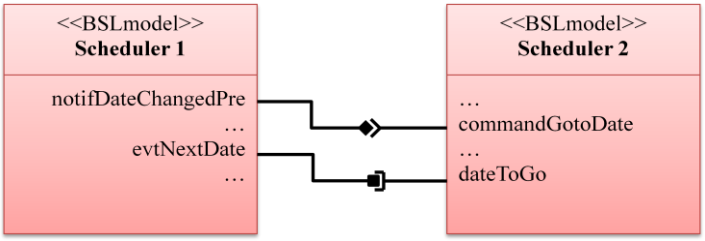

Fig. 5. Diagram of the connections between two schedulers.

Event-level Multi-threading

Noticing that events with same execution date and same priority are unordered, one might try to parallelize their execution so that they can be processed simultaneously. As detailed above, BASILES allows for the creation of new types of schedulers. We take advantage of this capability to design a dedicated scheduler whose goal is to parallelize executions of unordered events. Each time this scheduler has to execute more than one event with the same date and priority, it creates one or more new threads to execute these events. It then waits for all events to be executed and falls back to standard mono-thread execution, until a new set of unordered events is found (see Fig. 7). Obviously, such parallel execution works only for events that have no data interaction or causal dependencies. Note that with this mechanism, a same event will likely be executed by different threads during the simulation.

Such an approach is highly efficient when the execution time of parallelized events is high compared to the overhead of thread creation; on the contrary, performance will not be improved and can even be decreased if used in cases where sets of unordered events contain mostly short-execution events.

Event-level multi-threading is really easy to use: it does not depend nor rely on model implementation and can apply to almost any case (with more or less benefit, as exposed above). However, this approach has a significant drawback: parallelization in event-level multi-threading only happens based on basic events' data (i.e. date and priority). High-level separability (for instance natural separability between two models, coming from functional observations, see [2]) is totally concealed and cannot benefit from using event-level multi-threading.

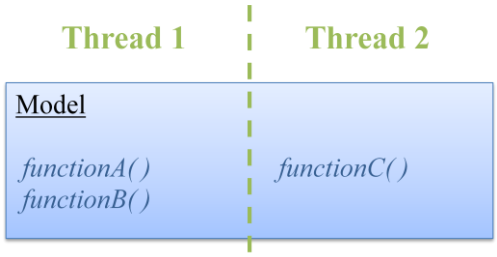

Fig. 6. Model-level multi-thread approach

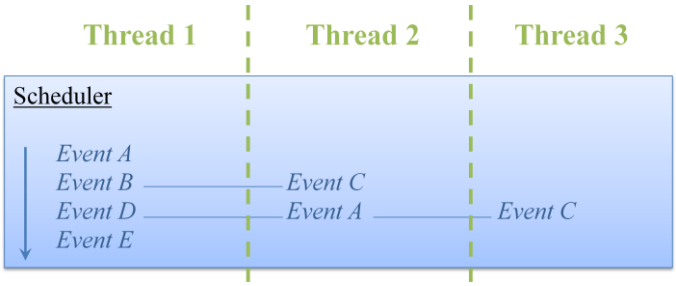

Fig. 7. Event-level multi-thread approach

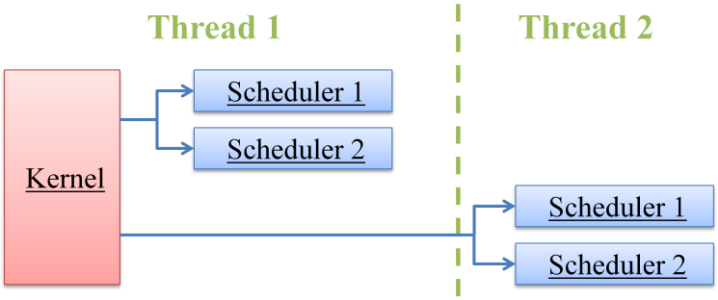

Fig. 8. Scheduler-level multi-thread approach

Scheduler-level Multi-threading

The scheduler-level multi-threading approach directly comes from the multi-scheduler concept. The latter allows to distinguish between events, and to direct each of them into a given scheduler when posted, according to a preset configuration. Since in a discrete events simulator, all action is done throughout events, being able to decide in which scheduler each event will be posted really gives the user high-level control of the simulation; parallelization should then occur from such a point of view, where one can identify separable parts in the simulator: this is exactly what the scheduler-level multi-threading achieves. This approach is a necessity for accelerating systems that have to simulate multiple flight software running on several processor emulators.
In this approach, each scheduler exists inside a given thread, according to the user configuration; this means that, if a scheduler is "moved to" a secondary thread, all events posted in this scheduler will be executed inside that secondary thread (see Fig. 8). The scheduler-level multi-threading approach concentrates all multi-thread problematic in one single place: models are scheduler agnostic, and also remain independent to the thread and synchronization issues.
Technically speaking, "moving" a scheduler out of the main thread is managed by the kernel that deals with thread management and synchronization. While the approach is simple in principle, several potential issues need deeper investigation, such as guaranteed causality and temporal coherence of variables, save and restore, single stepping of flight software or events, model and simulator introspection... Another important issue is that the ability for the user, when something goes wrong with a model or its external interface, to maintain control, visibility and introspection capability. All those aspects constitute main challenges; we have implemented promising solutions to address them but they remain very much a work in progress.

CONCLUSION

The increasing complexity of satellite systems requires the development of new simulation mechanisms, so as to maintain acceptable performance. One particular issue that has to be dealt with is the more and more common presence of several processors inside satellites, all of which needed to be simulated together through emulators. This leads to two major challenges: firstly, being able to import into a simulation framework several emulators, each with its own specific scheduler; and secondly, being capable of executing all these emulators and schedulers together in a simulation.
In the context of BASILES, we have designed a dedicated interface to manage import of any scheduler of any type to represent a timeline. This allows using any emulator into a simulation, and also has the added benefit of decoupling the emulator scheduler interface from the emulator core. BASILES' scheduler interface can also be used with profit to design custom schedulers, optimized to deal with specific posting schemes: two different types of schedulers are already included in BASILES, and their use greatly improves performance on some specific simulators.
BASILES' kernel is designed to deal with several schedulers at the same time in a simulation. One key feature of the mechanisms at stake is the ability for the models to remain scheduler-agnostic: information regarding the repartition of events in schedulers is managed globally, through a simulation configuration script, and saved by BASILES kernel so that posted events are directed in the right scheduler without any model action.
Multi-threading can greatly benefit from this multi-scheduler approach: each scheduler can be affected to a thread. In this way, the multithread configuration can be handled by the user at a high level, easily taking advantage of models functional separability. Again, a major aspect of this mechanism is that, from the above-mentioned scheduler-agnosticism, naturally comes thread-agnosticism. With this approach, main challenges remain standard ones, such as thread synchronization and causality guarantees.

REFERENCES

[1] ECSS, Collaboration website of the European Cooperation for Space Standardization (accessed February 26, 2015) - www.ecss.nl/forums/ecss/dispatch.cgi/publications/folderFrame/100127/0/def/9188

[2] F. Quartier, P. Verhoyen, N. Rousse, & F. Manon (2014, October). Mainstream Components for Near Hard Real-Time Distributed Simulation and Testing. In Proceedings of the 2014 IEEE/ACM 18th International Symposium on Distributed Simulation and Real Time Applications (pp. 11-17). IEEE Computer Society.