Efficient Implementation of Native Software Simulation for MPSoC

Patrice Gerin, Xavier Guérin, Frédéric Pétrot

System-Level Synthesis Group
TIMA Laboratory
46, Av Félix Viallet, 38031 Grenoble, France

Abstract

Efficient and precise simulation models at a high abstraction level are required in order to perform early design validations and architecture explorations of Multi-Processor System-On-Chip (MPSoC) platforms. Although native software simulation approaches provide interesting capabilities, they quickly become unsuitable when complex hardware architecture have to be considered.

In this paper, we present a SystemC-based MPSoC platform implementation that allows native software simulation while keeping details of the underlying hardware model. The key contribution of this work is a realistic memory mapping modelling that makes possible the simulation of Operating Systems and software applications on complex hardware models with multiple processors and DMA devices. This method also allows the reuse of different software components for the target processor(s). Experimental results show the efficiency of the proposed method to validate software on complex hardware architectures.

1. Introduction

Multi-Processor System-On-Chip (MPSoC) architectures face important constraints such as performance or power consumption. Unfortunately, the increasing complexity of these systems (in terms of number and heterogeneity of embedded processors) conflicts with short time-to-market.

Besides, the part of embedded processors in such systems becomes too important to fully take advantage of their programmability. In this context efficient and precise simulation platforms are needed to allow the early validation of the application and perform architecture explorations.

MPSoC architectures considered in this work are made of hardware and software nodes connected to a communication network (Fig. 1). A hardware node is a component that does not provide any programming capabilities. A communication network connects all the nodes together and provides the so called inter-communication of the MPSoC.

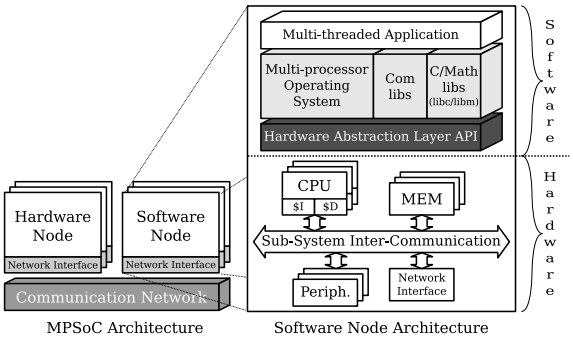

Figure 1. MPSoC and SW Node Architecture

Software nodes on which we focus in this paper provide the execution environment for the software tasks of MPSoC applications. The software is organised in layers which are the multi-threaded application layer, the Operating System (OS) and libraries layer (that supports the application execution) and the Hardware Abstraction Layer (HAL) that hides the hardware specificities. The hardware part of a software node called processor sub-system contains one or more processors with their peripherals (memories, interrupt controller, DMA ...).

As most of the design time is spent in software validation, fast and precise simulation platforms are required. Current methods to achieve simulation performance and flexibility are based on *native* execution of the software. A native simulation approach consists of compiling and executing the software code on the host machine processor instead of using Instruction Set Simulator (ISS) of the target processor to interpret the cross-compiled source code.

Our objective is a MPSoC simulation platform able to model different architectures and on which applications and Operating Systems can be executed natively while keeping details of the underlying hardware. As the source code is compiled for the host machine's processor, there is no consideration of a target hardware architecture platform.

The contribution of this paper is a methodology to im-

978-3-9810801-3-1/DATE08 © 2008 EDAA

plement SystemC MPSoC platforms model with memory mapping and hardware considerations for native software simulations. This approach allows a complete portability of the software components (OS, application and libraries) to the target platform.

The rest of this paper is organized as follows: Section 2 gives a review of related works. Section 3 presents the basic concepts used by our approach. Section 4 gives key ideas for native simulation and technical details of the implementation. Experiments in section 5 shows the interest of our approach for MPSoC simulation through a Motion-JPEG application.

2. Related Works

In the past few years, different frameworks have been proposed for System Level Design. [5] [9] or [14] are some of them. Most of the recently proposed approaches are based on native simulation to achieve high simulation performances and flexibility. However, native approaches are suitable only for high level simulation and show their limits if the underlying hardware have to be considered with more details. A typical example is the memory accesses from native software for which the host machine's simulator has no control. This problem is addressed in [9] ; Their framework solves this by the source code instrumentation which allows to catch and remap memory accesses. In our approach, we use annotations for time performance modelling but a non-annotated one can also be executed on the simulation environment, keeping the interaction with the underlying hardware. Considering the lack of Operating System model for the System Level Design raised in [11] and [7], different propositions have been presented in [4, 7, 8, 11, 13]. In these approaches, native simulation of software tasks is made over an abstract model of the Operating System. Compared to these approaches that introduce the final Operating System only at low level Instruction Set Simulators (ISS) based platforms, we propose, in this paper a hardware executable model able to handle a real Operating System very early in the design flow. Thus, an important part of the Operating System can be validated on such native platform.

The native simulation of a real Operating System has also been proposed in [3] but hardware resources like memories where considered local to the processors. This restriction is not suitable for Multi-Processors architectures where shared resources (memories, synchronization resources ...) become significant.

Finally, in most current approaches based on abstract hardware models, the interaction between the native software and the hardware model is generally kept implicit. We propose to fill this lack by allowing SystemC [2] hardware platform implementation providing detailed hardware view for native software execution.

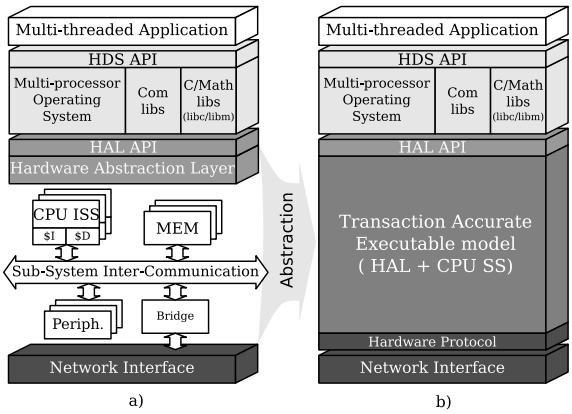

Figure 2. a) VP and b) TA levels

3. Basic Concepts

This work takes place in a MPSoC design flow composed of three abstraction levels : A low level implementation of MPSoC architecture called Virtual Prototype (Fig. 2.a) provides detailed and time accurate simulations at the expense of the simulation speed. At the opposite side, a System Level executable model allows very fast simulation speed but with very low timing accuracy. In this paper we focus on an intermediate abstract level called Transaction Accurate (Fig. 2.b).

The Transaction Accurate model represented in Fig. 2.b abstracts the processor sub-system of the software node and the Hardware Abstraction Layer, providing a HAL API to the software part of the application and a hardware interface to the other.

3.1 Hardware Abstraction Layer API

A Hardware Abstraction Layer API is a set of functions that allows software to interact with hardware devices at an abstract level rather than at a detailed hardware level. This abstraction layer hides the details of the physical hardware to the software that runs on it. HAL is especially important in our approach for early design of portable Operating System for different hardware platforms. This portability remains valid for the Transaction Accurate model since the HAL is completely hermetic. Our HAL API is similar to the eCos Operating System HAL API [1] which is well defined and provides a strict separation from the upper software layers needed by our approach. Each of the HAL API functions is defined as C macros and can be implemented by C functions or assembly code as needed. We identified two classes of APIs to provide a suitable HAL interface. These classes are processor and platform dependent.

3.1.1 Processor HAL API

The processor HAL API abstracts the specificities of the processor. Typical APIs for context management are `HAL_CONTEXT_[INIT|LOAD|SWITCH|SAVE]`, for low level interrupt management `HAL_IT_[MASK|UNMASK]` or `HAL_CPU_TO_PLATFORM` and `HAL_PLATFORM_TO_CPU` for endianess. `HAL_SPIN_[LOCK|UNLOCK]` provides low-level synchronization API which is essential for multiprocessor support.

3.1.2 Platform HAL API

In an heterogeneous architecture, processors can have different endianess. A reference endianess must be defined to allow communication between these processors. The reference endianess is attached to the platform and can be known with `PLATFORM_IS_[BIG|LITTLE]_ENDIAN`.

The platform HAL API also provides the abstraction of the platform memory mapping which is unusual in HAL. A classical way to define memory mapping of platform devices is very often hard coded addresses which is not suitable in native simulation. Therefore `PLATFORM_GET_ADDRESS(symbol)` HAL API has to be used to get the address of a *symbol* defined by the platform. This point is detailed in section 4.

3.2 Transaction Accurate Level

The executable model of the Transaction Accurate level is written in SystemC as proposed in [6], defining a unified methodology to implement both hardware/software components. Using this approach, we are able to define a model which provides a SystemC view of a software API on one side and a hardware interface on the other. At Transaction Accurate level, the HAL API is provided through SystemC software ports as depicted in Fig. 3 (1). The implementation of this API is made by SystemC modules which are interconnected through their ports.

The HAL API supports upper software layers execution. The C source code of these layers (composed of the Operating System, the libraries and the application) are compiled for the host machine and encapsulated in a dynamic library as depicted in the Fig. 3 (2). As SystemC is a C++ library, a wrapper (3) is needed to adapt the C HAL API view of the software dynamic library. At initialization time, this wrapper will act as a program loader by linking the dynamic library to the simulation executable.

Functions provided by the embedded software in the dynamic library can be called through the wrapper. This is typically the case of the Operating System *BOOT* function. From the application point of view, the functions provided by the API can be called as usual. The fact that they are not really implemented (in C or assembly) but emulated within SystemC is completly transparent. The wrapper will convert these C calls to SystemC port accesses (C++ method calls) and transfer execution to the connected module. Thus, the implementation of the C HAL macros at Transaction Accurate level consists only in function calls which are handled by the wrapper and redirected to the SystemC simulation model.

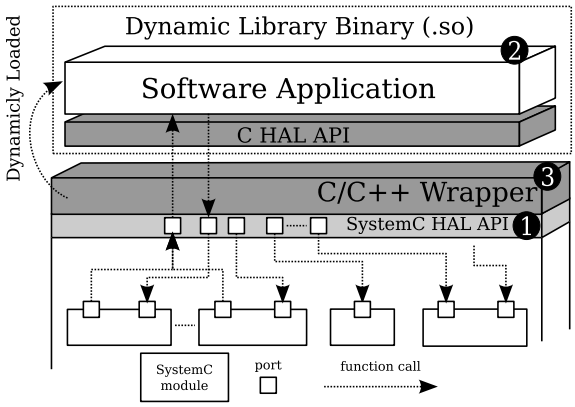

Figure 3. Transaction Accurate software API

The following source code shows an implementation of the *HAL_SPIN_LOCK* API for a SPARC processor and for native simulation.

```
// SPARC implementation
#define HAL_SPIN_LOCK(spin)                                    \
{                                                              \
  register uint32_t res;                                       \
  do {                                                         \
    __asm__ ("lda [%1]0x20,%0 ":"=r"(res):"r"(spin)); \
  } while (res != 0);                                          \
}

// NATIVE implementation
#define HAL_SPIN_LOCK(spin) \
  __wrapper_hal_spin_lock(spin)
```

4. Native Simulation for MPSoC

4.1 Key Ideas

The first key idea in this work is to keep the low level hardware details for software native simulation, such as shared resources between processors (e.g. variables in memory or synchronization mechanisms).

The following draws up a list of essential issues to model an efficient and accurate native software simulation at Transaction Accurate level:

Memory representation: As mentioned in the related works section, memories are usually considered private for each processor. This simplification is not suitable in multiprocessors architectures or more generally in architectures where multiple masters can access the same memory space.

Software execution: The Transaction Accurate model of the platform should be able to model multiple executions of

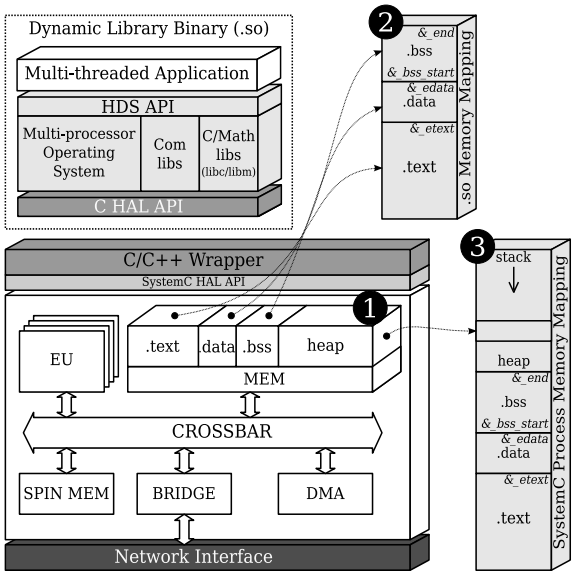

Figure 4. Simplified SW Node Architecture

the same software binary code, which is the basis of Symmetric Multi Processor-like architectures.

Synchronization: The support for a *spinlock* mechanism in a multi-processor architecture is necessary to guarantee the access to shared resources.

4.2 Memory Representation

Fig. 4 depicts a Transaction Accurate executable model using SystemC. In this figure, three memory mappings are represented :

1. The memory mapping of the processor sub-system (a CROSSBAR in this example). At Virtual Prototype level, this memory mapping is specified by the architecture designers.

2. The memory mapping obtained by the software compilation for the host machine (Application + OS + Libs). This mapping is host machine dependent and defined at the dynamic library creation.

3. The memory mapping seen by the SystemC executable process which handles the simulation of the designed architecture. This mapping is also host machine dependent.

Host machine memory representations (2) and (3) are classical mappings of executable software. The mapping can be summarized to three memory segments which are *.text* containing the binary code, *.data* for initialized program data and *.bss* for data to be zeroed.

The main difficulty is to manage two memory mapping point of views which are the host machine dependent and the user defined platform mappings (dynamic library and executable can be considered as one mapping). The solution which consists in using the target memory mapping in the sub-system model and using remapping techniques in the memory component is not suitable because of total overlappings between these two memory spaces. The value of an address cannot be used to determine which memory space is concerned.

The solution is to perform the memory mapping at execution time. The mapping addresses given to the *CROSSBAR* component correspond to valid addresses in the SystemC process memory space. This explains the need of a HAL API to get the addresses of platform devices.

Hardware devices must implement their internal registers in contiguous memory (e.g. array of integer). The memory mapping is built during the initialization phase of the simulation by requesting the start address and the size of the register array of each components. The memory mapping consistency is ensured by the host machine. Memory components that provide a memory address space are implemented using the same approach. In Fig. 4, the *heap* memory of the platform is allocated in the simulation executable address space.

The *.bss* and *.data* sections of the software application have to be accessible by hardware components (e.g. DMA devices). During native compilation of software code, symbols identifying the beginning and the end of the *.bss* and *.data* sections are automatically created by the compiler (e.g. *_bss_start* and *_end* for *.bss* section). These symbols are used by the *MEMORY* component and provided to the platform mapping to make these sections accessible through the *CROSSBAR*.

4.3 Software Execution

Software execution is supported by a SystemC module of the Transaction Accurate model called Execution Unit (EU) in Fig 5. This EU which acts as an abstract processor is connected to a *boot* port of the SystemC HAL API from where the entry point function provided in the dynamic library can be called (1). All the software is then executed sequentially (2) within the caller EU module context.

Since software code is executed in SystemC without time consideration, the application dynamic library has to be annotated in order to “consume” SystemC simulation time. Ongoing work on this subject aims at providing a transparent solution to insert annotations at compile time, but the proposed TA model does not depend on the annotation technique. The annotation approach used in [9], consisting in inserting C lines in an intermediate source code, as well as the approach based on “assembly level” C code [10] are adapted to our simulation model.

Although software annotation is a key component of our

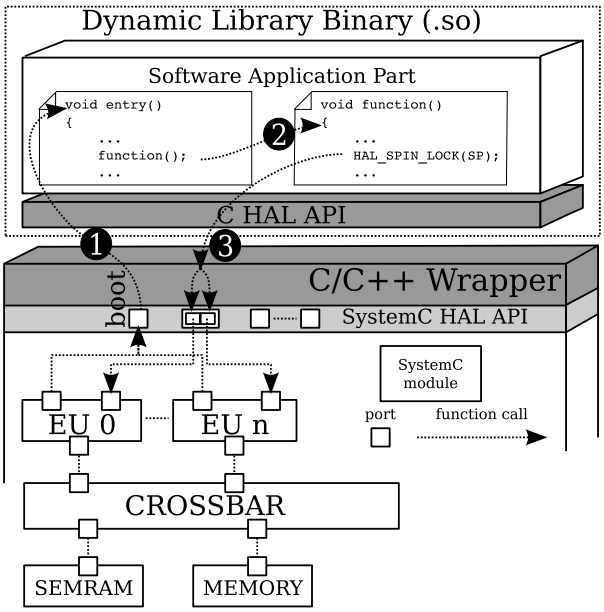

Figure 5. Software Execution on TA Model

approach, this is not mandatory and non annotated software can still be simulated on the proposed simulation platform (as if the target processor had infinite frequency). Special care should be taken to avoid simulation deadlock (for example in case of an active waiting infinite loop in the software code).

Several EU can be connected to the same *boot* port of the HAL API, which give to our simulation platform the Symmetric Multi-Processor capability. As the software application has no notion of the SystemC module on which it execute, the wrapper must redirect the C HAL API call to the correct port of the SystemC HAL API (3). To do this, we use the *sc_get_curr_simcontext()* SystemC function that handles different information like the current active module which can be used to identify the corresponding port.

4.4 Synchronization

As the platform provides a real processor parallelism environment, low level synchronization must be provided to protect shared resources accesses as well as for Virtual Prototype platforms. Since the `HAL_SPIN_[LOCK|UNLOCK]` API are processors specifics, they will be implemented by the Execution Unit SystemC module. The implementation will depend on the processor and communication we want to model. If the processor and the communication components do not support the *test-and-set* mechanism, additional dedicated components must be provided in the hardware platform model (e.g. the *SEMRAM* hardware semaphore in Fig. 5). As explained in section 3.2, the *HAL_SPIN_LOCK(spin)* HAL macro is defined as a *__hal_spin_lock(spin)* C function call. When caught by the C/C++ wrapper, this call is redirected to the correct Execution Unit through the corresponding SystemC HAL API port. The EU implementation of the spinlock is given below.

```
void EU::__hal_spin_lock(spin)                    \
{                                                 \
  register uint32_t res;                          \
  do {                                            \
    crossbar.read(UINT32,spin,res);               \
  } while (res != 0);                             \
}
```

If *test-and-set* are supported by the hardware, EU will used protected access provided by the communication module to *test-and-set* the semaphore directly in standard memory.

In these sections, we have presented the three key ideas that give our platforms the capacity to accurately model MPSoC architecture in terms of processors parallelism and memory representation. Interrupts have not been addressed in this paper but are already supported by our simulation platform and have been presented in [3].

5. Motion-JPEG Case Study

5.1 Hardware Architecture

The processor sub-system of the software node (Fig. 6) is a generic UMA (Universal Memory Access) platform that embeds a configurable number of processors, a global memory, a hardware semaphore RAM (SEMRAM) and some hardware terminals (TTY). Externals components are the traffic generator (TG) that takes its data from a MJPEG movie file and the viewer (RAMDAC).

The software node sub-system is implemented at the Transaction Accurate level according to the approach presented in this paper.

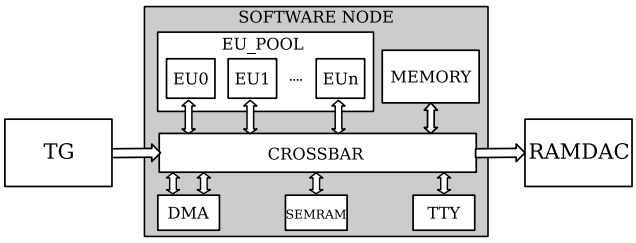

Figure 6. M-JPEG Hardware Platform

5.2 Software Architecture

Motion JPEG (M-JPEG) is a multimedia format where each video frame of a digital video sequence is separately

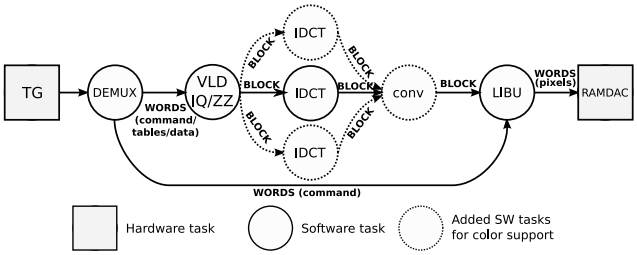

Figure 7. Functional Model of Motion-JPEG

compressed as a JPEG image. The MJPEG decoder application defines five initial software tasks and two hardware tasks (additionnal tasks are created for color images). Each software task is a POSIX thread and is part of a process network that reads the data stream from the TG and sends the uncompressed images to the RAMDAC viewer.

This application is executed on a POSIX compliant Operating System that supports SMP architectures called Mutek [12]. This OS come with a light C library. Maths and communication libraries have been added.

5.3 Approach Analysis

We first used this platform to validate evolutions implemented in the Mutek Operating System. Since the software is compiled natively, we naturally take advantage of the debugging tools available on the host machine. Combined with the Transaction Accurate model, we were able to validate complex synchronization mechanisms which are not possible in classical native software simulations due to the lack of hardware consideration. With low level ISS based simulation environments, Operating System validation quickly becomes a huge task.

The effective hardware/software interaction is illustrated here through the use of a DMA device to move data from the heap to the RAMDAC. SystemC waveforms can be used to display mixed hardware and software information. In Fig. 8, a blocking communication is presented when no DMA is used (1). With DMA, the LIBU thread starts the DMA (2) which performs the transfer (3) and another thread is activated (4).

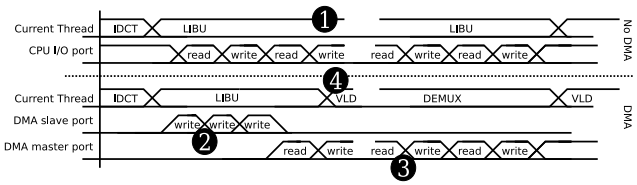

Figure 8. DMA transfert at TA level

This DMA example gives a good overview of the TA platform to closely model hardware and software interactions capabilities.

6. Conclusion

We proposed in this paper a SystemC-based implementation methodology for MPSoC platforms that allows native simulation of the software with detailed hardware models. This platform improves flexibility and simulation speed needed for architecture exploration. The main shortcoming of this solution is the need of software annotation to obtain timing analysis. That being said, the approach does not depend on the quality of this annotation. Future works will consist of providing precise annotation techniques adapted to multiprocessor architectures and then providing automation tools to assist architecture exploration on different kinds of hardware platforms.

References

[1] ecos homepage, http://ecos.sourceware.org/.
[2] Systemc homepage, http://www.systemc.org/.
[3] A. Bouchhima, I. Bacivarov, W. Youssef, M. Bonaciu, and A. A. Jerraya. Using abstract cpu subsystem simulation model for high level hw/sw architecture exploration. In *ASP-DAC, 2005*.
[4] A. Bouchhima, S. Yoo, and A. Jerraya. Fast and accurate timed execution of high level embedded software using hw/sw interface simulation model. In *ASP-DAC, 2004*.
[5] D. Gajski, J. Zhu, R. Dömer, A. Gerstlauer, and S. Zhao. *SpecC: Specification Language and Methodologie*. Kluwer Academic Publishers, Boston, March 2000.
[6] P. Gerin and al. Flexible and executable hardware/software interface modeling for multiprocessor soc design using systemc. In *ASP-DAC, 2007*.
[7] A. Gerstlauer, H. Yu, and D. D. Gajski. Rtos modeling for system level design. In *DATE, 2003*.
[8] M. A. Hassan, K. Sakanushi, Y. Takeuchi, and M. Imai. Rtk-spec tron: A simulation model of an itron based rtos kernel in systemc. In *DATE, 2005*.
[9] T. Kempf, K. Karuri, S. Wallentowitz, G. Ascheid, R. Leupers, and H. Meyr. A sw performance estimation framework for early system-level-design using fine-grained instrumentation. In *DATE, 2006*.
[10] M. Lajolo, M. Lazarescu, and A. Sangiovanni-Vincentelli. A compilation-based software estimation scheme for hardware/software co-simulation. In *CODES'99*.
[11] R. L. Moigne, O. Pasquier, and J.-P. Calvez. A generic rtos model for real-time systems simulation with systemc. In *DATE, 2004*.
[12] F. Petrot and P. Gomez. Lightweight implementation of the posix threads api for an on-chip mips multiprocessor with vci interconnect. In *DATE, 2003*.
[13] H. Posadas, J. Ádamez, P. Sánchez, E. Villar, and F. Blasco. Posix modeling in systemc. In *ASP-DAC, 2006*.
[14] W. Tibboel, V. Reyes, M. Klompstra, and D. Alders. System-level design flow based on a functional reference for hw and sw. In *DAC, 2007*.