Building Fast and Accurate SW Simulation Models Based on Hardware Abstraction Layer and Simulation Environment Abstraction Layer

Sungjoo Yoo Iuliana Bacivarov Aimen Bouchhima Yanick Paviot Ahmed A. Jerraya
System-Level Synthesis Group
TIMA Laboratory
Grenoble, France

Abstract

As a fast and accurate SW simulation model, we present a model called **fast timed SW model**. The model enables fast simulation by native execution of application SW and OS. It gives simulation accuracy by timed SW and HW simulation. When building fast timed SW models, we need to solve two problems: (1) how to enable timing synchronization between the native execution and HW simulation and (2) how to obtain the portability of native execution (that needs multi-tasking from simulation environments to emulate its multi-tasking operation) on different simulation environments (that give different types of multi-tasking). In this paper, to enable the synchronization, we present a synchronization function. To enable the portability, we present an adaptation layer called simulation environment abstraction layer. We present our case studies in building fast timed SW models.

1. Introduction

Since SW complexity grows rapidly in SoC designs, to achieve short time-to-market, SW validation needs to be fast and accurate. Conventionally, two methods of SW validation have been used in embedded systems design domain and in classical software design domain: ISS (instruction set simulator) execution and native execution of application SW and OS.

Figure 1 exemplifies the two methods. Figure 1 (a) shows a processor and the other HW of the system. The other HW includes processor local bus, peripherals, on-chip communication network, other processors and HW IPs connected to the network, etc. Figure 1 (b) shows classical ISS execution in timed HW/SW cosimulation. In this method, an ISS and a BFM (bus functional model) replaces the processor of Figure 1 (a). The BFM works as a cosimulation interface between the ISS and the other HW (at RTL) of the system. The BFM provides the ISS with two kinds of function: interrupt check and read/write operation. Although the advantage of this method is accuracy (at instruction/cycle/phase-accuracy), its main drawback is the slow simulation speed.

Figure 1 (c) shows the classical native execution of application SW and OS. In this method, application SW and OS are targeted on a simulation host OS, e.g. Unix. We call the classical native execution of OS *native OS*. An example of commercial native OS is VxSim™ of VxWorks [1].

The advantage of classical native execution method is simulation speed since the application SW and OS are not interpreted by a simulator, but executed natively on the simulation host machine. However, the main drawback of this method is lack of accuracy in terms of modeling SW execution time and in terms of HW modeling. It is because this method takes functional simulation for the SW and HW parts of the system.

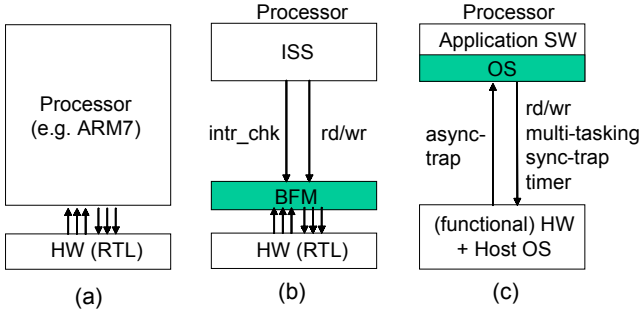

Figure 1 Processor to be modelled (a), classical ISS execution in timed HW/SW cosimulation (b) and classical native execution of application SW and OS (c).

In this paper, to take advantage of both of the advantages of the two classical methods, i.e. simulation accuracy and speed, we present a fast and accurate SW simulation model called *fast timed SW model*.

To achieve fast simulation, the fast timed SW model uses native execution of application SW and OS instead of running the ISS. To enable accurate simulation, it performs timed simulation of application SW and OS.

When building fast timed SW models, we need to solve two problems: (1) how to enable timing synchronization between the native execution and HW simulation and (2) how to obtain the portability of native execution (that needs multi-tasking from simulation environments to emulate its multi-tasking operation) on

different simulation environments (that give different types of multi-tasking). To solve the problem of synchronization, we present a function called **delay()**. To enable the portability, we present an adaptation layer called simulation environment abstraction layer (SEAL).

This paper is organized as follows. Section 2 and 3 give a short review of related work and preliminaries. Section 4 presents the fast timed SW model and our problems. Section 5 and 6 address our solution to the two problems. Section 7 gives our case studies. Section 8 concludes this paper.

2. Related Work

To have accurate SW simulation, we need to simulate the OS and HAL as well as application SW. In conventional cosimulation methods (e.g. described in [2]), only cosimulation with ISS can simulate them. In [3], an ISR is modelled as a part of BFM. However, the timing delay of ISR is not simulated and no OS is simulated. In [4], a method of automatic generation of OS simulation models, for cosimulation purpose, is presented. Our method improves the method by introducing the synchronization function, **delay()** and SEAL to achieve a systematic method of building timed simulation models of OS and application SW.

Conventionally, a HAL is used to ease OS porting on target processors/boards. WindowCE OAL (OEM abstraction layer) [5] and a HAL of eCos [6] are some of HAL examples. These HALs are usually dependent on OSs. a386 is a HAL which is originally dependent on i386 processor [7]. Later, a386 is ported on different processors (e.g. ARM7) than i386. Recently VSIA is working on a standard of embedded SW interface called HW dependent SW (HdS) by defining HAL APIs [11].

When building native OSs, the HAL is used to target the real OS on a simulation host OS [8][9]. In Xenomai project [8] (which is based on CarbonKernel OS simulation model [10]) and in Choices (a component-based OS design environment) [9], a nanokernel is used as a HAL. The nanokernel is processor-dependent codes in the OS. The nanokernel is ported on a simulation host OS. The other processor-independent OS codes run on top of the nanokernel. As mentioned in Section 1, native OS lacks in timed SW simulation and the support for timed HW/SW cosimulation.

In SoCOS [14], an emulation of OS APIs is supported in the simulation of entire SoC including embedded SW. However, the emulation does not validate a real OS design. In addition, the synchronization between the OS emulation and HW simulation including the propagation of processor interrupt is not clearly explained.

In [15], using a conventional native OS (VxSim in this case), OS simulation is performed in HW/SW cosimulation. Compared to the method, our method has three major contributions. One is to present a systematic method of developing the SW simulation model based on the simulation model of HAL while the method in [11] does not present a method of developing the simulation model. Another contribution is more flexible and accurate synchronization between the SW simulation model and HW simulation. The method in [11] offers a periodic synchronization while our method can offer more flexibility and timing accuracy in synchronizing both SW and HW simulation by annotating SW execution delay anywhere in the real code of application SW and OS. The other contribution is the portability of SW simulation model.

3. Preliminaries: SW in SoC Design

In SW design for SoC, the following two problems are crucial: (1) dealing with the design complexity of OS design and (2) hardware independent SW design. The OS design is complex since it includes sophisticated functionalities such as task scheduling, synchronization, interrupt management, memory management and I/O (device drivers). The OS design needs to be HW independent to enable the portability of OS and application SW over several HW architectures to enable the exploration of different HW architectures with the same SW code.

To tackle the complex OS design, the designer needs to implement the OS in a modular and incremental way. To enable the HW independent OS design, the designer needs an abstraction of underlying HW architecture.

Figure 2 shows a simplified view of SW in SoC. We design the SW over three abstraction levels: OS architecture level, HAL level, and ISA (instruction set architecture) level.

The application SW is designed using OS APIs. At the OS architecture level, the OS APIs are determined, but the specific implementations of OS APIs are not yet determined. To enable modular and incremental OS design, we divide the OS into three parts: task scheduling, interrupt management, and I/O (communication). At OS architecture level, the designer determines a set of OS APIs among possible sets of APIs (e.g. POSIX APIs).

HAL is an abstraction of underlying HW architecture. We define HAL as all the software that is directly dependent on the underlying HW. The examples of HAL include boot code, context switch code, codes for configuration and access to HW resources, e.g. MMU, on-chip bus, bus bridge, timer, etc. HAL provides the virtual SW component with a set of HAL APIs. As the HAL APIs, we can use a standard HAL for SoC design, e.g. HdS-API in VSIA [11], OS vendor-specific HAL APIs [6], or SoC architecture specific ones.

At HAL level, HAL provides the OS with HAL APIs. The OS is designed (from scratch or configuring an

existing OS) using the HAL APIs. At HAL level, the specific implementations of OS APIs are determined. However, those of HAL APIs are not yet determined.

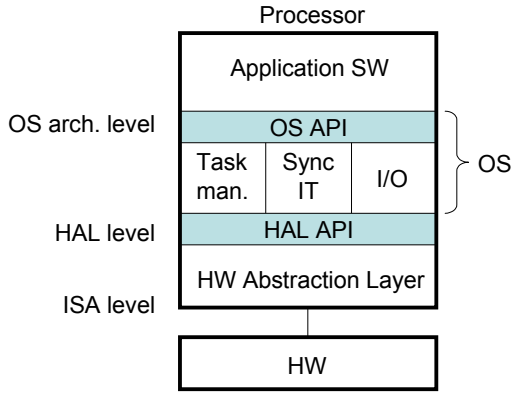

Figure 2 SW architecture and abstraction levels.

At ISA level, the designer implements the HAL APIs. At this level, all the SW code can be compiled and downloaded on to the target processor.

To validate the implementation at a level, the designer uses simulation models suitable to the abstraction level. At ISA level, an ISS is used as a SW simulation model. The fast timed SW model is a HAL level simulation model that uses ISA level delay information.

4. Proposed Simulation Model and Problems Definition

4.1 Fast Timed SW Model

Figure 3 shows a simple view of fast timed SW model. It consists of application SW, OS, the simulation model of HAL and a BFM called EBFM (extended BFM).

As the native execution of application SW and OS in Figure 1 (c), the fast timed SW model executes application SW and OS natively on the simulation host machine. To achieve accurate, i.e. timed simulation in the native execution, the SW execution delay is annotated into the code of application SW and OS and into the simulation model of HAL. In Figure 3, delay annotation is exemplified with '+delay'.

Compared with the classical ISS execution in Figure 1 (b), the fast timed SW model replaces the BFM with the EBFM. The EBFM consists of conventional BFM (performing conventional memory accesses) and simulation environment abstraction layer (SEAL).

Figure 4 shows code examples of fast timed SW model: an application SW task, an OS function (**OS_init()**), and a simulation model of a HAL API (**create_task()**). As shown in the example of OS function code in Figure 4, for delay annotation, we insert function **delay()** (with SW execution delay) into the real OS code as well as into the application SW code and the simulation model of HAL. For the delay calculation, we use conventional methods of SW execution delay estimation, e.g [12][16].

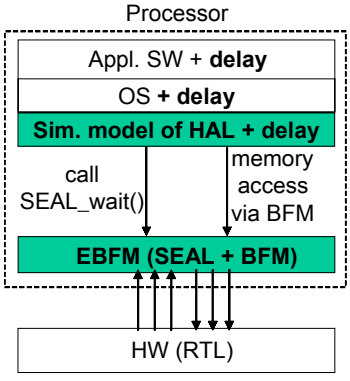

Figure 3 Fast timed SW model.

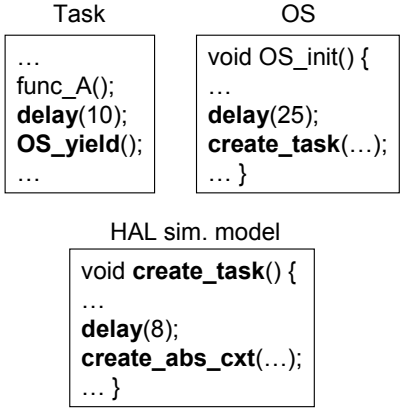

Figure 4 Code examples.

When building a native OS, first we need to separate the real OS into two parts: one is dependent on the target processor and the other is processor-independent. In our case, the target processor dependent part is HAL. The processor independent part is usually written in high-level languages, e.g. in C/C++. HAL can be written in assembly code or in high-level languages.

In the case of native execution of application SW and OS, since we execute it on a simulation host, we can use the processor independent code part of the OS (assuming that the application SW is processor-independent) for native execution without change. However, for the processor dependent part, i.e. HAL, since we cannot run the original HAL (possibly in assembly code) on the simulation host, we need to build a simulation model (for HAL) that can run on the simulation host.

Details of building the simulation models of HAL APIs are given in our previous work [4]. This paper is to focus on the timing synchronization and portability problems in building the fast timed SW model.

4.2 Problems Definition

When building fast timed SW models, we need to solve the following two problems: (1) how to enable timing synchronization between the native execution and HW simulation and (2) how to obtain the portability of native execution on different simulation environments.

Problem 1: Timing Synchronization
In the native execution, the task scheduler in the native OS emulates the multi-tasking of application SW tasks. The scheduling operation is closely related to the processor interrupt processing since task execution is pre-empted by the interrupts and new tasks can be scheduled by the task scheduler invoked by the processor interrupts. In terms of synchronization between SW and HW simulation, the processor interrupt is modelled by the events propagated from HW to SW simulation. Thus, the emulation of multi-tasking and synchronization between SW and HW simulation are closely related with each other via the simulation of processor interrupt. This problem is a new problem in the cosimulation area. To solve this problem, we present a solution of synchronization function **delay().** It performs the synchronization, propagates events that represent processor interrupts from HW to SW simulation, and enables the interrupt service routines to be invoked by the processor interrupts.

Problem 2: Portability of Simulation Model
A simulation environment is an ensemble of SW and/or HW simulators (e.g. SystemC, SpecC, etc) and/or host machine OSs (e.g. Unix, Linux, Windows, etc.). For the emulation of multi-tasking operation of native OS, we need multi-tasking functions from the simulation environment. Since there are various simulation environments (different environments and different versions of environments), depending on each of them, the multi-tasking implementations are different. The implementation of the same native OS on different simulation environments can take a lot of efforts of manual adaptation. Thus, the portability problem will limit the (re)usage of SW simulation models in different or future simulation environments. To enable the portability of simulation model, we need to abstract simulation environments, especially their multi-tasking implementations. To this problem, we present an adaptation layer called simulation environment abstraction layer (SEAL) for the abstraction of simulation environments.

5. Timing Synchronization between SW and HW Simulation

In our method, function **delay()** enables timed SW simulation and timing synchronization between SW and HW simulation. Function **delay()** works in collaboration with a SEAL API, **SEAL_wait()**. Figure 5 and 6 show pseudo codes of **delay()** and **SEAL_wait()**.

```
1   void delay(int delay) {
2     int last_time;
3     time2consume = delay:
4
5     while( time2consume > 0 ) {
6       last_time = cur_SW_time;
7       SEAL_wait(time2consume, event_return);
8       cur_SW_time = event_return->time;
9       time_elapsed = cur_SW_time – last_time:
10      time2consume -= time_elapsed;
11      if( event_return->flag == true ) ISR();
12    }
13  }
```

Figure 5 Function delay().

```
1   void SEAL_wait(int delay, ext_event* event_value) {
2       target_SW_time = cur_HW_time + delay;
3
4       while( cur_HW_time < target_SW_time ) {
5             if(proc_intr->new_event == true) {
6                   event_value->flag = true;
7                   event_value->time = cur_HW_time;
8                   return;
9             }
10            advance_HW_time();
11      }
12      event_value->flag = false;
13      event_value->time = cur_HW_time;
14  }
```

Figure 6 Function SEAL_wait()

When function **delay()** is executed in the fast timed SW model, the SW execution delay value is sent to SEAL in line 7 of Figure 5 by calling a SEAL API, **SEAL_wait()**. As shown in Figure 6, **SEAL_wait()** advances HW simulation time watching on external events, i.e. processor interrupts (in line 5 and 10 of Figure 6).

SEAL_wait() returns in two cases. Before time period **delay** elapses, if there is a processor interrupt event, the function returns (line 5-8 in Figure 6). If there is no interrupt event during the time period, it returns after the entire time period **delay** elapses (after line 12-13).

When function **SEAL_wait()** returns in function **delay()** of Figure 5, both SW and HW simulation times are synchronized (in line 8 of Figure 5). Note that the

return value of **SEAL_wait()**, **event_value->time** is set to current HW simulation time (**cur_HW_time**) in line 7 or 13 of Figure 6.

If there is an interrupt before time period **delay** elapses, there is a remaining delay for the preempted SW task or OS code. Thus, the remaining delay value is calculated (line 10 of Figure 5). Then, if there is an interrupt, the interrupt service routine (ISR) is simulated (line 11). When the ISR returns, if there remains still a time delay for the (preempted) SW task or OS code, function **SEAL_wait()** is called again (line 7).

Note that, even in the ISR execution, function **delay()** can be executed and that during the ISR execution, the OS scheduler can be called and another (preempted) task can be resumed.

As shown in Figure 5 and 6, function **delay()** detects events for processor interrupts (line 11 in Figure 5) and propagates them to the native execution by calling the simulation model of ISR. It also synchronizes SW and HW simulation time (line 8 in Figure 5).

6. Simulation Environment Abstraction Layer

To abstract different multi-tasking implementations in different simulation environments, SEAL gives **an abstraction of task context** to the simulation model of HAL. To do that, it provides for a data structure of an abstract context and a set of APIs to use the abstract contexts. The abstract context includes host machine register values and stack pointer. There are three APIs for the abstract contexts: **create/delete_abs_cxt()**, **abs_cxt_switch()**.

Figure 7 shows an example of SEAL API. In the figure, we implement the SEAL API with Unix as a simulation environment. In this case, the data structure of abstract context **abstract_cxt** has, as the only member, a data structure of Unix user-level context (**ucontext_t**).

In Figure 7, SEAL API **create_abs_cxt()** creates an instance of abstract context. As shown in the figure, a HAL API, **create_task()** uses the SEAL API, **create_abs_cxt()**. In this case, when we use another simulation environment (e.g. Windows) than Unix, to build a new fast timed SW model, we have only to port the SEAL APIs on the new simulation environment without changing the simulation models of HAL APIs and the other codes of application SW and OS.

7. Case Studies

In our case studies, to show the application of synchronization function **delay()** and SEAL, we implement three different cases of execution model for the fast timed SW model. By the execution model, we mean a configuration of simulation environment. Figure 8 shows the three cases.

```
// SEAL abstract task data structure
class abstract_cxt { ucontext_t task_cxt; };
abstract_cxt cxt_inst[NO_TASKS];

// SEAL API
create_abs_cxt(int id, func_ptr task_fcn) {
    cxt_inst[id].task_cxt->uc_link=0;
    cxt_inst[id].task_cxt->uc_stack.ss_sp=malloc(STACK);
    cxt_inst[id].task_cxt-> uc_stack.ss_size=STACK;
    cxt_inst[id].task_cxt-> uc_stack.ss_flags=0;
    makecontext(cxt_inst[id].task_cxt, task_fcn, 1); }
// HAL API
create_task(int task_id, func_ptr* task_entry_func) {
    …
    create_abs_cxt(task_id, task_entry_func);
    … }
```

Figure 7 SEAL API example.

In Case 1 (in Figure 8 (a)), we implement a native OS as a Unix process. For HW simulation, we use SystemC. Since we run SystemC simulation and the native OS in different Unix processes, we use Unix IPC for the communication between two processes. In this case, the native OS itself implements Unix-specific multi-tasking for the emulation of its multi-tasking operation. Thus, the EBFM has only to support function **SEAL_wait()** for the function **delay()** called in the native execution of application SW and OS and in the execution of HAL simulation model. Case 1 is very useful when we use native OSs from commercial OS vendors (e.g. VxSim [1]). In this case, the native OS already exists as a separate process on a simulation host.

In Case 2, we integrate the native OS into SystemC environment. In Figure 8 (b), a bold rectangle represents a SystemC module. As shown in the figure, application SW, OS, and HAL are contained in a SystemC module. In this case, we use only one Unix process for the SystemC simulation. The HAL of the native OS uses the SEAL APIs of EBFM (in this case, using Unix user-level multi-tasking). In Case 2, when we move the SystemC model on another simulation environment, e.g. Windows or Linux, we have only to change the SEAL without changing the other part of fast timed SW model. In terms of simulation speed, Case 2 can be a better solution than Case 1 since IPC is not used for the communication between the native OS and HW simulation in Case 2.

In Case 3, we do not use Unix user-level multi-tasking in SEAL, but use SystemC multi-tasking function for the emulation of native OS multi-tasking. To do that, we model a SW task as a SystemC module. In Figure 8 (c), each of three application tasks T_1, T_2, and T_3 is represented as a SystemC module, respectively. By using SystemC modules, context switch between tasks can be emulated with SystemC multi-tasking, i.e.

notify(sc_event) and wait(event). Case 3 is one of candidate solutions of SW validation when building OSs using SystemC

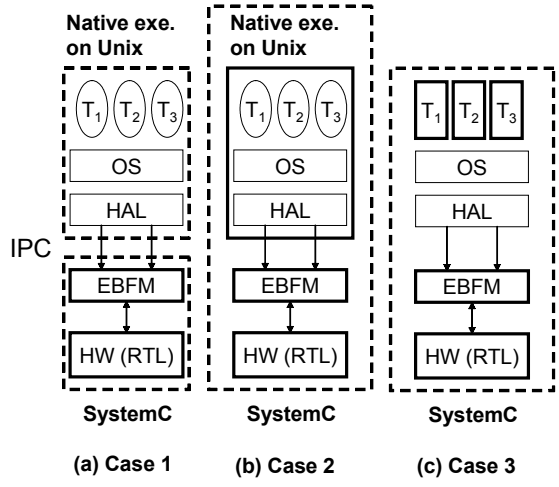

Figure 8 Three execution models.

In our case studies, we used three system examples: McDrive, VDSL, and IS-95 [13]. As the target architectures, McDriver system has one ARM7, three IPs, and point-to-point (p2p) interconnections. VDSL system has two ARM7s, one IP, and p2p interconnections. IS-95 system has two ARM7s, two 68000s, and p2p interconnections.

In terms of simulation runtime, the fast timed SW model gives orders of magnitude higher performance compared to cycle-accurate cosimulation using ISSs. An implementation of Case 3 for the IS-95 example, for the simulation of 0.4 sec in real time, the fast timed SW model (communication between processors are modelled at a transaction-level) gives 37 sec of simulation runtime while cycle-accurate cosimulation with four ISSs (ARMulators and 68000 ISSs) and SystemC (HW simulation) gives 72,744 sec of simulation runtime.

8. Conclusion

As a fast and accurate SW simulation model, we present a model called **fast timed SW model**. The model enables fast simulation by native execution of application SW and OS and simulation accuracy by timed SW and HW simulation.

To solve the problem of timing synchronization between SW and HW simulation, we present function **delay()** to detect and propagate processor interrupt events from HW to SW simulation. To enable the portability of native execution of application SW and OS on different simulation environments, we present an adaptation layer called simulation environment abstraction layer. In our case studies, we present three cases of execution model for fast timed SW models and simulation runtime results.

References

[1] VxSim, Windriver Systems Inc. Available at http://www.windriver.com/products/html/vxsim.html

[2] James A. Rowson, "Hardware/Software Co-Simulation", Proc. DAC, 1994.

[3] L. Semeria and A. Ghosh, "Methodology for Hardware/Software Co-verification in C/C++", Proc. ASPDAC, 2000.

[4] S. Yoo, G. Nicolescu, L. Gauthier and A. A. Jerraya, "Automatic Generation of Fast Timed Simulation Models for OS in SoC Design", Proc. DATE, 2002.

[5] Microsoft Windows CE 3.0, hal component, http://msdn.microsoft.com/library/default.asp?url=/library/ en-us/wcesdkr/htm/_wcesdk_hal_component.asp

[6] eCos, http://sources.redhat.com/ecos/

[7] a386, http://a386.nocrew.org/

[8] Xenomai project, http://savannah.gnu.org/projects/xenomai/

[9] S. M. Tan, *et. al.*, "Virtual Hardware for Operating System Development", Technical rep., UIUC, Sep. 1995. http://choices.cs.uiuc.edu/uChoices/Papers/uChoices/vchoices/vchoices.pdf

[10] Carbon Kernel, http://www.carbonkernel.org/

[11] Virtual Socket Interface Alliance, http://www.vsi.org

[12] M. Lajolo, M. Lazarescu, A. Sangiovanni-Vincentelli, "A Compilation-based Software Estimation Scheme for Hardware/Software Co-simulation", Proc. CODES, 1999.

[13] S. Yoo, *et. al.*, "Fast Prototyping of an IS-95 CDMA Cellular Phone: a Case Study", Proc. APCHDL, Oct. 1999.

[14] D. Desmet, D. Verkest, H. De Man, "Operating System Based Software Generation for Systems-on-Chip", Proc. DAC, pp. 396-401, June 2000.

[15] M. Bradley and K. Xie, "Hardware/Software Co-Verification with RTOS Application Code", Mentor Graphics Inc. http://www.mentor.com/soc/fulfillment/mentorpaper_10280.pdf

[16] C. Brandolese, W. Fornaciari, F. Salice, D. Sciuto, "Source-Level Execution Time Estimation of C Programs", Proc. CODES, pp. 98-103, May, 2002.