A High-Level Programming Paradigm for SystemC

Mark Thompson and Andy D. Pimentel

Department of Computer Science, University of Amsterdam
Kruislaan 403, 1098 SJ Amsterdam, The Netherlands
{mthompsn,andy}@science.uva.nl

Abstract. The SystemC language plays an increasingly important role in the system-level design domain, facilitating designers to start with modeling and simulating system components and their interactions in the very early design stages. This paper presents the SCPEx language which is built on top of SystemC and which extends SystemC's programming model with a message-passing paradigm. SCPEx's message-passing paradigm raises the abstraction level of SystemC models even further, thereby reducing the modeling effort required for developing the (transaction-level) system models applied in the early design stages as well as making the modeling process less prone to programming errors. Moreover, SCPEx allows for performing automatic and transparent gathering of various simulation statistics, such as statistics on communication between components.

1 Introduction

The heterogeneity of modern embedded systems and the varying demands of their target applications greatly complicate the system design. It is widely agreed upon that traditional design methods fall short for the design of these systems as such methods cannot deal with the systems' complexity and flexibility. This has led to the notion of *system-level design* in which designers already start with modeling and simulating system components and their interactions in the very early design stages (e.g., [5, 8]). More specifically, system-level models typically represent application behavior, architecture characteristics, and the relation (e.g., mapping, hardware-software partitioning) between application(s) and architecture. These models do so at a high level of abstraction, thereby minimizing the modeling effort and optimizing simulation speed that is needed for targeting the early design stages. This high-level modeling – of which the class of *transaction-level models* [2] is especially gaining interest – allows for early verification of a design and can provide estimations on the performance, power consumption or cost of the design.

The SystemC language [1] plays an increasingly important role in the system-level design domain. This language allows for modeling systems at a relatively high (behavioral) level of abstraction, after which the models can be gradually refined towards a level of abstraction at which synthesis of the modeled design is possible.

In this paper, we present the SCPEx (SystemC Pearl Extension) language that is built on top of SystemC v2.0, and which extends SystemC's programming model with a message-passing paradigm – based on the Pearl simulation language [7] – raising the abstraction level of SystemC models even further. This increase of abstraction level reduces the modeling effort required for developing the (transaction-level) system models applied in the early design stages, and makes the modeling process less prone to programming errors. Moreover, SCPEx also allows for automatic and transparent gathering of various statistics on, for example, communications between components.

The paper is organized as follows. In Section 2, we motivate the development of SCPEx. Section 3 provides an overview of SCPEx's simulation primitives with their semantics. To illustrate SCPEx's programming paradigm and demonstrate its modeling power, Section 4 presents SCPEx code samples taken from a realistic case study. This section also illustrates SCPEx's support for YML (Y-chart Modeling Language) [3] which provides an explicit description of the structure of SCPEx models (i.e., how components are connected and parameterized). Finally, Section 5 concludes the paper.

2 Raising the abstraction level of SystemC

To cope with the growing complexity of (embedded) systems, system-level design has been increasing the level of abstraction at which systems are modeled and simulated. A clear example of this trend is the emergence of the high-level modeling and simulation language SystemC [1]. Hardware description languages (HDLs) such as VHDL and Verilog are slowly being replaced by languages such as SystemC (and related variants like SpecC [6]) in especially the early stages of design. The rationale behind this transition to higher-level languages is that they allow for more efficient exploration of system-level design tradeoffs and enable earlier verification of the entire system, thereby reducing risks. In addition, software is playing an increasingly important role in systems design, where traditional HDLs typically are hardware oriented. By building the new modeling and simulation languages on existing high-level languages such as C/C++, existing code can easily be (re-)used during system design. Applying these high-level design languages usually means that a system initially is modeled at a high (behavioral) level of abstraction, after which the model is gradually refined towards a level of abstraction at which synthesis of the modeled design is possible.

In the past decade, we have gained a lot of experience with our own high-level architecture modeling and simulation language called Pearl [7, 4]. This object-based discrete-event simulation language has proven to be highly efficient for abstract performance modeling and simulation of computer architectures, both in terms of modeling effort as well as simulation speed. Pearl's modeling efficiency is mainly due to the message-passing programming paradigm it adheres to. A Pearl program consists of a collection of objects running concurrently, where each object executes ordinary sequential code and has its own data space which cannot directly be modified by other objects. When an object wants to modify some remote data, it sends a message to the object with a request to change the data. Subsequently, the remote object will only change the data after explicitly accepting the request.

Comparing Pearl's programming paradigm and associated primitives to those of SystemC v2.0, one can observe a number of areas where Pearl adopts a higher level of abstraction. Pearl, for example, abstracts from the concept of ports and explicit channels between ports as applied in SystemC. Pearl objects only contain a single "link" with each neighboring object and can communicate with these objects using *remote method calling* implemented via the aforementioned message-passing mechanism. Implementing remote method calling by means of message passing implies that objects remain autonomous in their execution: an object decides itself when to process an incoming remote method call. This also means that in Pearl buffering of messages at objects is taken care of by the run-time system, rather than having to implement explicit buffering as one would have to do in SystemC. As we will show later on, this message-passing programming paradigm nicely fits the modeling of transactions in transaction-level models. In addition, Pearl's message passing primitives (performing a remote method call) transparently incorporate inter-object synchronization, while this is done by separate and explicit event notifications in SystemC. The latter may be more prone to programming errors. Another valuable benefit of Pearl's message-passing paradigm is that the run-time system can automatically gather statistics on communications between objects, utilization of objects, critical paths between objects, and so on. Such automatic feedback may be a significant help to recognize early bottlenecks in the modeled system.

This paper presents SCPEx (SystemC Pearl Extension), which is a software-library on top of SystemC v2.0 (see Figure 1) that provides SystemC modelers with Pearl's message-passing programming paradigm. Hence, it allows for raising the abstraction level of SystemC models, which can be especially beneficial for the efficient development of initial system-level models. As illustrated in Figure 1, modelers have the choice of *i)* using a so-called Model of Computation (MoC) – like process networks, dataflow networks, CSP, and so on – that has been implemented on top of SystemC/SCPEx, *ii)* using SCPEx to start the modeling after which models can be refined towards SystemC v2.0 in a later stadium, or *iii)* immediately start modeling in SystemC v2.0.

As will be explained later on, SCPEx also features support for the Y-chart Modeling Language (YML) [3] that is used in our Sesame system-level modeling and simulation framework [4] to describe the structure of models, i.e., how model components are connected and parameterized. YML support for SCPEx gives us the choice of using either Pearl or SCPEx architecture models in Sesame, where the translation of Pearl models to SCPEx models (and vice versa) is relatively straightforward. The latter will facilitate exporting Pearl models to SystemC-based environments as well as importing SystemC (IP) models in Sesame.

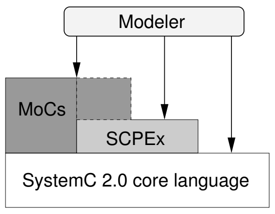

Fig. 1. SCPEx: Adding a layer on top of SystemC v2.0.

3 SCPEx

The SCPEx library extends SystemC's `sc_module` class with an `scpex_module` class to provide the modeler with all the required functionality to use the Pearl programming paradigm in SystemC. Whereas SystemC modules can contain many (different types of) processes, SCPEx only allows one process in a module in order to maintain a strict message-passing paradigm for exchanging data between processes/modules. The process in an SCPEx module is managed by the simulation scheduler as a non-preemptive user-level thread. As there is a one-to-one mapping of execution threads to modules, we speak of the "execution of a module". The simulation scheduler schedules a module when it is ready to run. If – at some point in simulated time – there is more than one module ready to run, the scheduler will select them in turn. The modeler should assume that the scheduling is non-deterministic as to prevent that the correct working of the simulation depends on the order in which modules are scheduled.

An SCPEx module consists of C++ code with any number of functions, methods and SCPEx primitives. Functions and methods differ only in that execution of methods may be requested by other modules, whereas functions are only used internally by a module. There are two types of SCPEx primitives: block and call primitives. A call primitive performs a remote method call, merely being a *request* to another module to execute one of its methods. A block primitive specifies that a module is going to wait for the simulation clock, for one or more method(s) to be called by a remote module, or a combination of these two. As mentioned before, the remote method calling is performed via a message passing mechanism of which the implementation details are hidden from the modeler by means of the call/block primitives. In the remainder of this section, we present the semantics of these primitives and explain the message passing mechanism.

There are two types of call primitives: synchronous and asynchronous. They have the following abstract syntax:

abstract syntax	`{a}synch_call( remote_module,`
	`method,`
	`argument_1, ..., argument_n )`
examples	`int res = synch_call(alu, add, 2, 4);`
	`asynch_call(alu, add, 2, 4);`

A `call` statement sends a request to `remote_module` to execute `method` with the associated arguments. Here, `remote_module` is a pointer to the remote module and `method` is the name of the method that should be executed in the other module where the `argument_x` variables have to match the argument types of that method. The `synch_call` statement does not return until it receives an acknowledgment from the remote module. To illustrate this, let us assume that the calling module is named `module1`. A `synch_call` will halt the thread of `module1` and yield control to the scheduler. The scheduler will then select one of the other modules to resume execution. At some point, the remote module (`remote_module`) may accept and execute the requested `method`. Doing so, `method` will send an acknowledgment back to `module1`. After receiving this acknowledgment, the scheduler knows that `module1` is runnable again and may be scheduled for execution. The `synch_call` statement in `module1` then evaluates to the value of the acknowledgment with which the method of `remote_module` has replied. Replies are sent using a special reply primitive:

abstract syntax	`reply( reply_data )`
example of a	`void add(int a, int b) {`
reply in a	`       /* model, e.g., computation time */`
synchronous	`       reply(a+b);`
method	`}`

The asynchronous call statement has a similar syntax to `synch_call`. This asynchronous version of the call primitive also sends a request to execute a method with the given arguments to `remote_module`. However, this statement does not wait for `remote_module` to handle the request, but returns immediately. It is not necessary for the remote method to issue an acknowledgment using the reply primitive. This explains why `asynch_call` has no reply value. We note that methods that issue a reply may be called synchronously as well as asynchronously (the reply-value is then ignored), while synchronously calling a method without a reply causes the caller to block indefinitely.

On the implementation side, a remote method call made by `synch_call` and `asynch_call` results in a message being sent from the calling module to the remote module. This message contains all the information a remote module requires to handle the request. This includes a method identifier and the actual parameters of the method as well as the reply data and pointer to the calling module required to notify the calling module upon completion of the method.

A module may receive more method-call requests (i.e., messages) than it can process at any given time. Therefore, there is a need to store unprocessed messages. Any two communicating modules are connected using a *message queue* that accepts writing of messages by the calling module and reading by the receiving module. This is totally transparent to the modeler as the call and block primitives handle the queuing and dequeuing of messages.

abstract syntax	`block ( method1, method2, ..., methodN )`
	`blockt( time )`
	`blockt( time-out, method1, method2, ..., methodN )`

examples	`block (add, sub, mult, div);`
	`blockt(20);`
	`blockt(10, read, write);`

Semantically, the block primitive – being similar but not identical to SystemC's `wait` primitive – indicates what behavior of a module will be simulated in the following simulation time-steps. One instance of this primitive – the `blockt` statement – is identical to SystemC's `wait(time-units)` as it simulates that a module is busy for a fixed number of time-steps after which the module resumes execution.

To model that a module is ready to process a request (i.e., an incoming remote method call), the `block` statement is used. It allows for specifying a selection of methods in the argument list in order to accept only requests for these particular methods. Applying the special keyword `any` as an argument, the `block` statement will accept any incoming request. The implementation of the `block` statement first checks whether or not an acceptable request is already stored in the module's inbound message queue. If present, then the first message for `methodX` (being one of the arguments of the `block` statement) is dequeued and `methodX` is called in the local module with the parameters stored in the message. If there is no request stored in the message queue, then the module halts execution and yields control to the scheduler. The module will be rescheduled whenever a matching request arrives. A `block` statement does not return control to the next statement until one of the specified methods has been executed. The last form of the block primitive is a combination of the previous two: it waits for one or more methods to be called with a specified time-out. So, if no incoming method call was received during this time-out period, then control is returned after `time-out` time-steps (thus operating like a `blockt` statement).

Figure 2 illustrates the semantics of the aforementioned primitives using a simple simulation model of a processor, cache and memory. We note that, for the purpose of brevity, we slightly simplified the code (e.g., we omitted instantiations and initializations). The next section will present real SCPEx code samples. In the code of Figure 2, the processor reads a word, prefetches another, and reads the two words again. The prefetch is simulated by calling the `fetch` method asynchronously: the processor does not wait for the result. The cache implements the `fetch` method. If the requested data is in the cache, it takes only 2 time-steps to complete the request, otherwise it takes 4 time-steps plus the time needed to fetch the data from memory. The memory implements a fetch that always takes 15 time-steps. The graph in Figure 2 shows the activity of the different modules during the time of the simulation: solid lines indicate that the module is busy, dashed lines indicate that the module is waiting for the results of a synchronous method call, while blanks imply that the module is idle and waiting for a method call. The scheduler terminates the simulation when all modules are blocking and no new requests are made. In this case, the simulation is stopped at $t = 50$.

It is not difficult to see that SCPEx nicely aligns with the modeling support needed for implementing transaction-level models. The transactions in transaction-level models – being atomic transfers of high-level data and/or control – perfectly map onto SCPEx's communication primitives.

```
class processor: scpex_module {
   void main() {
     synch_call(cache, fetch, 0x0000);
     blockt(4); /* model some computation */
     asynch_call(cache, fetch, 0x0020);
     blockt(9);
     synch_call(cache, fetch, 0x0000);
     blockt(4);
     synch_call(cache, fetch, 0x0020);
   }
}
```

```
class memory: scpex_module {
   void main() {
      while (1)
         block(fetch)
   }
   void fetch(int addr) {
      blockt(15);
      reply(lookup(addr));
   }
   /* module functions: */
   int lookup(int addr) {...}
}
```

```
class cache: scpex_module {
   void main() {
     while (1) /* main loop of module */
        block(fetch)
   }
   void fetch(int addr) {
     blockt(2);
     if (present(addr))
        reply(lookup(addr));
     else {
        int data =
           synch_call(memory, fetch, addr);
        blockt(2);
        store(data);
        reply(data);
     }
   }
   /* module functions: */
   int present(int addr) {...}
   int lookup(int addr) {...}
   int store(int data) {...}
}
```

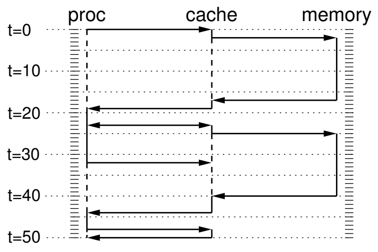

Fig. 2. Illustrating SCPEx's primitives

4 Illustrating SCPEx: a case study

To show SCPEx in action, we present several code snippets from a case study that was performed earlier with our Sesame system-level modeling and simulation framework [4] (using Pearl), and that we have repeated using SCPEx. In this case study, we mapped a Motion-JPEG encoder application – modeled as a Process Network – onto an architecture model that models a bus-based shared-memory multiprocessor architecture. This mapping is performed via trace-driven co-simulation in which the M-JPEG application model generates *application events* representing the workload imposed on the architecture, while the (transaction-level) architecture model simulates the performance consequences of these application events. Architecture models in Sesame usually account for performance constraints (i.e., timing behavior) only and do not model functional behavior since the latter is already captured in the application model. For a comprehensive description of Sesame, the reader is referred to [3, 4].

In Sesame, YML (Y-chart Modeling Language) [3] is used to describe the structure of simulation models and their parameterization. YML is based on XML so as to be simple and flexible and was developed to allow for rapid creation and modification of simulation models. Sesame applies YML to describe both application and architecture models as well as the mapping between these models. Besides statically describing model components and their connections, YML also features support for dynamic descriptions using *scripting*. This scripting allows, for example, for rapidly defining models that consist of many similar components (such as a large lattice of switches in a network). Moreover, as library support is inherent to XML, it is possible to make libraries of (parameterized) model component descriptions. Since YML supports hierarchy, this means that it is easy to reuse model components together with their descriptions as part of different models. To simplify the use of YML even further, a YML editor has been developed to compose model descriptions using a graphical interface. As the properties of YML can also be very beneficial for SystemC-based models, we added YML support to SCPEx. An additional advantage to adding YML support is that Sesame architecture models can either use the Pearl simulation language or SCPEx.

To illustrate YML, the code in Figure 3 shows a part of the architecture model description for the M-JPEG case study. Each `node` represents a simulation module, which in our example are a processor (`mp`) and a `bus` component. The value of the `class` property describes which SCPEx class needs to be instantiated for this object. The `port` objects together with the `link` element describe the connection between the processor and the bus. The `init` property describes the initialization values of variables of a module. In SCPEx, the initialization values can be integers, port names, or arrays of one of the two types. Before port names are assigned to the corresponding variable, they are transformed to module references (which are used by the call primitives). To do so, the YML interface of SCPEx traces the links of the port (possibly crossing several levels of hierarchy in YML) until it reaches the node it is connected to.

Figure 4 shows (most of) the real SCPEx code of the processor and bus modules from the M-JPEG case study. We only omitted the inclusion of header files and the `store` method in the processor module since the `store` is almost identical to the `load` method. Modules have to implement a constructor with a fixed prototype, as shown in the declaration of the proc and bus classes (two top boxes in Figure 4). This

```
<network xmlns=".../YML" name="MJPEG-arch" class="net">
 ...
 <node name="mp" class="scpex_object">
    <property name="class" value="proc"/>
    <property name="init" value="net,10,[359,0,0,0,154,1,45,0,4,154]"/>
    <port name="net" dir="out"/>
    <port name="vp" dir="in"/>
 </node>
 ...
 <node name="bus" class="scpex_object">
    <property name="class" value="bus"/>
    <property name="init" value="mem,1,8"/>
    <port name="input" dir="in"/>
    <port name="mem" dir="out"/>
 </node>
 ...
 <link innode="mp" inport="net" outnode="bus" outport="input"/>
```

Fig. 3. Example YML code for processor and bus modules.

constructor is used by the YML interface to instantiate a module. The INIT macro is used to initialize module variables with values given in the YML specification file.

A processor module may be requested – by means of an application event from the application model – to model the timing consequences of a computation (`compute` method) or a communication (`load` and `store` methods). The `opers` array in the `compute` method contains the latencies of the different computational operations that can be performed by the processor, while the `operindx` argument specifies which operation is performed by the application at some moment in time. In Figure 3, it can be seen how the `opers` latency-array is passed to the processor module as part of a YML `init` property. The `load` and `store` methods – modeling the processor's communication – simply are synchronous calls to the communication network which in this case is a bus. This implies that the bus, and subsequently the memory that is attached to the bus, will account for the timing consequences of the communication. The bus module accounts for a `setup` latency which models the overhead to gain access to the bus without contention (pure arbitration overhead, etc.). Communication latency due to bus contention (one request waiting for another) is implicitly modeled – so without an explicit arbiter – which is straightforward because of SCPEx's message-passing paradigm. While the bus module services a communication from one processor module, a request for communication from another processor module will simply be stored in the SCPEx message queue of the bus module until the bus is available again. Bus arbitration is thus performed using SCPEx's FCFS message-queue policy, which may be especially useful in early design stages for rapidly obtaining an initial system model. Naturally, an explicit arbiter can be added to realize different arbitration policies.

Explicit references to target modules with which is communicated – such as the processor module in Figure 4 that directly communicates with the bus module – may limit the reusability of SCPEx modules. To avoid such problems, SCPEx models can apply the concept of subtyping. For example, in Figure 4 we can write "`interconnect *net`" instead of "`bus *net`", where `interconnect` is an abstract class (merely being an interface). In addition, the first argument of the `synch_calls`, containing the class of the remote module, should also be adapted accordingly. Subsequently, using the YML description, we can map a bus module, which is a subtype of the `intercon-`

```
struct proc: public scpex_module {
   bus *net;
   int nopers;
   int *opers;

   void main();
   void compute(int operindx);
   void load(int nbytes, int addr);
   void store(int nbytes, int addr);

   proc(sc_module_name name,
        (init_values **) is):
        scpex_module(name) {
           INIT(net);
           INIT(nopers);
           INIT(opers);
   }
};
```

```
void proc::main() {
   while (1)
     block(any);
}

void proc::compute(int operindx) {
   blockt(opers[operindx]);
   reply();
}

void proc::load(int nbytes,
                 int addr) {
   synch_call(bus, net, load,
              nbytes, addr);
   reply();
}

// store method has been omitted
```

```
struct bus: public scpex_module {
   memory *mem;
   int setup;
   int width;

   void main();
   void load(int nbytes, int addr);
   void store(int nbytes, int addr);


   bus(sc_module_name name,
       (init_values **) is):
       scpex_module(name) {
          INIT(mem);
          INIT(setup);
          INIT(width);
   }
};
```

```
void bus::main() {
   while (1)
     block(any);
}

void bus::load(int nbytes, int addr) {
   blockt(setup);
   synch_call(memory, mem, load,
              nbytes, addr, width);
   reply();
}

void bus::store(int nbytes, int addr) {
   blockt(setup);
   synch_call(memory, mem, store,
              nbytes, addr, width);
   reply();
}
```

Fig. 4. The SCPEx code of processor and bus modules of the M-JPEG case study. The top code boxes show the declaration of the modules, while the bottom boxes show their implementation.

`nect` class and which provides the implementation of `interconnect`'s methods, onto the `net` variable. Doing so, we can, for example, substitute the bus in our architecture model by alternative interconnects (such as a crossbar, point-to-point links, etc.) in a plug-and-play fashion.

It should be clear from the code samples presented so far that SCPEx reduces the channel connections needed between modules in comparison to plain SystemC. For any two connected modules, SCPEx uses only a single predefined channel for communication of all data, and this data can be of any type. In SystemC, either a port and associated channel per data type, or channels that communicate composite data (structs or classes) would be needed. The first solution may result in a considerable number of ports/channels and associated read/write statements, while the latter solution requires to explicitly pack and unpack data elements before respectively writing and reading them. Moreover, SCPEx's communication primitives, which can provide synchronization in a transparent manner, make the manual creation and use of SystemC-events redundant. As a result, the errors that are typically caused by complicated event notification schemes are avoided.

Another benefit of SCPEx is that simulation statistics can be gathered automatically at runtime. This is possible since the two mechanisms that are responsible for SCPEx's semantics (message passing and synchronization) are accessible exclusively through the call/block primitives. For example, the utilization of a module – indicating the amount of time a module is busy or idle – can be computed because the simulation scheduler exits or enters the single thread of a module only on a `call` or `block` statement. Utilization can be computed if the primitives record whether the module was busy or idle in the period between being halted and rescheduled. We are currently working on the implementation of more advanced statistics such as contention and bandwidth analysis as well as profiling and call-graph analysis. Of course, a modeler can add his own statistics, either directly in the simulation code or by modifying the SCPEx primitives.

5 Conclusions

In this paper, we have presented the SCPEx language which is built on top of SystemC v2.0 and which raises SystemC's abstraction level by extending its programming model with a message-passing paradigm. Connections and synchronizations between modules in SCPEx do not have to be explicitly programmed, reducing the modeling effort required for implementing transaction-level models as well as reducing the probability of programming errors. To demonstrate SCPEx's modeling power, a case study was presented that uses SCPEx combined with our Sesame system-level modeling and simulation environment. Future research will focus on methods for refining SCPEx models to plain SystemC and mixed SystemC/SCPEx models.

Acknowledgment

We thank Pieter van der Wolf for his valuable comments on a draft version of this paper.

References

1. SystemC initiative. http://www.systemc.org/.
2. L. Cai and D. Gajski. Transaction level modeling: An overview. In *Proc. of CODES-ISSS*, pages 19–24, Oct. 2003.
3. J. E. Coffland and A. D. Pimentel. A software framework for efficient system-level performance evaluation of embedded systems. In *Proc. of the ACM Symposium on Applied Computing (SAC '03)*, pages 666–671, March 2003.
4. A. D. Pimentel et al. Towards efficient design space exploration of heterogeneous embedded media systems. In *Embedded Processor Design Challenges: Systems, Architectures, Modeling, and Simulation*, pages 57–73. Springer, LNCS 2268, 2002.
5. F. Balarin et al. Metropolis: An integrated electronic system design environment. *IEEE Computer*, 36(4), April 2003.
6. D. D. Gajski, J. Zhu, R. Dömer, A. Gerstlauer, and S. Zhao. *SpecC: Specification Language and Methodology*. Kluwer Academic Publishers, Dordrecht, The Netherlands, 2000.
7. H.L. Muller. *Simulating computer architectures*. PhD thesis, Dept. of Computer Science, Univ. of Amsterdam, Feb. 1993.
8. A.D. Pimentel, P. Lieverse, P. van der Wolf, L.O. Hertzberger, and E.F. Deprettere. Exploring embedded-systems architectures with Artemis. *IEEE Computer*, 34(11):57–63, Nov. 2001.