Virtual Integration of Cyber-Physical Systems by Verification

Panagiotis Manolios
Northeastern University
pete@ccs.neu.edu

Vasilis Papavasileiou
Northeastern University
vpap@ccs.neu.edu

***Abstract*—In this position paper, we advocate the use of verification technology to tackle the virtual integration problem for cyber-physical systems. In particular, we advocate the use of high-level modeling languages that allow designers to declaratively specify *what* properties their architectural models should have, not *how* to achieve them. We further advocate the use of verification technology to analyze such models, in particular to synthesize concrete architectural models that certifiably satisfy all properties. We discuss our work in this direction and outline some of the challenges. Specifically, we show that virtual integration in the presence of real-time scheduling constraints leads to problems that cannot be dealt with in a compositional manner. They can, however, be handled in a semi-compositional fashion, as we outline in this paper.**

I. Introduction

The design of cyber-physical systems is an increasingly important problem. Cyber-physical systems use software to interact with and exert control over the physical world. Examples abound in the aerospace, automotive, and medical fields. Cyber-physical systems tend to consist of many interacting, complex components that share resources and operate under real-time constraints. Their design and development often involves large teams working for many years. Design flaws routinely remain undiscovered until integration testing, at which point they are prohibitively expensive to deal with. The idea of virtual integration testing is to analyze models of cyber-physical systems before they are built, in order to find and correct design flaws as soon as possible.

In this position paper, we propose tackling the virtual integration problem for cyber-physical systems by using verification technology. More specifically, we propose using declarative high-level models, from which more traditional, architectural models can be synthesized, using verification technology. The synthesized models will come with a certificate showing that they satisfy their architectural-level properties.

Currently, the highest level models used are *architectural* models. Such models describe the high-level structure of a system, which includes a set of components interacting through their connectors. Several *architecture description languages* (ADLs) have been developed to describe and reason about architectural models. Examples include AADL [5] and ACME [6]. Medvidovic and Taylor have written a survey on architecture description languages [9]. Notice that in an architectural model, one has to explicitly specify component connections.

A point of departure from current methods, is that we insist on the use of a declarative high-level modeling language. The reason is that it is crucially important to have analyzable, formal models as early as possible in the design cycle. The gap between declarative high-level models and architectural models can be significant, representing multiple person-years of effort. We use verification technology to enable designers to automatically synthesize architectural models that certifiably satisfy their design specifications. Not only does our approach relieve the designers from the multi-year effort to define an architectural model, but, more importantly, it allows designers to get immediate feedback on their designs, allowing them to more fully explore the design space, to find design flaws early, and, thereby, to design superior cyber-physical systems.

The rest of the paper is organized as follows. In Section II, we describe the CoBaSA language, a high-level declarative language along the lines we have been proposing. We have used CoBaSA to solve what we call the *system assembly* problem: from a sea of available components, which should be selected and how should they be connected, integrated, and assembled so that the overall system requirements are satisfied? [8] Section II includes a simple, representative example designed to give a sense of what is involved in using CoBaSA to synthesize architectural models. CoBaSA works by transforming the system assembly problem into a satisfiability problem, that is then handled by a verification tool. Why is this not the end of the story? Well, one of CoBaSA's major limitations is that it cannot handle real-time scheduling constraints. In Section III, we describe static cyclic scheduling, which is non-preemptive and quite difficult to satisfy. In Section IV, we show that the combination of the system assembly problem and the scheduling problem cannot be solved in a compositional way, and outline a *semi-compositional* approach to solving the problem. Conclusions and directions for future work are given in Section V.

II. System Assembly

The CoBaSA system, as described in [8], allows us to express architectural constraints between components and automatically synthesize architectural models that satisfy these constraints. CoBaSA provides an object-oriented language

for describing the structural properties of the components, and mechanisms for imposing constraints on how these components will inter-operate. In this section we provide an introduction to the CoBaSA language. To this end, we use as an example an architectural model from the aerospace domain and terms consistent with the ARINC standards 651-1 and 664-7 [1].

The basic datatypes in CoBaSA are integers (bounded or arbitrary precision), Booleans, and strings. CoBaSA *entities* correspond to classes in an object-oriented programming language. Entity inheritance is supported. Data can be organized in single- or multi-dimensional arrays.

Resource allocation is the main objective of the CoBaSA system. We think of the resource allocation problem in terms of mapping resource *consumers* to resource *providers*. CoBaSA objects can consume resources, or provide resources, or both. In our model, resource providers are processors that provide CPU time, RAM and other resources. Processors reside in different locations `"Loc1"`, ... `"Locn"`. Our consumers are *jobs* (or hosted functions) and *global memory spaces* with certain resource requirements. The corresponding CoBaSA entities are:

```
entity processor {
  ; id STRING
  ; location STRING
  ; cpu-time-available 1000000
  ; nvm-memory-available 16384
  ; ram-memory-available 131072
  ; rom-memory-available 65536
  ; no-of-rx-vls-available 384
  ; no-of-tx-vls-available 128
  ; rx-vl-bandwidth-available 100000
  ; tx-vl-bandwidth-available 100000
}

entity job {
  ; id STRING
  ; location STRING
  ; rate INT
  ; cost INT
  ; cpu-time-req INT
  ; nvm-memory-req INT
  ; ram-memory-req INT
  ; rom-memory-req INT
}

entity gms {
  ; id STRING
  ; nvm-memory-req INT
  ; ram-memory-req INT
  ; rom-memory-req INT
}
```

The `processor` entity has an identifier, a string corresponding to the location where the processor resides, and fields for the resources provided: CPU time, different kinds of memory (non-volatile, RAM, and ROM), the number of *virtual links* the processor can receive and transmit, and the bandwidth available for the incoming and outgoing virtual links. Since all processors in this model are homogeneous, all the fields corresponding to resources have default values; thus, we do not have to define them when instantiating the entity.

The `job` entity has fields for the ID, the location of the job (a set of locations, as above), the rate and cost (how many times per second the job is executed, and how long each such execution lasts), the total CPU time requirement (the product of rate and cost), and the different memory requirements.

The entity `gms` has fields for the ID and the memory requirements of each global memory space.

We define instances of the above entities as follows:

```
var
  ; processor P_1 =
      {; "P_1" ; "Loc1" ; ; ; ; ; ; ; ;}
  ; job J_1 =
      {; "J_1" ; "Loc1" ; 20 ; 8750
       ; 175000 ; 64; 8192 ; 4096}
  ; processor[4] processors-Loc1 =
      [P_1, P_2, P_3, P_4]
  ; job[64] jobs-Loc1 =
      [J_1, J_2, ..., J_64]
```

In this example, we defined a processor with `id "P_1"` and location `"Loc1"` that has default values for the remaining fields, and a job with `id J_1` that can be mapped to processors at location `"Loc1"` and has the resource requirements shown. We also defined an array of 4 processors whose location is `"Loc1"`, and an array for the 64 jobs that can be mapped to them. Separate arrays are used for the processors at different locations and the jobs that can be mapped to them. For jobs can be mapped to multiple locations, we have yet another array.

CoBaSA *maps* are functions from resource providers to resource consumers. A map constraint relates two arrays, which we call the *domain* and the *range* of the map. CoBaSA actually supports more expressive syntax, where domains and ranges include a set of arrays of objects, but we will focus on simplicity. Each component in the domain is mapped to exactly one component in the range. The following definition maps jobs to processors:

```
map jobs-to-procs-Loc1
    jobs-Loc1 processors-Loc1
constraint jobs-to-procs-Loc1
  ((cpu-time-req,
    nvm-memory-req,
    ram-memory-req,
```

```
    rom-memory-req))
  ((cpu-time-available,
    nvm-memory-available,
    ram-memory-available,
    rom-memory-available))
```

A map definition is usually accompanied by *field constraints*, which relate resource requirements to resource availability. In our model, we need the above four field constraints for the CPU time and the different kinds of memory. According to the first field constraint, the sum of the CPU time requirements of the jobs mapped to a specific processor cannot exceed the available CPU time in the processor.

The mapping of jobs to processors in the model is subject to requirements like job *separation* (a pair of jobs have to be mapped to different processors). For example:

```
for_all p in processors-Loc1
  {(jobs-to-procs-Loc1(J_7, p)
    implies
    (not jobs-to-procs-Loc1(J_8, p)))}
```

The constraint above states that jobs `J_7` and `J_8` cannot be co-located. The `for_all` statement was used above to apply a constraint to a whole array. *Map references* indicate whether an element in the domain of a map is mapped to an element in the range: `jobs-to-procs-Loc1(J_7, p)` is true if and only if job `J_7` is mapped to processor `p`. The usual logical connectives are provided. A separation constraint as in the example above can be used to deal with job replication requirements. For system reliability reasons, we may need multiple copies of a job running on different processors. We can deal with such constraints by defining multiple jobs, one per instance, with the same values for the resource requirements, and with associated separation constraints.

We similarly encode job *co-location* constraints: such constraints state that a pair of jobs have to be mapped to the same processor.

The designer of a safety-critical system has to consider the possibility of failures. *Spare* processors allow us to operate safely in the presence of a limited number of processor failures. In our architectural model, a set of processors per location are considered spare processors. We are given sets of jobs at most one of which can be mapped to a spare processor. The jobs that do not appear in any of these sets cannot be mapped to a spare processor. In our model, `P_4` is the only spare processor for location `Loc1`. The constraints below state that at most one of the jobs `J_36`, `J_48` and `J_52` can be mapped to a spare processor and that `J_53` cannot be mapped to a spare processor.

```
(+ jobs-to-procs-Loc1(J_36, P_4)
   jobs-to-procs-Loc1(J_48, P_4)
   jobs-to-procs-Loc1(J_52, P_4))
<= 1
(not jobs-to-procs-Loc1(J_53, P_4))
```

Notice that in the example we added map references: in CoBaSA, Booleans are treated as 0 or 1 when used in an arithmetic context.

In some cases, a resource consumer has to be mapped to more than one resource provider. In our architectural model, multiple copies of global memory spaces are allowed. This can happen if multiple jobs that reside on different processors need read-only access to the same memory space. In this case, each copy of the memory space consumes resources from the processor it is mapped to. We define a *generalized map* from memory spaces to processors:

```
gmap >= 1 gmss-copies-to-procs
          gmss-copies procs
constraint gmss-copies-to-procs
  ((nvm-memory-req,
    ram-memory-req,
    rom-memory-req))
  ((nvm-memory-available,
    ram-memory-available,
    rom-memory-available))
```

The map above states that each memory space in the `gmss-copies` array is mapped to at least one processor. We have a separate array `gmss` and an associated map for the global memory spaces that have to be mapped to exactly one processor. In addition to the generalized map, we need constraints that force a copy of the global memory region to be co-located with a job that needs read-only access to it. The map above also illustrates a case in which resource consumers of different kinds (namely, jobs and global memory spaces) share the same resources. CoBaSA handles this correctly.

Other families of constraints that a safety-critical system has to satisfy can be encoded with arbitrary Boolean variables and constraints on them.

The jobs in our architectural model communicate through *virtual links*. Every link has a exactly one publisher, but potentially multiple subscribers. The sum of the bandwidth required for the virtual links that the jobs located on a processor publish or subscribe to cannot exceed the available outgoing or incoming bandwidth of the processor, respectively. In case two or more subscribers of the same virtual link are mapped to a specific processor, we count the incoming bandwidth of the link only once.

We encode the incoming virtual link constraints as follows. We use a Boolean variable to denote the fact that a processor receives a specific virtual link. For example, jobs `J_12`, `J_25` and `J_38` subscribe to the virtual link `VL_1`. We define `P_3-sub-VL_1` to be true if and only if at least one of `J_12`, `J_25` or `J_38` is mapped to processor `P_3`:

```
(P_3-sub-VL_1 iff
```

```
 (jobs-to-procs-Loc1(J_12, P_3) or
  jobs-to-procs-Loc1(J_25, P_3) or
  jobs-to-procs-Loc1(J_38, P_3)))
```

Our architectural model involves 110 virtual links. Each processor has a limit on its incoming bandwidth, which is 100,000 Kbps. The incoming bandwidth requirements of the virtual links `VL_1`, `VL_2` and `VL_110` are 8,000 12,000 and 16,000 Kbps respectively. Thus,

```
(+ (* P_3-sub-VL_1 8000)
   (* P_3-sub-VL_2 12000)
   ...
   (* P_3-sub-VL_110 16000))
<= 100000
```

In our model, virtual links are comprised of messages. Jobs really need access to messages, not virtual links. Therefore, the messages they need determine what virtual links they subscribe to and they read only the subset of messages they care about from the virtual links they subscribe to. Jobs buffer a given number of bytes for each message, a number that can differ among subscribers of the same message. Each processor provides a single buffer for both message transmission and reception. The sum of the buffering requirements for the messages that the jobs mapped to a processor publish or subscribe to cannot exceed the capacity of the buffer. When multiple jobs on the same processor subscribe to the same message with different buffer requirements, the maximum buffer requirement is taken into account. Finally, each hosted function that subscribes to a message uses either a queue buffer, or a sampling buffer for it and hosted functions that subscribe to the same message but use different buffer types cannot reside on the same processor. The CoBaSA framework is expressive enough to accommodate messages with the constraints described above.

The CoBaSA framework is fully automated. The system receives as input a program in the language we have described and returns an allocation of resource consumers to resource providers. The way the framework works is by translating the constraints to a propositional Satisfiability (SAT), pseudo-Boolean Satisfiability or Integer Linear Programming (ILP) problem and calling an off-the-shelf SAT or ILP solver. The generation of constraints involves some novel ideas [2], [7]. The satisfying assignment that the solver returns corresponds to a solution of the system assembly problem. There is no clear winner as the backend solver for system assembly problems: we have seen instances for which SAT is better, pseudo-Boolean SAT or ILP is better. Since we strive to offer a general framework, it is beneficial to support all three.

CoBaSA can also be used in optimization mode, in which we look for a solution that is optimal with regards to an objective function. For example, we can use this feature for load balancing: we can minimize the difference between the minimum load among the processors and the maximum load.

Prior work [3], [4] has explored one- and multi-dimensional bin-packing techniques for tackling the resource allocation problem. However, we are not aware of any methodology that allows automated synthesis of large-scale architectural models like the one described in this section. For example, resources like bandwidth cannot be treated as bin packing dimensions, because multiple co-located jobs subscribing to the same virtual link can share the bandwidth cost. The CoBaSA Boolean and arithmetic operators provide a very general way to express constraints, while resource allocation algorithms based on bin-packing techniques may have to be expanded each time a new class of constraints is considered.

III. Scheduling

In this section we describe static cyclic scheduling, which is non-preemptive and periodic. It is used in industry and it happens to be a very demanding type of scheduling, which has its advantages and disadvantages. The disadvantage is that it is often very hard to determine if even relatively small sets of jobs are static cyclic schedulable on a single processor. The advantage is that static cyclic scheduling provides very strong guarantees, which can dramatically simplify the kinds of safety analyses often required for safety-critical systems.

We now define the static cyclic scheduling problem. Time will be divided into an infinite number of cycles, each of s slots. A slot is the smallest interval of time we will consider. For example, we might divide time into cycles, each of which lasts for a second, consisting of $s = 10^6$ slots. In this case a slot corresponds to a microsecond. The parameter s is really processor-dependent. If we have a collection of heterogeneous processors, then we can account for their different rates of speed by setting their s parameters appropriately.

Jobs will be denoted by a pair (r, c), where r is the rate of the job and c is the cost. We require that for any job (r, c) that is to be scheduled on a processor with s slots per cycle, that r divides s. For example, if a cycle corresponds to a second and $s = 10^6$, as above, then $r = 100$ is allowed, whereas $r = 7$ is not. Given a multiset of jobs, a schedule is simply a starting time (slot) t for each job such that $t < s$ and no two jobs occupy the same slot. The slots occupied by a job are slots of the form $t + k\frac{s}{r} + i$, for k a natural number and $0 \leq i < c$. That is, if a job is scheduled to start during slot t, then it occupies c consecutive slots starting with slot t and this process repeats at slot $t + \frac{s}{r}$, $t + 2\frac{s}{r}$, etc. For example, if a cycle corresponds to a second and $s = 10^6$, as above, and $(r, c) = (100, 5)$, then once the job starts, it has to be scheduled every $\frac{1}{1000}^{th}$ of second for $\frac{5}{10^6}$ consecutive seconds each time.

Given a multiset of jobs, the uniprocessor static cyclic scheduling problem is to determine whether there exists a

schedule. This problem is NP-complete, as is the multiprocessor version of the problem.

IV. Semi-Compositionality

The question we address in this section is: what if we have a model that includes both the kind of system assembly constraints we saw in Section II and the kinds of scheduling constraints we saw in Section III? Do we need a completely new way of dealing with the combination of these constraints, or can we compose the verification algorithms we use for solving the system assembly problem with the scheduling algorithms used for static-cyclic scheduling?

Unfortunately, we cannot simply run one algorithm after the other. To see this, consider the following simple multiset of jobs:

$$\{j_1 = (2, 1), j_2 = (3, 1), j_3 = (3, 1), j_4 = (6, 1)\}$$

where j_1 and j_2 must be co-located and j_2 and j_3 must be separated. In addition we have two processors each with 12 slots. If we solve the system assembly problem, a possible solution is to map j_1, j_2, and j_4 to processor 1 and j_3 to processor 2. Unfortunately, this allocation is not schedulable.

If, instead we try to schedule the jobs first, then we might wind up with j_1 and j_4 on processor 1 and j_2 and j_3 on processor 2. Unfortunately this does not satisfy the system assembly constraints.

What this example shows is that we cannot solve the problem in a compositional way.

This conclusion may seem surprising in the light of previous research results that achieve task resource allocation and scheduling with the same algorithm [3], [4]. However, the scheduling policy in these cases is rate-monotonic scheduling. Rate monotonic schedulability is guaranteed, if the load does not exceed certain bounds. The models we deal with have CPU utilization over 90%, which is well above any limit that guarantees schedulability. Under these circumstances, rate-monotonic scheduling can still be done compositionally in practice. However, static cyclic cannot. Thus it is impossible to treat CPU time as any other resource and a separate decision procedure is required.

The best we can hope for is a *semi-compositional* approach where we use the same algorithms as before for solving the system assembly problem and the scheduling problem. However, the algorithms are now part of a framework that enables them to interact in a way that allows us to solve the combined problem. We are experimenting with various approaches that are motivated by work from the verification community on combining decision procedures.

V. Conclusions and Future Work

We advocated the use of verification technology to tackle the virtual integration problem for cyber-physical systems. The idea is to use high-level modeling languages that allow designers to declaratively specify *what* properties their architectural models should have, not *how* to achieve them. From such high-level models, we proposed to use verification technology to automatically synthesize architectural models that certifiably satisfy the high-level properties. One of the challenges is dealing with virtual integration in the presence of real-time scheduling constraints. While this problem is inherently not compositional, we advocated the use of semi-compositional methods.

Acknowledgment

We are indebted to John Chilenski of Boeing Commercial Airplanes for providing extensive guidance, support, feedback, and collaboration. This research is funded in part by NASA Cooperative Agreement NNX08AE37A.

References

[1] ARINC. ARINC specifications and reports. See https://www.arinc.com/.

[2] B. Chambers, P. Manolios, and D. Vroon. Faster SAT solving with better CNF generation. In *DATE, Design, Automation and Test in Europe*, pages 1590–1595. IEEE, 2009.

[3] D. de Niz and P. H. Feiler. On Resource Allocation in Architectural Models. In *ISORC*, 2008.

[4] B. Dougherty, J. White, J. Balasubramanian, C. Thompson, and D. C. Schmidt. Deployment Automation with BLITZ. In *ICSE*, 2009.

[5] P. H. Feiler, D. P. Gluch, and J. J. Hudak. The Architecture Analysis & Design Language (AADL): An Introduction, 2006.

[6] D. Garlan, R. T. Monroe, and D. Wile. Acme: Architectural Description of Component-Based Systems. In G. T. Leavens and M. Sitaraman, editors, *Foundations of Component-Based Systems*, pages 47–68. Cambridge University Press, 2000.

[7] P. Manolios, S. K. Srinivasan, and D. Vroon. BAT: The Bit-Level Analysis Tool. In *Computer Aided Verification, CAV*, volume 4590 of *Lecture Notes in Computer Science*, pages 303–306. Springer, 2007.

[8] P. Manolios, D. Vroon, and G. Subramanian. Automating component-based system assembly. In *ISSTA*, pages 61–72. ACM, 2007.

[9] N. Medvidovic and R. N. Taylor. A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Transactions on Software Engineering*, 26(1):70–93, 2000.