Synthesizing Cyber-Physical Architectural Models with Real-Time Constraints

Christine Hang, Panagiotis Manolios, and Vasilis Papavasileiou

Northeastern University
christine.d.hang@gmail.com, {pete,vpap}@ccs.neu.edu

Abstract. We present techniques that enable designers to algorithmically synthesize cyber-physical architectural models with real-time constraints. We do this by providing a *meta*-architectural specification language that allows designers to specify *what* properties their architectural models should have, not *how* to achieve them. This provides designers with a qualitatively new level of abstraction that enables the exploration of design spaces at the earliest stages of design, when doing so provides the most benefit. Our key technical contribution is the development of an Integer linear programming Modulo Theories (IMT) solver along with a scheduling theory solver. Our solver was used to automatically synthesize cyber-physical architectural models with hard real-time constraints from a large-scale industrial design.

1 Introduction

The complexity of cyber-physical systems, such as ground, air, space, and sea vehicles, continues to increase at an exponential rate, independent of any reasonable metric used. These systems tend to be distributed and consist of numerous interconnected components that share resources, interact in complex, safety-critical ways, and have real-time constraints. Real-time constraints are particularly important because cyber-physical systems interface with the physical world and have to respond to physical events in real-time; if they do not, collision and loss of life are possible outcomes. The design of such systems is a major challenge, *e.g.*, the verification and validation of critical avionics software is estimated to cost seven times as much as its software development costs.

There is wide consensus that the design of complex systems requires raising the level of discourse by utilizing high-level modeling. The highest-level models commonly used are *architectural* models: they describe the structural properties of components and the connections between components. Many *architecture description languages* (ADLs), such as AADL [8], have been proposed to describe and reason about this structure [12]. Even such high-level modeling languages require users to specify what components are to be used and how they are to be connected. The effort required to do this can be significant, *e.g.*, the current approach used by our industrial partner required several engineering teams closely working together over a considerable amount of time to develop an architectural model for the case study we consider.

In this paper, we revisit synthesis with the goal of enabling designers to work at a higher level of abstraction and to algorithmically synthesize cyber-physical architectural models that are correct by construction. Designers should only specify *what* they want, not *how* to achieve it. The emphasis on *what* not *how* is a departure from current high-level design methods. Such a shift will enable designers to rapidly explore the design space during the earliest stages of design. This is when the benefits of design exploration are greatest, as it is well known that errors tend to become exponentially more expensive to correct the further along the life-cycle they are discovered.

To solve the synthesis problem, in Section 4, we introduce the idea of *Integer linear programming Modulo Theories (IMT)* solvers. An IMT solver resembles an SMT (Satisfiability Modulo Theories) solver, except that instead of using a SAT solver at the core, we use an ILP (Integer Linear Programming) solver. To our knowledge, this is the first time that the combination of ILP with background theories appears in the literature. We believe that the IMT approach has the potential to be widely applicable, as many practical problems from Operations Research and Engineering are routinely expressed using mathematical programming tools such as CPLEX. If one wants to consider the combination of such problems with other theories, then the IMT approach has the potential to enable the kinds of advances enabled in verification by SMT.

For cyber-physical systems, dealing with real-time constraints is of paramount importance. We consider static cyclic scheduling, an industrially-relevant and particularly demanding type of real-time scheduling problem in Section 3. We show how to solve multi-processor static-cyclic scheduling constraints using a theory solver that can be used in our IMT approach and which is parameterized by a decision procedure for the uniprocessor static-cyclic problem. Not surprisingly, care is needed in the generation of theory lemmas, as discussed in section 5. With synthesis problems, the expectation is that constraints are satisfiable. In Section 6 we show how to take advantage of this by using a general-purpose resource limit mechanism that tends to bias the solver towards parts of the state space that are easier to reason about, often leading to dramatic performance improvements.

We successfully implemented and used our approach on an industrial case study from a very complex state-of-the-art aerospace design provided to us by our industrial partners. Section 2 presents a high-level overview of the class of non-scheduling constraints appearing in our case study and how they are modeled in CoBaSA, our modeling language. CoBaSA and how it solves architectural synthesis problems that do not include scheduling constraints was introduced in previous work [11, 10]. In section 8, we experimentally evaluate our work using the above mentioned case study. We were able to quickly and fully automatically synthesize cyber-physical architectural models with real time constraints for very complex aerospace designs.

2 Models and Constraints

In this section, we describe the *system assembly* constraints, the class of non-scheduling constraints found in our case study, and how they are modeled in CoBaSA [11], our modeling language. The case study is based on a real, production design provided by an industrial partner in the aerospace domain. We use terms consistent with the ARINC standards 651-1 and 664-7 [1]. A more detailed description of the constraints is also available [10].

We considered a number of models during the course of several years. The basic components of the models included: anywhere from 8 to 22 *cabinets* (cabinets provide various resources such as processors and battery backup units), anywhere from 177 to 257 *applications* (we also refer to applications as *hosted functions* or *jobs*), and anywhere from 70 to 288 *global memory spaces* (GMSs) (GMSs allow applications to share memory). Other components include *messages* (for communicating between applications, sensor, and other components) and *virtual links* (virtual links are part of a publish-subscribe network and are used to aggregate and multicast messages). The models had between 1,000 and 2,000 virtual links and between 10,000 and 20,000 messages.

Resource Utilization: Cabinets provide various resources, including CPU time, RAM memory, ROM memory, non-volatile memory, buffers, and send and receive bandwidth for virtual links. Cabinets also have limits on how many virtual links they can receive and transmit. Hosted functions, global memory spaces, virtual links, and messages consume these resources. Our meta models include constraints stating that the sum of any resource used does not exceed the amount of the resource that we have available.

Hosted Function Allocation: Hosted functions have to be mapped to cabinets subject to the resource utilization constraints above, but we also have to satisfy constraints of the following types. (a) *Fixed cabinet constraints* specify that a particular hosted function has to be mapped to a particular cabinet. (b) *Separation constraints* state that no pair of hosted functions in a given set can reside on the same cabinet. (c) *Co-location constraints* state that given a non-empty sequence of non-empty sets of hosted functions, we have to create m groups, where m is the maximum of the cardinalities of the sets in the sequence. Furthermore, each hosted function in a set has to be assigned to one of the m groups, no two hosted functions in the same set can be assigned to the same group, and all hosted functions in a group have to be assigned to the same cabinet.

Spare Cabinets and Hosted Functions: Spare cabinets allow us to operate safely in the presence of a small number of cabinet failures. To that end, *spare* cabinets are only allowed to run *spare* hosted functions. We are given a non-empty set of hosted functions and are allowed to map at most one hosted function from the set to a spare cabinet. Hosted functions that do not appear in such constraints cannot be mapped to spare cabinets. If a non-spare cabinet fails, the idea is to migrate its jobs to a spare cabinet.

Global Memory Spaces and Hosted Functions: Constraints between GMSs and hosted functions include. (a) *Fixed cabinet constraints* specify that a par-

ticular global memory space has to be mapped to a particular cabinet. (b) *Co-location constraints* specify that a particular global memory space and all hosted functions in a given non-empty set have to be mapped to the same cabinet. (c) *Read-only constraints* specify that a particular global memory space has to be allocated to all of the cabinets that a given set of hosted functions map to. Note that read-only GMSs can be arbitrarily replicated.

Virtual links: Hosted functions publish and subscribe to virtual links. Virtual links have exactly one publisher, but can have multiple subscribers. The sum of the bandwidth required for the virtual links that the hosted functions located on a cabinet publish or subscribe to cannot exceed the available outgoing or incoming bandwidth, respectively. For the incoming bandwidth constraints, if multiple hosted functions located on the same cabinet subscribe to the same virtual link, then the cost of the virtual link is only counted once.

Message buffering: A virtual link is comprised of a non-empty set of messages. Hosted functions are only really interested in messages, not virtual links. Therefore, they only read the messages they care about from the virtual links they subscribe to. Hosted functions buffer a given number of bytes for each message; this can differ among subscribers of the same message. Each cabinet provides a single buffer that is used for both message transmission and reception. The sum of the buffering requirements for the messages that the hosted functions located in the cabinet publish or subscribe to cannot exceed the capacity of the buffer. When multiple hosted functions on the same cabinet subscribe to the same message with different buffer requirements, they share space, so only incur the cost of the maximum number of bytes buffered. 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 cabinet.

Objective Functions: Our framework also allows for objective functions, which we have used for load balancing, minimizing the maximum bandwidth used per cabinet, etc.

Figure 1 shows simplified snippets of CoBaSA code for modeling a small subset of the constraints. CoBaSA includes an object-oriented modeling language, *e.g.*, the cabinet *entity* above can be thought of a *class* with three fields, where the last two correspond to resources and have default values. `C_1` is an instance of the cabinet entity, and `cabs` is an array of cabinets. Similarly, we define an array of jobs, `jobs` (though the entity definition is not shown).

CoBaSA also includes a declarative language for describing constraints. For example the line starting with `map` defines a map, `jobs-to-cabs` from `jobs` to `cabs`. The next five lines constrain `jobs-to-cabs` by requiring that cabinets have enough CPU and RAM resources to satisfy all jobs mapped to them.

The constraint starting with `for_all` is a separation constraint: it states that jobs `J_1` and `J_2` have to reside on different cabinets. The example demonstrates the high-level, declarative way in which we describe the system. We specify only *what* properties our model should have, not *how* to connect jobs and cabinets to achieve these properties. Figure 2 visualizes one of the solutions we synthesized.

```
entity cab {
  ; id STRING
  ; cpu-time-avail 1000000
  ; ram-memory-avail 4294967296
}

var
  ; cab C_1 = { ; "C_1"; ; }
  ; cab[12] cabs =
      [C_1, C_2, ..., C_12]
  ; job[200] jobs =
      [J_1, J_2, ..., J_200]
```

```
// mapping jobs to cabinets
map jobs-to-cabs jobs cabs
constraint jobs-to-cabs
  ((cpu-time-req,
    ram-memory-req))
  ((cpu-time-avail,
    ram-memory-avail))

// jobs J_1 and J_2 separated
for_all c in cabs
  { jobs-to-cabs(J_1, c) implies
    (not jobs-to-cabs(J_2, c)) }
```

Fig. 1. CoBaSA Modeling Language Examples

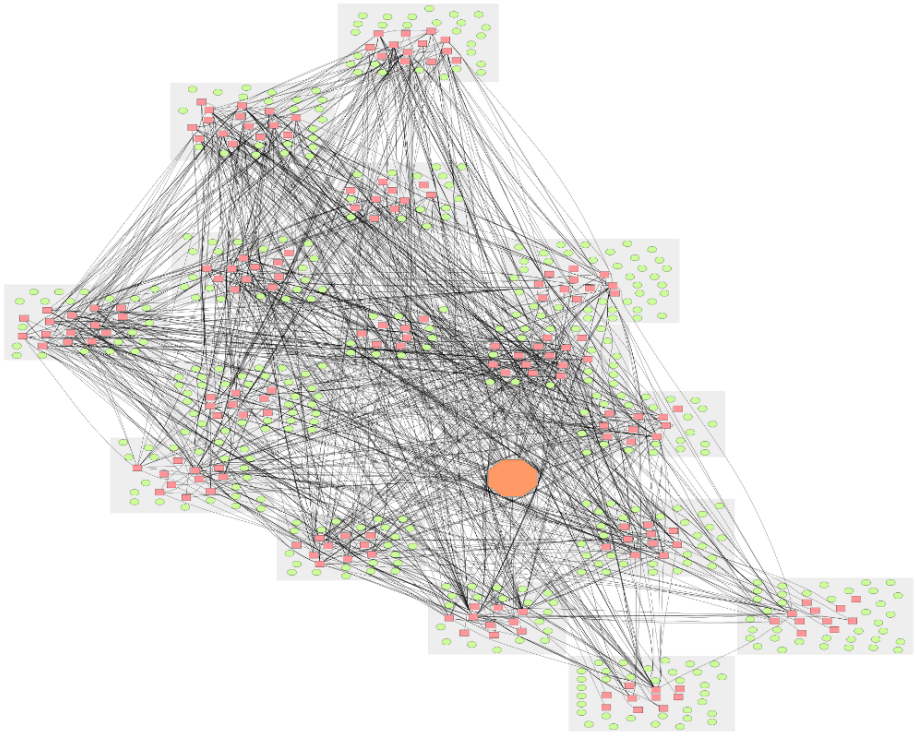

Fig. 2. One of the architectural models we synthesized, visualized as a graph. Pink rectangles correspond to hosted functions, and green ovals to memory spaces. The grey containers are cabinets. Edges between hosted functions visualize communication through virtual links. The orange octagon is the external network.

We end this section by noting that many other cyber-physical systems (*e.g.*, consider the automotive industry) will have similar types of constraints, thus, we expect that our approach will be applicable to these systems as well.

3 Static Cyclic Scheduling

While our approach is independent of the types of real-time scheduling constraints used, in this paper, we consider static cyclic scheduling constraints. Static cyclic scheduling is non-preemptive and periodic; it is easy to describe, difficult to satisfy, and is used in industry.

We first describe uniprocessor static cyclic scheduling (USCS). Time is divided into infinitely many slots. A job is defined as a triple (p, c, i). The period, p, is the number of slots between successive executions of the job. The cost, c, is the number of slots each execution takes. The identifier, i, is a natural number that uniquely identifies the job. Given a set of jobs, a schedule is simply a starting time t for each job such that $t < p$ and no two jobs occupy the same slot. The slots occupied by a job are slots of the form $t + k \cdot p + m$, for $k \in \mathbb{N}$ and $0 \leq m < c$. 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 $t + p$, $t + 2 \cdot p$, ...

Given a set of jobs, the USCS problem is to determine whether there exists a schedule.

Theorem 1. *The USCS problem is NP-complete.*

Proof. We reduce the NP-complete *set partition problem* to the USCS problem. Given a multiset $S = \{n_1, n_2, \ldots, n_k\}$ of natural numbers, we construct a set of jobs

$$J = \{(s, n_i, i) \mid 1 \leq i \leq k\} \cup \{(s/2, 1, k + 1)\}$$

where $s = 2 + \sum S$. It is not hard to see that S can be partitioned into S_1 and S_2 such that $\sum S_1 = \sum S_2 = (\sum S)/2$ iff J is schedulable. To see that USCS is in NP, notice that we can verify a schedule by checking than no pair of jobs leads to a collision. With a little bit of analysis, this can be done by checking for all pairs of jobs (p_1, c_1, i_1) and (p_2, c_2, i_2) starting at t_1 and t_2 respectively that if $i_1 \neq i_2$ then $t_1 \neq t_2$, so without loss of generality assume $t_1 < t_2$, and we have $c_1 \leq t_2 - t_1 \leq \gcd(p_1, p_2) - c_2$.

In Multiprocessor Static Cyclic Scheduling (MSCS), we have multiple processors that run in different speeds. The number of slots per cycle (*e.g.*, per second) is processor-dependent; a processor is defined as a pair (s, i) where $s \in \mathbb{N}$ is the number of slots that the processor provides per cycle and $i \in \mathbb{N}$ is an identifier unique among processors. In MSCS, we denote a job by a triple (r, c, i), where r is the rate of the job, c is the cost, and i is the identifier. The period of a job depends on the processor: a job (r, c, i) has period s/r on a processor that provides s slots per cycle. We assume that for each job and each processor, the rate of the job r divides the number of slots per cycle s that the processor provides.

Given a set of jobs, J, and a set of processors, P, the multiprocessor static cyclic scheduling (MSCS) problem is to determine whether there exists a mapping from J to P such that for each processor, the set of jobs mapped to that processor are schedulable. As USCS is a special case of MSCS, the following corollary follows trivially:

Corollary 1. *The MSCS problem is NP-complete.*

We conclude this section by noting that given a static cyclic schedule, we can statically determine the exact slots allocated to any job. On the one hand, this makes static cyclic scheduling hard.[1] On the other hand, this makes the schedule very predictable, which in turn dramatically simplifies the analysis of the system as a whole. This advantage is why static-cyclic scheduling is used for complex cyber-physical systems.

4 ILP Modulo Theories

Our models include both system assembly constraints (as described in section 2) and static cyclic scheduling constraints (as described in section 3). We cannot simply run the ILP solver to assemble the components, and then run the scheduler, as the solution provided by the ILP solver might not be schedulable even if a solution to the system assembly and scheduling constraints exists. To solve this problem, we develop an *ILP Modulo Theories (IMT)* solver that combines a background decision procedure with an ILP solver in much the same way that SMT allows for the integration of theory solvers with SAT solvers [2, 4, 14].

We explored SAT as an alternative to ILP for our class of problems. We tried different encodings of the system assembly constraints into SAT and different SAT solvers. Performance was always at least three orders of magnitude worse than using an ILP solver. We conjecture that the reason for this is that our resource constraints are heavily arithmetic. Therefore, in this paper, we assume an ILP solver as our core solver, but note that our work can also be used in an SMT framework or with a pseudo-Boolean core solver.

4.1 Formal Preliminaries

Let J be a set of jobs and P be a set of processors. Let $V_{J,P} = \{V_{(j,p)} \mid j \in J, p \in P\}$ be a set of propositional variables. Note that we can represent any mapping, M, from J to P as an assignment to $V_{J,P}$. The variable $V_{(j,p)}$ is *true* iff M maps j to p. The variables in $V_{J,P}$ are called map variables.

Definition 1. *Let J be a set of jobs, P a set of processors, $V_{J,P}$ a set of map variables, and A an assignment to the variables in $V_{J,P}$. We say that the assignment A is consistent with respect to a scheduling theory iff*

$$\left\langle \forall p \in P :: \mathbf{sched}(\{j \in J \mid A(V_{(j,p)}) = \mathit{true}\}, p) \right\rangle$$

[1] For example, for many scheduling problems, if the processor utilization is less than some constant, then there exists a polynomial-time algorithm that is guaranteed to find a schedule; alas this is not true for static cyclic scheduling.

where $\mathbf{sched}(J, p)$ *is a predicate that evaluates to true iff* J *is schedulable on* p.

Let F be an ILP formula (a conjunction of linear equalities and inequalities) over a set of integer and Boolean variables V_F, such that $V_F \supseteq V_{J,P}$, where $V_{J,P}$ is a set of map variables for a set of jobs J and processors P, as described above. We say that an assignment A T-entails $G = F \wedge \mathit{Schedulable}(J, P, V_{J,P})$ (written as $A \models_T G$) iff A satisfies the linear constraints of F, and A is consistent with respect to the scheduling theory. We say G is T-satisfiable iff there is an assignment A such that $A \models_T G$. The problem of determining whether G is T-satisfiable is called the ILP Modulo (Scheduling) Theory problem.

Note that an IMT solver allows us to tackle the MSCS problem as described in section 3. In this case, the formula F has constraints over $V_{J,P}$ that force each job to be mapped to exactly one processor.

4.2 Lazy IMT Approach

Our IMT solver (algorithm 1) involves an ILP solver and a specialized decision procedure for scheduling. The ILP solver suggests assignments which are checked by the scheduler. Each failed attempt to produce a schedule from an assignment results in the learning of some theory lemmas. In analogy to [14], we call this approach the lazy IMT approach, because it learns lemmas only when necessary.

We did try to express the scheduling and system assembly constraints as a monolithic problem, thus following a more eager approach. The time required by the solver was unreasonable. We believe that the clean separation between the real-time and the architectural constraints was one of the reasons why we were successful: a specialized solver works best for the former, an ILP encoding for the latter, and IMT allows for efficient coordination between the two.

Algorithm 1 The IMT algorithm

```
1: procedure IMT-TOP-LEVEL(J, P, F)
2:     Lemmas ← true
3:     while true do
4:         Ans ← ILPSOLVER(F ∧ L)
5:         if Ans = UNSAT then
6:             return UNSAT
7:         else
8:             A ← Assignment from Ans
9:             Schedulable ← true
10:            for all p ∈ P do
11:                J_p ← {j ∈ J | A(V_(j,p)) = true}
12:                if ¬sched(J_p, p) then
13:                    Lemmas ← Lemmas ∧ LEARN(J_p)
14:                    Schedulable ← false
15:            if Schedulable then
16:                return Ans
```

5 Theory Lemmas

To prevent the ILP solver from wasting its time on assignments that will not lead to valid schedules, we generate theory lemmas that rule out as many assignments as possible. Observe that when a set of jobs J is unschedulable on some processor p, it is possible that some strict subset J' of J is also unschedulable on p. By identifying such subsets, we can generate "good" theory lemmas that preclude the allocation of any set of jobs that are "at least as hard" as J' on processors that are "at most as powerful" as p.

5.1 Unschedulable Cores

Given an unschedulable set of jobs J, we identify the unschedulable subsets of J that do not contain any unschedulable proper subset themselves. We call such subsets unschedulable cores.

Definition 2. *C is an unschedulable core of a set of jobs J, if (a) $C \subseteq J$, (b) C is unschedulable, and (c) every proper subset of C is schedulable.*

Property 1. Let J be an unschedulable set of jobs. $\forall j \in J$, if $J \setminus \{j\}$ is schedulable, then j must be in every unschedulable core of J.

Given a set of jobs, J, that has been found to be unschedulable on processor p, the recursive function ALL-CORES(J, p) shown in algorithm 2 returns the set of all unschedulable cores. To acquire all cores, we consider all possible ways of removing jobs from J. Some jobs will be in any unschedulable core of J by property 1, so we only try to remove the remaining jobs (set D). If none of the jobs in J can be removed, then J is an unschedulable core. Otherwise we recurse on all sets $J \setminus \{d\}$ for $d \in D$.

The algorithm can be thought of as performing search on a tree: if a node corresponds to a set of jobs J, the children of the node are the $|J|$ subsets of J with cardinality $|J| - 1$ (one of the jobs removed). Note that instead of finding all possible cores we can terminate the search after finding some given number of cores.

We can make algorithm 2 more efficient by using a DAG instead of a tree, since the tree may have distinct nodes annotated with the same set. In addition, jobs with the same rate and cost can be aggregated and handled separately. Space constraints prohibit a full accounting.

5.2 Subsumption

Given an unschedulable core, we want to introduce a lemma that precludes the co-location of sets of jobs "harder" than the core. To this end, we need to formalize the notion of "hard" with respect to jobs and sets thereof, and the notion of "powerful" with respect to processors.

Definition 3 (Measurement of Difficulty for Jobs). *$\preceq_J$ is a partial order on jobs: $(r_1, c_1, i_1) \preceq_J (r_2, c_2, i_2)$ iff r_1 divides r_2 and $c_1 \leq c_2$.*

Algorithm 2 Finding all unschedulable cores

```
1: procedure ALL-CORES(J, p)
2:     D ← {j ∈ J | ¬sched(J \ {j}, p)}
3:     if D = ∅ then
4:         return {J}
5:     else
6:         A ← ∅
7:         for all d ∈ D do
8:             A ← A ∪ ALL-CORES(J \ {d}, p)
9:         return A
```

Definition 4 (Measurement of Difficulty for Sets). *$\preceq_S$ is a partial order on sets of jobs: $S \preceq_S T$ iff $|S| \leq |T|$ and there exists an injective mapping F from S to T such that $\forall j \in S, j \preceq_J F(j)$. We say that T is* subsumed *by S.*

Intuitively, given an unschedulable set of jobs J, if we replace each job j in J with a job j' such that $j \preceq_J j'$ (j' is "at least as hard" as j) then the resulting set of jobs $(J \setminus \{j\}) \cup \{j'\}$ will also be unschedulable. Similarly, given a set of jobs J which is schedulable, if we replace each job j in J with a job j' such that $j' \preceq_J j$, then the resulting set of jobs will be schedulable as well. Thus, the following property holds:

Property 2. For sets of jobs S, T such that $S \preceq_S T$, (a) if S is unschedulable, then T is unschedulable, and (b) if T is schedulable then S is schedulable.

Definition 5 (Measurement of Power for Processors). *$\preceq_P$ is a partial order on processors: $(s_1, i_1) \preceq_P (s_2, i_2)$ iff s_1 divides s_2.*

5.3 Lemma Generation

Given a set of jobs, J, and a set of processors, P, $\forall j \in J, \forall p \in P$, let the map variable $V_{(j,p)}$ denote that job j is allocated to processor p. Let $C = \{h_1, ..., h_m\}$ be an unschedulable core on some processor $q \in P$. We generate theory lemmas for each processor $p \in P$ where $p \preceq_P q$.

Non-Exhaustive Lemmas

For each processor p where $p \preceq_P q$, and for each job $h_i \in C$ we construct a *bucket* $B_{(i,p)}$ that contains jobs that are "at least as hard" as h_i: $B_{(i,p)} \subseteq \{j \in J \mid h_i \preceq_J j\}$. The buckets do not overlap ($B_{(i,p)} \cap B_{(j,p)} = \emptyset$ for $i \neq j$). Each job j that can be mapped to processor p[2] and is harder than one of the jobs h_i ($h_i \preceq_J j$) has to be included in one of the buckets. If we replace any $h_i \in C$ with a job in the corresponding bucket, we get an unschedulable set of jobs. We

[2] If we can deduce that j cannot be mapped to p, we record this fact and use it to determine which jobs can be mapped to which processors.

generate the lemma $\neg \bigwedge_{1 \leq i \leq m} \left(\bigvee_{j \in B_{(i,p)}} V_{(j,p)} \right)$, which states that if we allocate at least one job from each bucket to processor p, p will be unschedulable.

We attempt to construct buckets in a way that gives rise to useful lemmas. Whenever a job j can go to multiple buckets, we choose (1) the bucket corresponding to the job itself, if the job is in the core; otherwise (2) randomly among the buckets corresponding to jobs that have the same rate and cost as j; if there are no such jobs, (3) among the buckets corresponding to jobs that have the same rate as j; if there are no such buckets, (4) among the remaining buckets. The rationale behind these choices is to ensure that we rule out the allocation of the exact unschedulable set of jobs (and sets very similar to it) to any processor. This guarantees that we make progress towards a solution and ensures termination.

Notice that the lemmas we generate are *non-exhaustive*. It is possible that an assignment A maps a set of jobs S such that $C \preceq_S S$ to a processor p such that $p \preceq_P q$, and A is consistent with our lemmas. This is because some of the jobs of S could have gone to multiple buckets, but our choices when building the lemma resulted in a bucket not being "inhabited" by any $j \in S$. We use non-exhaustive lemmas because their encoding is small.

Exhaustive Lemmas

We can also construct buckets that include the map variables for *all* jobs that are "at least as hard" as a job $h_i \in C$ and can be mapped to processor p: $B'_{(i,p)} = \{j \in J \mid h_i \preceq_J j\}$. Now a job can be in more than one bucket. For each set of jobs $\{j_1 \in B'_{(1,p)}, j_2 \in B'_{(2,p)}, \ldots, j_m \in B'_{(m,p)}\}$ such that $\forall i, k, 1 \leq i < k \leq m : j_i \neq j_k$ we can introduce the clause $\bigvee_{1 \leq i \leq m} \neg V_{(j_i,p)}$. If all j_i were mapped to a processor p such that $p \preceq_P q$, we would have allocated a set harder than the core C, which is therefore unschedulable.

With this encoding, we need $\prod_{1 \leq i \leq m} |B'_{(i,p)}|$ clauses in the worst case. The lemma allows us to rule out *all* sets of jobs S such that $C \preceq_S S$. We use exhaustive lemmas for cores below a certain size, because smaller cores are more frequently applicable and the product above remains manageable.

The recursive function EXHAUSTIVE-LEMMA (algorithm 3) generates all clauses for a list of buckets B' and a processor p. The argument c is a partial clause (initially empty) that corresponds to finding a job j_k for each bucket $B'_{(k,p)}$, where $1 \leq k \leq |c|$; c it is of the form $\bigvee_{1 \leq k \leq |c|} \neg V_{(j_k,p)}$. We use suffixes that the clauses share to keep the encoding more compact. Assume that we have constructed a partial clause c of length l, and that the jobs $j, j' \in B'_{(l+1,p)}$ do not appear in any subsequent bucket. The clauses with prefixes $c \vee \neg V_{(j,p)}$ and $c \vee \neg V_{(j',p)}$ share the suffixes corresponding to the buckets $B'_{(i,p)}$, where $i > l+1$. We use an auxiliary variable for the disjunction $V_{(j,p)} \vee V_{(j',p)}$, instead of generating a separate set of clauses for each of j, j'.

Algorithm 3 Exhaustive lemma generation

```
1: procedure EXHAUSTIVE-LEMMA(p, c, B′)
2:     if B′ = ∅ then
3:         output clause c
4:     else
5:         L ← {j ∈ first(B′) | ∀b ∈ rest(B′), j ∉ b}
6:         if L ≠ ∅ then
7:             v ← ⋁_{j∈L} V_(j,p)
8:             EXHAUSTIVE-LEMMA(p, c ∨ ¬v, rest(B′))
9:         for all j ∈ (first(B′) \ L) do
10:            B′_new ← {b \ {j} | b ∈ rest(B′)}
11:            EXHAUSTIVE-LEMMA(p, c ∨ ¬V_(j,p), B′_new)
```

5.4 Memoization

The purpose of memoization is to avoid making expensive calls to the scheduler when the (un)schedulability of a set of jobs can be inferred from the result of a previous call. For sets S and T such that $S \preceq_S T$, if we have memoized that S is unschedulable, T is also unschedulable by property 2 and we don't have to call the scheduler. Similarly, if we know that T is schedulable we can immediately infer that S is schedulable. The result of each call to the scheduler is memoized in a list as a pair containing the set of jobs on which it was called and the corresponding (un)schedulability. Note that we can decide whether $S \preceq_S T$ in polynomial time by reducing the problem to matching in a bipartite graph.

When the (un)schedulability of a set of jobs is questioned, we access the memoization list sequentially from its head until we find an element from which we can infer (un)schedulability. To speed this operation up we eliminate redundant elements: if the (un)schedulability of some set S in the memoization list becomes inferable from the result of a new call to the scheduler for some set T, then we memoize the (un)schedulability of T, and also remove S from the memoization list. In addition, we keep sets ordered by their success rate (number of successful inferences), so that the most frequently used sets are towards the beginning of the list. If the list becomes too long, we can forget the least useful sets.

The memoization list is complementary to lemmas. For example, the memoization list allows us to quickly infer that a set of jobs is *schedulable*, whereas lemmas only rule out certain assignments. Even for unschedulable assignments the memoization list can complement lemmas. For example, (1) we can infer unschedulability before a lemma is introduced, say during core generation (algorithm 2) and (2) we catch instances subsumed by a core but not caught by the corresponding non-exhaustive lemma.

6 Resource Limits

It is possible that a query to the theory solver takes a long time to complete or uses too much of some other resource (like memory). Since we expect that our synthesis problems have solutions, it make sense to avoid such queries and to instead bias our solver towards solutions that are easier to justify. To that end, we introduce *resource-limit* lemmas that allow us to quickly rule out difficult-to-solve (but potentially satisfiable) scheduling instances in the hope that the ILP solver will find instances that require fewer resources to justify. In order to maintain completeness, we have to sometimes undo resource-limit lemmas to prevent the ILP solver from becoming over-constrained. When resource-limit lemmas are undone, we allow the exploration of previously blocked parts of the search space with increased resource limits. Due to space limitations, we informally describe how resource-limit lemmas work.

We set resource limits for both the scheduler and the ILP solver. In our class of problems, it makes the most sense to restrict time, since that is the bottleneck. If the scheduler times out on some instance, we consider that instance to be unschedulable and generate lemmas to prevent the ILP solver from generating similar scheduling problems. These lemmas are called resource-limit lemmas. They are used in the same way as regular lemmas during ILP solving and will be kept as long as the resulting ILP problem is satisfiable. However, when the ILP problem becomes unsatisfiable, resource-limit lemmas will be removed and the resource limit for the scheduler will be increased. The rationale behind this is that the previous resource-limit lemmas might have over-constrained the search space of the ILP solver and might have led to unsatisfiability. Therefore, removing the lemmas and increasing the resource limit will give the ILP solver a chance to explore a potentially bigger search space. When the ILP solver times out, not only will we remove resource-limit lemmas and increase the resource limit for the scheduler, but we will also increase the resource limit for the ILP solver.

The idea of resource-limit lemmas can also be used in the context of SMT as a technique to manage the balance of resources used by the SAT solver and theory solvers. The technique is likely to be useful in applications like synthesis, where we expect a satisfying assignment to exist. In this case, steering the core solver towards an area of the search space where we can find solutions and justify them easily makes sense. In contexts where we expect the problem to be unsatisfiable, the technique may be counterproductive since resource limits will eventually have to be made large enough to fully explore the search space. On the other hand, purging resource-limit lemmas is somewhat similar to a restart and may exhibit similar benefits.

Note that we can use resource-limit lemmas selectively. For example, if the last round of scheduling attempts led to the discovery of regular lemmas, we can decide to not use any resource-limit lemmas. In this way we make progress (new regular lemmas), but do not needlessly constrain the ILP (or SAT) problem with resource-limit lemmas. Other options for selectively using resource-limit lemmas include filtering the lemmas or using some small number of them per round.

7 Related Work

Architecture Description Languages (ADLs), such as AADL, can be used to model and reason about safety-critical systems [12, 8]. There has been work on ADLs that takes scheduling into account. However, these approaches check that a particular architectural model satisfies scheduling and other constraints [6]. In contrast, we *synthesize* the architectural models, and they are correct by construction.

Different kinds of real-time constraints have been studied. Liu and Layland [9] laid out the basis for the real-time scheduling theory by studying the Rate-Monotonic (RM) and Earliest Deadline First (EDF) algorithms. Sha et al. [17] provide a historical overview of the topic. In contrast to these kinds of scheduling, static cyclic is non-preemptive. In addition, Liu and Layland proved that if utilization is below a specific bound then a rate-monotonic schedule exists. Thus, there is a simple schedulability test for RM. This is not the case for static cyclic scheduling.

There is recent work on allocating jobs to processors with using multi-dimensional bin-packing algorithms, in the presence of scheduling, and other constraints [5, 7]. The scheduling policy in these cases is rate-monotonic. Under restrictions on CPU utilization, rate-monotonic schedulability is ensured. Therefore, such scheduling problems can be turned into bin-packing problems. Unfortunately, this approach is not complete, *e.g.*, there are schedulable problems for which this approach will not find solutions. In addition, the approach does not work for static-cyclic scheduling. Finally the approach is too restrictive to express the constraints we need and has not been shown to scale to the complexity of designs our work handles.

There are many task allocation algorithms for distributed real-time systems that have been studied. This includes, but is not limited to, branch and bound algorithms [15], SAT solving [13], and linear programming [3]. However, the scheduling constraints studied in these cases are not as demanding as static cyclic, and the class of constraints that can be handled is limited.

8 Experimental Evaluation

We summarize our experiences in synthesizing architectural models from the constraints provided by our industrial partner. We only focus on the most complex problem we considered.

Our framework can be parameterized in many ways. We used a 0.1-second CPU resource-limit for the scheduler and, in the presence of resource-limit lemmas, a 1 minute CPU resource-limit for the core solver. If either the scheduler or the core solver time out, then we increase the CPU limit by 40%. We specified a minimum load of 80% per processor, and we limited the search for cores to one per unschedulable processor. If the scheduler timed out while trying to determine the schedulability of a particular processor, we only generated resource-limit lemmas when we failed to extract regular cores from any processor.

We ran our framework with both CPLEX and bsolo [16] as the core ILP solvers. The experiments were run on an eight-core, 3.2 GHz Intel Xeon EM64T server with 96GB of memory (we never needed more than 1GB of memory for any experiment). Our decision procedure for scheduling and the IMT solver were implemented in OCaml. With the set of parameters described above, CPLEX provided a solution after 103 seconds and required 13 iterations. bsolo required 1834 seconds and 23 iterations.

To evaluate the importance of resource-limits, we also ran the above experiment with resource-limits disabled. Both CPLEX and bsolo fail to provide an answer after two hours. CPLEX goes through 5 iterations and bsolo goes through 4 iterations. Analysis shows that the reason for this failure is that we spend lots of time trying to determine the schedulability of processors. The resource-limit mechanism works because it steers CoBaSA towards parts of the search space with easier scheduling problems. We get the same failures (for both CPLEX and bsolo) if we only disable scheduler resource-limits. On the other hand, if we only disable solver resource-limits, then this has no effect on CPLEX (since it does not timeout), but bsolo goes through 22 iterations and times out.

CoBaSA interacts with a collection of tools that our group has developed and with off-the-shelf solvers. In order to increase our confidence in the validity of the solutions CoBaSA generates, we implemented an independent checker that validates solutions. The independent checker helped us identify several bugs in our handling of constraints. Also, our industrial partner checks our solutions in several independent ways. This has also been useful because there were cases where we received incorrect specifications and constraints, or there was miscommunication between different groups.

The experiments show that our approach is capable of synthesizing industrial-scale cyber-physical architectural models with real-time constraints. According to our industrial partner, the current process by which architectural models are created requires significant iteration between multiple engineering teams. Our experiment evaluation clearly shows that our IMT approach leads to significant performance and cost improvements.

9 Conclusions and Future Work

We showed how to algorithmically synthesize cyber-physical architectural models by using a *meta*-architectural specification language that allows designers to specify *what* constraints their architectural models must satisfy, not *how* to achieve them. We did this by developing an ILP Modulo Theories (IMT) solver with a resource-limit capability and a theory solver for static cyclic scheduling. We successfully implemented and used our approach on an industrial case study from a very complex state-of-the-art aerospace design. We believe that the IMT approach has the potential to be widely applicable, as many practical problems are routinely handled using ILP solvers, and the IMT approach allows one to combine the power of ILP with specialized solvers for background theories. For future work, we plan to further develop and explore the IMT approach.

Acknowledgments

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

References

1. ARINC. ARINC Specifications and Reports. See https://www.arinc.com/.
2. C. Barrett, D. Dill, and A. Stump. Checking satisfiability of first-order formulas by incremental translation to SAT. In *CAV*, 2002.
3. H. Cambazard, P.-E. Hladik, A.-M. Deplanche, N. Jussien, and Y. Trinquet. Decomposition and Learning for a Hard Real Time Task Allocation Problem. In *CP*, 2004.
4. L. de Moura and H. Ruess. Lemmas on Demand for Satisfiability Solvers. In *SAT*, 2002.
5. D. de Niz and P. H. Feiler. On Resource Allocation in Architectural Models. In *ISORC*, 2008.
6. J. Delange, L. Pautet, A. Plantec, M. Kerboeuf, F. Singhoff, and F. Kordon. Validate, simulate, and implement ARINC653 systems using the AADL. In *SIGAda*, 2009.
7. B. Dougherty, J. White, J. Balasubramanian, C. Thompson, and D. C. Schmidt. Deployment Automation with BLITZ. In *ICSE*, 2009.
8. P. H. Feiler, D. P. Gluch, and J. J. Hudak. The Architecture Analysis & Design Language (AADL): An Introduction, 2006.
9. C. L. Liu and J. W. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *JACM*, 20(1):46–61, 1973.
10. P. Manolios and V. Papavasileiou. Virtual Integration of Cyber-Physical Systems by Verification. In *AVICPS*, 2010.
11. P. Manolios, D. Vroon, and G. Subramanian. Automating component-based system assembly. In *ISSTA*, 2007.
12. 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.
13. A. Metzner and C. Herde. RTSAT – An Optimal and Efficient Approach to the Task Allocation Problem in Distributed Architectures. In *RTSS*, 2006.
14. R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Solving SAT and SAT modulo theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T). *JACM*, 53(6):937–977, 2006.
15. D.-T. Peng, K. Shin, and T. Abdelzaher. Assignment and Scheduling Communicating Periodic Tasks in Distributed Real-Time Systems. *IEEE Transactions on Software Engineering*, 23:745–758, 1997.
16. J. Santos and V. M. Manquinho. Learning Techniques for Pseudo-Boolean Solving. In *LPAR Workshops*, 2008.
17. L. Sha, T. Abdelzaher, K.-E. Arzen, A. Cervin, T. Baker, A. Burns, G. Buttazzo, M. Caccamo, J. Lehoczky, and A. K. Mok. Real Time Scheduling Theory: A Historical Perspective. In *RTSS*, 2004.