ArchEx: An Extensible Framework for the Exploration of Cyber-Physical System Architectures

Dmitrii Kirov[1], Pierluigi Nuzzo[2], Roberto Passerone[1], Alberto Sangiovanni-Vincentelli[3]
[1] University of Trento, Italy {dmitrii.kirov, roberto.passerone}@unitn.it
[2] University of Southern California, Los Angeles, USA, nuzzo@usc.edu
[3] University of California, Berkeley, USA, alberto@eecs.berkeley.edu

ABSTRACT

We present ArchEx, a framework for cyber-physical system architecture exploration. We formulate the exploration problem as a mapping problem, where "virtual" components are mapped into "real" components from pre-defined libraries to minimize an objective function while guaranteeing that system requirements are satisfied. ArchEx leverages an extensible set of patterns to enable formal, yet flexible, requirement specification, a graph-based internal representation of the system architecture, and algorithms based on mixed integer linear programming to solve the mapping problem. Its effectiveness is demonstrated on two industrial case studies: an aircraft power distribution network and a reconfigurable automated production line.

1. INTRODUCTION

Architecture exploration of complex cyber-physical systems (CPSs) is a major design challenge due to the lack of abstractions able to capture heterogeneous requirements and enable efficient co-design. CPS design would substantially benefit from methodologies that can express different design concerns (e.g., energy, timing, reliability) using a common formalism. Similarly, tools that can support these methodologies and are able to generate efficient architectures, while guaranteeing design correctness, are highly desirable [3, 6].

In this paper, we introduce ArchEx 2.0, an optimization-based framework for CPS architecture exploration. We leverage a generic representation of an architecture as a network of components. On this network we are able to express a variety of system requirements, such as connectivity, reliability, and timing, and find an optimized architecture that satisfies them. We build on our seminal work on reliability-driven optimized architecture design [3, 11], and regard architecture exploration as an optimized mapping problem, where "virtual" components satisfying a set of functional requirements are associated with "real" ones from a domain-specific library of possible implementations. However, we offer a new *problem formulation* that separates the *component selection* problem, i.e., whether a "virtual" component should be used in the architecture, from the *mapping* problem, i.e., which library component best implements the "virtual" one. This separation of concerns results in a Mixed Integer Linear Program (MILP) encoding of the mapping problem that is more general and more efficient than the previously proposed one [3, 11]. Moreover, we support a richer set of requirements, including timing constraints, and introduce *specification patterns* that can significantly reduce the burden of formulating the exploration problem. Our contributions are summarized below:

- We develop a framework for formulating and solving CPS architecture exploration problems, based on a general mathematical formulation and a reusable and extensible software infrastructure, which can be customized to support a variety of applications.
- To lower the problem formulation effort, we provide an extensible set of patterns to express requirements on the architecture and allow automatic translation into mixed integer linear constraints.
- We demonstrate the effectiveness and performance of our framework on two industrial case studies: an aircraft power distribution network and a reconfigurable production line.

A large body of design space exploration techniques and tools has appeared over the years in different domains. Our approach is complementary and can be combined with simulation-based methods [4–6], which have been recently proposed for CPS design. Alternative approaches use symbolic constraint satisfaction techniques based on ordered binary decision diagrams [10] or Satisfiability Modulo Theory (SMT) solvers to rapidly search for a feasible system configuration [12]. This work differs from these approaches, since it targets optimal solutions. It also differs from research efforts that propose SMT-based techniques to solve optimization problems, such as the "symbolic optimization" approach [8]. The focus of this effort is, instead, on facilitating the formulation and improving the usability of MILP-based techniques, which have proven to be effective in various domains, including avionic systems [11], building design automation [13], and wireless sensor networks [14].

2. PROBLEM FORMALIZATION

We assume that a CPS *architecture* is represented by a network of interconnected components, which are selected from a *library* (collection) $\mathcal{L}$ and comply with a set of *composition rules*. Each component in $\mathcal{L}$ has a set of *attributes* capturing its functional and extra-functional properties. Extra-functional properties include, for instance, energy consump-

DAC '17, June 18 - 22, 2017, Austin, TX, USA

ISBN 978-1-4503-4927-7/17/06…$15.00

DOI: http://dx.doi.org/10.1145/3061639.3062204

tion, processing delay, and cost. Each component has a set of *terminals* parameterized with *terminal variables*. Input and output terminals are used to send and receive signals or the values of terminal variables. Composition rules define which connections are allowed and how terminal variables may be assigned. Components can have different *types*, i.e., different roles or functions in the system [3]. We focus on networks whose components exchange entities or quantities via *flows* (e.g., message flow, power flow, product flow). Therefore, certain components have the role of sources or sinks.

We model the system architecture as a directed graph (V, E), where V is a set of components (nodes) while an edge $e_{ij} \in E$ represents an interconnection from v_i to v_j, with $i, j \in \{1, \ldots, |V|\}$, $|V|$ being the cardinality of V. Edges are interpreted as binary variables that evaluate to one (zero) to indicate the presence (absence) of interconnections between nodes. We also write e_{v_i,v_j} to denote an edge from node v_i to node v_j. A *template* $\mathcal{T}$ is a reconfigurable architecture, i.e., a graph with a fixed set of nodes V but variable set of edges E. A *configuration* is an assignment over the variables in E. Both nodes and edges in the graph are labeled with types, terminal variables, and attributes corresponding to those from the library $\mathcal{L}$. We denote with $\mathcal{M} : V \to \mathcal{L}$ the map that associates each "virtual" component (a graph node) with a "real" one in the library. We represent this map by assigning to each pair (v_i, l_j), with $v_i \in V$ and $l_j \in \mathcal{L}$, a binary variable $m_{ij} \in M$, which is one if v_i is mapped to l_j and zero otherwise. Finally, we say that a component v_i is *instantiated* (or used) if at least one incoming edge variable e_{ij} or outgoing edge variable e_{ji} evaluates to one. Edges are directly mapped to a pre-defined set of connection elements in the library, e.g., switches, wires, or wireless links.

We call P a *partition* over V, such that all components belonging to the same subset in P have the same type. A *path* $\pi(v_0 \to v_n)$ is a sequence of distinct nodes $\{v_0, \ldots, v_n\}$ such that $e_{v_i v_{i+1}} \in E$ evaluates to one for each i. We write $|\pi|$ to denote the length of π. Edges can also be labeled with binary variables y^π, where y^π_{ij} is one iff e_{ij} connects the nodes (v_i, v_j) in π. Let S_1 and S_n in P include, respectively, all the sources and the sinks of the network. Then, a *functional link* F_i is the set of all paths from any source in S_1 to a sink $v_i \in S_n$. Such links are essential for a system to operate correctly. For instance, in a power distribution network, they represent the paths between electrical loads and power sources. A *functional flow* $\mathcal{F}$ is an ordered sequence of component types $(t_1, \ldots, t_n)$ that are needed to implement a link between a source and a sink.

Given a template $\mathcal{T} = (V, E)$ and a library $\mathcal{L}$, we use optimization to find a configuration E^* and a map M^* that satisfy a set of requirements (e.g., interconnection, reliability, timing), while minimizing a cost function. Our decision variable set is $D = E \cup M$. The final assignment over D provides an optimal architecture, i.e., a network topology, in which a subset of the nodes and edges in $\mathcal{T}$ is used, and the mapping of nodes to components in $\mathcal{L}$. Below, we formulate the cost function, mapping constraints, and system requirements in terms of mixed integer linear constraints.

Cost Function. Every node and every edge in $\mathcal{T}$ is associated with a cost value. This may represent the monetary cost as well as other cost parameters, such as idle time, energy, weight. We then consider cost functions that can be expressed as the sum of the costs of all the instantiated components (nodes) and connections (edges):

$$\sum_{i=1}^{|V|} \delta_i c_i + \sum_{i=1}^{|V|} \sum_{j=1}^{|V|} e_{ij} \tilde{c}_{ij}, \tag{1}$$

where c_i is the cost of component v_i, $\tilde{c}_{ij}$ is the cost of the edge e_{ij}, and δ_i is a binary variable equal to one if the component is instantiated and zero otherwise. We also assume $e_{ii} = 0$ for all i. Depending on the specific problem, some of the terms in (1) may be omitted. The overall cost function is a weighted sum of different concerns, where some of the weights can also be set to zero by the user.

Interconnection Constraints. Linear arithmetic constraints can be used for enforcing valid connections between components or limiting the number of allowed connections. Let A, B and C be sets in P. Then, interconnection constraints can assume the following forms:

$$\sum_{j=1}^{|B|} e_{a_i b_j} \geq (\leq, =)\ 1 \qquad \forall i \in \mathbb{N} : 1 \leq i \leq |A|, \tag{2a}$$

$$\bigvee_{i=1}^{|A|} e_{a_i b_j} \leq \bigvee_{k=1}^{|C|} e_{b_j c_k} \quad \forall j \in \mathbb{N} : 1 \leq j \leq |B|, \tag{2b}$$

where $e_{a_i b_j}$ is an edge from node a_i to node b_j (and similarly $e_{b_j c_k}$). Constraints (2a) prescribe that there exists at least (at most, exactly) one connection from a node in A to a node in B. Constraints (2b) state that if node b_j has a connection to any node in A, then it must also have a connection to at least one node in C.

Mapping Constraints. We denote as $\mathbf{m^k}$ the mapping matrix for type k, where $m^k_{ij} = 1$ iff a virtual component (graph node) $v_j \in P_k$ is implemented by component $l^k_i \in \mathcal{L}_k$. $\mathcal{L}_k$ and P_k are the subsets of $\mathcal{L}$ and V including all the elements of type k in $\mathcal{L}$ and V. Let $\mathbf{e}$ be the adjacency matrix of $\mathcal{T}$, i.e., $e_{ij} = 1$ if there is a connection from node v_i to node v_j, and 0 otherwise. Then, the mapping constraints for type k assume the following form:

$$\bigvee_{i=1}^{|\mathcal{L}_k|} m^k_{ij} = \bigvee_{i=1}^{|V|} (e_{ij} \vee e_{ji}) \quad \forall j \in \mathbb{N} : 1 \leq j \leq |P_k|, \tag{3a}$$

$$\sum_{i=1}^{|\mathcal{L}_k|} m^k_{ij} \leq 1 \quad \forall j \in \mathbb{N} : 1 \leq j \leq |P_k|. \tag{3b}$$

Constraints (3a) state that each component of type k that is instantiated must be mapped to one of the components in $\mathcal{L}_k$. Constraints (3b) ensure that virtual components are never mapped to more than one library component. Similar constraints are enforced for all the types in $\mathcal{T}$. The encoding approach in this paper facilitates the exploration of different implementation alternatives. A change in $\mathcal{L}$ only affects the mapping constraints, which makes this formulation more general than the one in [3, 11], as the mapping constraints are not hard-coded as a part of the interconnection constraints. Moreover, this approach is more efficient. Let ℓ be the number of library options available to implement each component. Solving an equivalent mapping problem with the formulation in [3, 11] requires a number of decision variables that is quadratic in ℓ. This number becomes, instead, linear in ℓ by using the approach in this paper.

Flow Constraints. Each edge e_{ij} of $\mathcal{T}$ can be associated with a real variable λ_{ij} that expresses the flow rate through

the edge. We assume that the flow originates from a source and propagates to a sink via connections and intermediate components. Let node b_j have an intermediate type in the functional flow $\mathcal{F}$, which is neither a source nor a sink. The input flow rate at b_j can then be expressed as $\sum_{i=1}^{|V|} \lambda_{ib_j}$. Flow rates for output edges e_{b_jk} can then be assigned as follows:

$$\sum_{i=1}^{|V|} \lambda_{ib_j} e_{ib_j} \;=\; \sum_{k=1}^{|V|} \lambda_{b_jk} e_{b_jk}, \tag{4}$$

which is a balance equation at the terminals of b_j. A flow rate variable is forced to zero if the corresponding edge variable is zero. Some of the constraints (3a)-(4), including products between real and binary variables, are nonlinear, but can be linearized using standard techniques [3].

Workload Constraints. Each node in V can be labeled with a *throughput* μ, e.g., if this node represents a processor core or an industrial machine. To avoid *overloading*, we can bound the incoming workload for node v_j as follows:

$$\sum_{i=1}^{|V|} \lambda_{iv_j} \;\leq\; \mu_j, \tag{5}$$

requiring that a valid input is processed before the next one arrives. If m_{ij}^k is the element of the mapping matrix $\mathbf{m^k}$ associated with v_j and component l_i^k in $\mathcal{L}_k$ and $\mu^{\mathcal{L}_k}$ is the vector of throughputs for the components in $\mathcal{L}_k$, then we have $\mu_j = \sum_{i=1}^{|\mathcal{L}_k|} m_{ij} \mu_i^{\mathcal{L}_k}$.

Timing Constraints. We call *cycle time* the time required for a signal or message to get from a source to a sink. We assume that each node is labeled with a propagation delay τ and consider a simplified delay model, where the delay of a cascade of components is equal to the sum of the delays of each component. Let Π^{ab} be the set of all paths from source v_a to sink v_b. We can ensure that the cycle time of each path π in Π^{ab} does not exceed τ^* by requiring

$$\sum_{i=1}^{|V|} \tau_i w_i^\pi \;\leq\; \tau^* \quad \forall\, \pi \in \Pi^{ab}, \tag{6}$$

where τ_i is the delay of node v_i in $\mathcal{T}$ and w_i^π is a binary variable which evaluates to one if $v_i \in \pi$ and zero otherwise. Formally, $w_i^\pi = 1$ iff $\sum_{j=1}^{|E|} (y_{ij}^\pi \vee y_{ji}^\pi) \geq 1$. Values for y_{ij}^π can be assigned by adding a balance equation $\mathbf{c}(y^\pi)^T = z^\pi$ to the list of constraints, where $\mathbf{c}$ is the incidence matrix obtained by considering all the possible edges in $\mathcal{T}$ and z^π is a vector of length $|V|$, such that $z_a^\pi = 1$, $z_b^\pi = -1$ and the remaining values of z^π are zero. Auxiliary MILP constraints can be added to require a certain number of distinct paths between two nodes, as further detailed in [14].

We can finally characterize the *idle rate* of a component (e.g., a processing unit) as the difference between the processing rate and the incoming flow rate. We can then use the following constraint to limit the total idle rate of an architecture to be below a required value η^*:

$$\sum_{j \in Idx(PU)} \left(\mu_j - \sum_{i=1}^{|V|} \lambda_{ij} \right) \leq \eta^*, \tag{7}$$

where $Idx(PU)$ is a set of indices corresponding to the nodes in V that are labeled as processing units.

Reliability Constraints. In safety-critical applications, reliability requirements prescribe that a functional link should be guaranteed with a certain probability for a system to operate correctly, that is, the probability for a sink to be disconnected from all sources should be less than a desired threshold. To capture this class of constraints, we leverage the MILP encoding techniques introduced in [3].

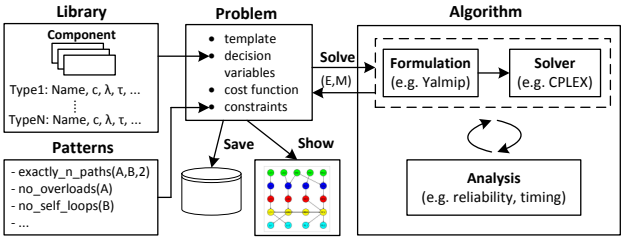


Figure 1: Structure of the ArchEx 2.0 toolbox.

Algorithms. ArchEx suppports two methods to solve the mapping problem detailed above [3]. The *eager* optimization method solves a *monolithic* problem, which includes all the optimization constraints. Since some of them may originate from approximations (e.g., reliability constraints), optimality is only guaranteed within the error bound due to the approximation. Alternatively, the *lazy* method leverages a coordination of specialized solvers. In this paradigm, the MILP solver is called *iteratively* on smaller problem instances including only a subset of constraints (e.g., interconnection constraints) to generate candidate configurations. The validity of these configurations is then checked against the other constraints via exact analysis methods. If these constraints are violated, a conflict-driven learning function is called between iterations of the MILP solver to incrementally add new constraints to the original formulation based on the analysis of previous outcomes, prune the search space, and rapidly progress towards a feasible solution. Solving a small number of simpler problem instances can significantly reduce the execution time with respect to a monolithic approach. However, global optimality is no longer guaranteed.

3. ARCHEX IMPLEMENTATION

We detail the structure of our framework and the set of patterns used to capture design requirements.

Tool Structure. As shown in Figure 1, the software structure directly reflects the modular, component-based approach adopted in Section 2, amenable to extensibility and design reuse. Generic classes are used to capture and manipulate mapping problems. The class *Problem* provides routines for formulating and solving an architecture exploration problem, such as reading and processing input files, specifying the component library, the decision variables, the constraints and the cost function, visualizing and saving the results. The *Component* class represents a generic component with several attributes, e.g., type, subtype, cost. The class *Library* is a collection of Component objects. It provides methods for creating a library from a text file, querying it for different components and attributes, and defining the mapping constraints. Reusable data structures, namely, *AdjacencyMatrix* and *LibraryMapping*, store the decision variables for the architecture selection and mapping problems.

The input to the toolbox consists of two text files: problem description and library. In the former, the users specify general information about the problem (e.g., component types, functional flows), template structure (e.g., the maxi-

mum number of components of each type), and the requirements in terms of patterns. The library is organized as a list of records grouped by component types. Each record represents a distinct component, and includes a name and a list of attributes belonging to each type. Components in the library can also be labeled with subtypes. For instance, power sources can be partitioned into low- and high-voltage sources while still being of the same type, e.g., AC or DC. Finally, components having the same type and sub-type may be grouped using *tags* based on the problem domain. For instance, as shown in Sec. 4.1, we distinguish between left and right AC buses, based on their location, as this poses restrictions on the feasible connections. Types, sub-types, and tags can all be used as parameters in the patterns.

The class *Algorithm* includes methods for solving MILP problems as well as the interfaces to the solvers. To report on the flexibility and usability of ArchEx for different applications, we use monolithic optimization in the numerical evaluation of Section 4. However, we also provide an infrastructure to design generic iterative schemes, including interfaces to analysis and conflict-driven learning routines that can be domain-specific. Current support for iterative schemes focuses on reliability analysis and the MILP modulo reliability algorithm [3, 11] for the exploration of reliability-driven CPS architectures.

Patterns. A pattern has a name that reflects the associated requirement and a list of arguments including the component types to which it applies. Each pattern is used to automatically generate MILP constraints over the input arguments, operating on corresponding subsets of decision variables. The access to the actual problem variables and the internal data structures is transparent to the user, which makes it easier to formulate and solve exploration problems. A system developer can then leverage these higher-level primitives to encode a problem, rather than manually generating the underlying optimization constraints, which is a tedious, error-prone task, often requiring the touch of an optimization expert. In this way, patterns can also contribute to reducing the chances of errors, and therefore the debugging effort, by virtue of their abstract nature.

Table 1 shows a representative list of patterns currently supported by ArchEx. Each pattern is an intuitive abbreviation of a requirement, which is close to a natural-language expression but still preserves a formal semantics. For example, to express that "there must be at least one connection between components of type A and components of type B", one can use the pattern `at_least_n_connections(A,B,1)`, which is later translated into a constraint as in (2a). Similarly, `in_conn_implies_out_conn` is used to encode balance constraints on the component connections (edges): if there is a certain type of incoming edge then a certain type of outgoing edge must be present. As further exemplified in Sec. 4, the patterns in Table 1 cover the categories of constraints in Sec. 2 and can be reused across domains of applications, which is an indication of their power to capture the essence of the problems of interest. Finally, the list can be incrementally extended to create more expressive, domain-specific languages based on simpler primitives.

Table 1: Representative List of Supported Patterns.

General	`at_least_n_components(T,S',N)` `at_least_n_paths(T1,T2,N)`

Connection	`at_least_n_connections(T1,T2,N)`
	`in_conn_implies_out_conn(Tin,T,Tout)`
	`bidirectional_connection(T1,T2)`
	`no_self_loops(T)`
	`cannot_connect(T1,S'1,T2,S'2)`

Flow	`flow_balance(T,S')` `no_overloads(T,S')`
Timing	`max_cycle_time(T,N)` `max_total_idle_rate(T,N)`
Reliability	`min_redundant_components(T,N)` `max_failprob_of_connection(T1,T2,N)`

T is a component type, S is a subtype, N is a numerical parameter. Primed parameters (e.g., S') are optional.

4. NUMERICAL EVALUATION

ArchEx 2.0 is offered as a Matlab toolbox [2] and currently uses Cplex [1] to solve MILP problems and Yalmip [9] to facilitate their formulation. We demonstrate the effectiveness of our approach on two applications. Numerical experiments were performed on an Intel Xeon 3.6-GHz 4-core processor with 24 GB of memory.

4.1 Aircraft Power Distribution Network

We first apply our approach to the avionics case study from [3, 11]. The number of electronic components installed on modern aircrafts has increased over the past years, which makes the design of safety-critical subsystems challenging. An aircraft Electrical Power distribution Network (EPN), such as the one in Figure 2a, is one example. Power is delivered from a set of sources to a set of loads (sinks) via AC and DC buses. The system is divided into left and right parts, but the corresponding generators (L/R-GEN) and auxiliary power units (APUs) can power both parts. Components can be further classified as high voltage (HV) and low voltage (LV). Rectifier units (RU) are used to convert AC power to DC power, while HV levels can be converted into LV levels using a transformer-rectifier unit (TRU). Sensors monitor the health state of generators and buses and inform the controller, which actuates a set of switches (contactors) to keep critical loads powered even if the components fail.

As a first step, we create a library $\mathcal{L}$ with components of the following types: generators (G), AC buses (A), rectifiers (R), DC buses (D), and loads (L). Each component is labeled with cost c, subtype s, and failure probability p. Common subtypes are HV and LV, while generators and rectifiers have extra subtypes, APU and TRU, respectively. Generators, buses, and loads are labeled with power ratings g, power capacities b, and power requirements l, respectively. Contactors are modeled with edges. We further assume that contactors and loads have no failures, the other components fail with probability 2×10^{-4}, and contactors have a fixed cost. Finally, some loads are sheddable, while others are non-sheddable (critical) and have a tighter failure probability requirement, 10^{-9} versus 10^{-5}.

Next, we create a problem description file. We set the functional flow to $\mathcal{F}$ = (G,B,R,D,L) and specify *tags* to group components based on their location, i.e., left (LE), right (RI), and middle (MI). The latter is reserved for APUs, which can be connected to both parts. We define the template $\mathcal{T}$ by declaring the maximum number of components for each type and tag, as summarized in Table 2.

Table 2: Template and library for the EPN example.

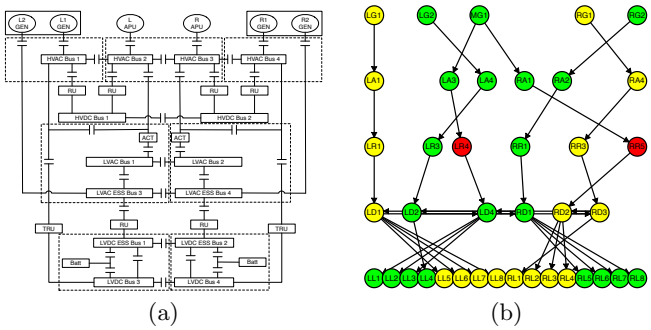
Type	Max # in $\mathcal{T}$ (Left,Right)	Cost	g,b,l (kW)	
			HV	LV
Generator	2,2 + 2 APU	g/10	60,80,150	20,30
AC bus	4,4	2000	150	30
Rectifier	5,5	2000	-	-
DC bus	4,4	2000	30	5
Load	8,8	0	{7,8,...,20}	1,2,3,4,5

We then specify the requirements using patterns. For instance, requirements like "each load must be connected to exactly one DC bus" or "a rectifier connected to a DC bus must also be connected to an AC bus" are specified using connectivity patterns. Similarly, we restrict the connections between left and right components, except for AC and DC buses, which can be used to connect both sides. The pattern `cannot_connect` is used to restrict direct connections between HV and LV components (which is possible only with a TRU). A domain-specific pattern, `has_sufficient_power`, is used to require that left and right generators are able to power all the corresponding loads. Finally, the pattern `max_failprob_of_connection` specifies the failure probability for each functional link using the encoding proposed in [3]. Patterns hide the details of the MILP formulation that can be massive. Our specification file only consists of 46 patterns, and a total of 90 lines of code, including variable declarations and composition rules, while the automatically generated MILP formulation in standard form amounts to more than 100,000 lines and 20,000 variables. This shows the advantage of raising the level of abstraction of design capture using patterns.

By using the monolithic optimization approach, ArchEx generates an EPN topology of complexity comparable with the one in Fig. 2a in about 5 h, of which 2.5 h are used to encode the problem. The resulting failure probability is 0.5×10^{-9} for every functional link and the overall cost is 106,000. Green and yellow nodes in Figure 2b represent HV and LV components, respectively. Red components are TRUs connecting the HV and LV portions of the system. Horizontal connections between DC buses increase the system reliability, by creating redundant paths from loads on one side of the system to sources on the other side or APUs.

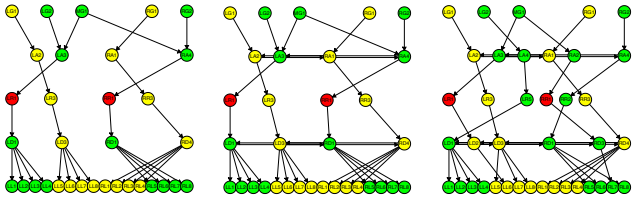
By using the iterative approach, the same problem is solved in three iterations, as summarized in Fig. 3a-3c. After the first iteration (Fig. 3a) every load has only one path to a generator, which is not enough to satisfy the reliability requirements. Then, ArchEx connects existing DC and AC buses together (Fig. 3b) and, finally, two extra AC and DC buses are added, one of subtype HV and one of subtype LV (Fig. 3c). The resulting failure probabilities are $(0.38, 0.19) \times 10^{-9}$ for the (HV,LV) functional links. The cost is 108,000, slightly higher than the one obtained with monolithic optimization, for a total execution time of 56 s, of which 98% are used for the problem formulation. The MILP encoding has around 5,000 constraints and 1,500 variables.

Finally, we tested the performance of ArchEx 2.0 on the benchmarks used in [3]. We achieve problems with up to one half of the constraints reported in [3] and 2-4x faster execution speeds, which shows the effectiveness of the newly proposed encoding for architecture selection and mapping. With respect to its predecessor, ArchEx 2.0 can efficiently generate more complex architectures, e.g., including HV and LV power distributions, and supports a richer set of requirements, including timing and flow constraints.



(a) (b)

Figure 2: (a) Simplified diagram of an EPN adapted from a Honeywell patent [11]; (b) EPN architecture generated by ArchEx using monolithic optimization.



(a) (b) (c)

Figure 3: EPN architectures and reliabilities (HV,LV) generated by ArchEx using the iterative approach: (a) $r = (0.6, 0.8) \times 10^{-3}$; (b) $r = (0.2, 0.32) \times 10^{-6}$; (c) $r = (0.38, 0.19) \times 10^{-9}$.

4.2 Reconfigurable Production Line

The recent concept of "Industry 4.0" advocates the usage of CPSs in factory automation as a major goal [7]. In particular, Reconfigurable Production Lines (RPL) provide a great value as they are able to adjust according to the company's and customers' needs. An RPL consists of a source that provides parts to be processed (assembled, packaged) on the line, a set of machines connected by conveyors, and a sink that collects the final product. A typical shop floor can have several RPLs for different product types. Reconfiguration is, in particular, related to having different *operation modes*. For instance, at some time, manufacturing of the product on line 1 is not required, while there is an increased demand for another product, processed by line 2. Instead of installing a fully parallel line, which can be costly, it is possible to reuse (reconfigure) line 1 to increase the processing rate. Junction conveyors can be used to connect existing lines together.

We use ArchEx to generate cost-effective RPL architectures that are subject to flow, workload, and timing constraints as well as operation requirements. Our library $\mathcal{L}$ consists of 4 component types: Source (SRC), Machine (M), Conveyor (C), and Sink (SNK), all of them labeled with a cost c and a subtype s. Sources and machines are also characterized by a flow rate λ and a throughput μ, respectively. In our example, two product types, with tags A and B, are assembled on two separate production lines, each of them having two machines along the path. We can then set $\mathcal{F}_A$ = $\mathcal{F}_B$ = (SRC,C1,M1,C2,M2,C3,SNK), while $\mathcal{T}$ and $\mathcal{L}$ are

Table 3: Template and library for the RPL example.

Type	Max # in $\mathcal{T}$ (A,B)	Cost, $\times 10^3$	λ, μ (parts/min) A	B	AB
Source	1,1	0	12	10	-
Machine	3,2	{2,3,...,15}	3,6,20	3,5,13	10
Conveyor	3,2	0.5,1	-	-	-
Sink	1,1	0	0	0	-

shown in Table 3. As there is only one component type in $\mathcal{L}$ to implement a machine, we use the subtypes A,B, and AB to, respectively, categorize the machines that can be used only for product A, B, or both. The RPL must support two operation modes. In mode Ω_1, both A and B must be simultaneously produced with rates λ_A and λ_B. In mode Ω_2, A is produced with a double rate, $2\lambda_A$, while line B is stalled.We assume that λ_A and λ_B are fixed and that conveyors can automatically adjust to any input rate.

In addition to the connectivity constraints, specified as in Sec. 4.2, the pattern `has_operation_mode`(Ω) creates the flow rate matrices $\Lambda^{k,x}$ with $k \in \{\Omega_1, \Omega_2\}$ and $x \in \{A, B\}$, where $\lambda_{ij}^{k,x}$ is a decision variable representing the flow rate of product type x in operating mode k along the edge e_{ij}. $\Lambda^{\Omega_1,A}$ and $\Lambda^{\Omega_1,B}$ set to zero all the flow rates between components associated with different product types, as no line can be borrowed for another product. This is not the case for $\Lambda^{\Omega_2,A}$, since the line associated with product B may be reused for product A in mode Ω_2. Finally, $\Lambda^{\Omega_2,B}$ is a matrix of zeros. We then use pattern `no_overloads` to require that all machines be able to handle the input rate in every mode, and `flow_balance` to guarantee that the input flow is correctly split between conveyors and machines. Our specification consists of 63 instances of 12 patterns.

The monolithic MILP formulation has approximately $5,000$ constraints and $3,000$ variables. The MILP solver terminates in 0.4 s, while the overall execution time is 28 s. The resulting configuration, shown in Fig. 4a, uses only one machine of type 1 and one machine of type 2 for each product line. To support Ω_2, part of the flow of product A is redirected to line B (C1A2 $\rightarrow$ C1B1), where reconfigurable machines, marked by a red color, are installed. Processed parts of product A are then sent back to Sink A (C3B2 $\rightarrow$ C3A1 $\rightarrow$ SnkA). For the given template $\mathcal{T}$ and library $\mathcal{L}$, reusing line B is more cost-effective than installing additional conveyors and machines on line A.

As a second example, we introduce the additional requirement that the sum of the idle rates of all the machines should be at most 10 parts/min, which can be done using the `max_total_idle_rate` pattern. We then obtain the architecture in Fig. 4b, synthesized in 31 s, which still reuses line B in mode Ω_2. However, it is now more convenient to implement M1 and M2 in line A by inserting two additional machines in parallel. Each machine is slower than in the previous design, but we achieve a total idle rate of 8 parts/min instead of 28 parts/min.

5. CONCLUSIONS

ArchEx 2.0 is a framework for CPS architecture exploration based on a high-level pattern-based specification language and MILP-based architecture selection algorithms. We demonstrated its effectiveness on two industrial case studies. As a future work, we will investigate extensions of our approach to other design problems, such as topology synthesis and node placement in wireless networks.

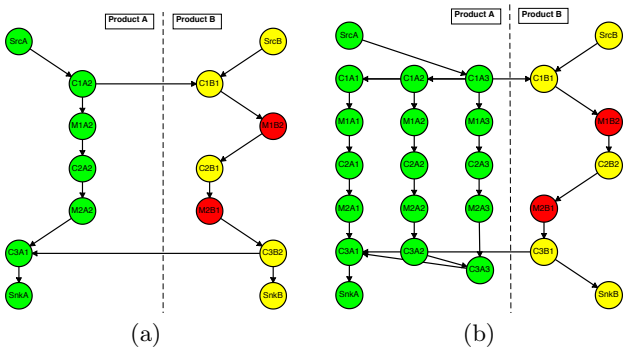


Figure 4: Generated RPL architectures: (a) line B is reused for product A in one of the operation modes; (b) machines are added in parallel to achieve 3.5× idle rate reduction.

Acknowledgments. The authors wish to acknowledge Nikunj Bajaj and Michele Lora for contributions to the implementation of an earlier prototype of the toolbox, and the support of Terraswarm, one of six centers of STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA.

6. REFERENCES

[1] (2016, Nov.) IBM ILOG CPLEX Optimizer. [Online]. www.ibm.com/software/commerce/optimization/cplex-optimizer/.

[2] (2017, Apr.) ArchEx 2.0: CPS Architecture Exploration Framework. [Online]. https://bitbucket.org/regkirov/archex.

[3] N. Bajaj, P. Nuzzo, M. Masin, and A. Sangiovanni-Vincentelli. Optimized selection of reliable and cost-effective cyber-physical system architectures. In *Proc. Design, Automation and Test in Europe*, pages 561–566, 2015.

[4] A. Canedo and J. H. Richter. Architectural design space exploration of cyber-physical systems using the functional modeling compiler. *Procedia CIRP*, 21:46–51, 2014.

[5] A. Davare et al. METROII: A design environment for cyber-physical systems. *ACM Transactions on Embedded Computing Systems*, 12(1s), March 2013.

[6] J. Finn, P. Nuzzo, and A. Sangiovanni-Vincentelli. A mixed discrete-continuous optimization scheme for cyber-physical system architecture exploration. In *Proc. IEEE/ACM Int. Conf. Computer-Aided Design*, pages 216–223, 2015.

[7] M. Hermann, T. Pentek, and B. Otto. Design Principles for Industrie 4.0 Scenarios. In *Proc. Hawaii International Conference on System Sciences*, pages 3928–3937, 2016.

[8] Y. Li, A. Albarghouthi, Z. Kincaid, A. Gurfinkel, and M. Chechik. Symbolic optimization with SMT solvers. In *ACM SIGPLAN Notices*, volume 49, 2014.

[9] J. Löfberg. YALMIP : A Toolbox for Modeling and Optimization in MATLAB. In *Proceedings of the CACSD Conference*, Taipei, Taiwan, 2004.

[10] H. Neema, Z. Lattmann, et al. Design space exploration and manipulation for cyber physical systems. In *Proc. Workshop Design Space Exploration of Cyber-Physical Systems*, 2014.

[11] P. Nuzzo, H. Xu, N. Ozay, J. B. Finn, A. L. Sangiovanni-Vincentelli, R. M. Murray, A. Donzé, and S. A. Seshia. A contract-based methodology for aircraft electric power system design. *IEEE Access*, 2:1–25, 2014.

[12] S. Peter and T. Givargis. Component-Based Synthesis of Embedded Systems Using Satisfiability Modulo Theories. *ACM Trans. on Des. Automation of Electr. Systems*, 20(4), 2015.

[13] A. Pinto, M. D'Angelo, C. Fischione, E. Scholte, and A. Sangiovanni-Vincentelli. Synthesis of embedded networks for building automation and control. In *Proc. American Control Conference*, pages 920–925, 2008.

[14] A. Puggelli, M. M. R. Mozumdar, L. Lavagno, and A. L. Sangiovanni-Vincentelli. Routing-aware design of indoor wireless sensor networks using an interactive tool. *IEEE Systems Journal*, 9(3):714–727, 2015.