Submitted, Hybrid Systems: Computation and Control (HSCC) 2013
http://www.cds.caltech.edu/~murray/papers/xom13-hscc.html

A Domain-Specific Language for Reactive Control Protocols for Aircraft Electric Power Systems

Huan Xu
California Institute of Technology
1200 E. California Blvd.
MC 104-44
Pasadena, CA 91125
mumu@caltech.edu

Necmiye Ozay
California Institute of Technology
1200 E. California Blvd.
MC 107-81
Pasadena, CA 91125
necmiye@caltech.edu

Richard M. Murray
California Institute of Technology
1200 E. California Blvd.
MC 107-81
Pasadena, CA 91125
murray@cds.caltech.edu

ABSTRACT

This paper describes the use of a domain-specific language, and an accompanying software tool, in constructing correct-by-construction control protocols for aircraft electric power systems. Given a base topology, the language consists of a set of primitives for standard specifications. The accompanying tool converts these primitives into formal specifications, which are used to synthesize control protocols. We can then use TuLiP, a Python-based software toolbox, to synthesize centralized and distributed controllers. For systems with no time involved in the specifications, this tool also provides an option to output specifications into a SAT-solver compatible format, thus reducing the synthesis problem to a satisfiability problem. We provide the results of our synthesis procedure on a range of topologies.

1. INTRODUCTION AND MOTIVATION

Domain-specific languages are languages adapted to a particular application or set of tasks. While general purpose languages (e.g., C or Java) may offer broader programming features, domain-specific languages (e.g., HTML or Verilog) provide more expressiveness and ease of use within a given domain [6]. Examples of languages used in the context of cyber-physical systems can be found in [1] and [3].

In aerospace, next-generation aircraft have moved away from purely mechanical and hydraulic subsystems, instead increasing reliance on electric power to supply subsystems, including flight-critical ones [8]. The growing complexity of electric power systems on aircraft, coupled with the need for safety, reliability, and autonomy, has increased the need to utilize formal methods and verification tools in order to properly analyze and design such large-scale systems. Previous work on embedded control software synthesis has been explored by Piterman and Pnueli [10]. Wongpiromsarn et al. [13] and Ozay et al. [9] have used the temporal logic planning toolbox TuLiP [14] to address the issue of creating correct-by-construction control protocols for aircraft management systems. Given a system topology, Xu et al. [15] show how text-based requirements can be converted to linear temporal logic (LTL), a formal specification language (see [11] for more details) to synthesize centralized and distributed controllers.

While the use of formal specification languages and correct-by construction synthesis methods is beneficial in the area of controller design, unfamiliarity of formal methods amongst engineers may provide a challenge to widespread implementation of formal methods. We propose a domain-specific language that combines tools already in existence: visual programs for single-line diagrams, which engineers are familiar with, and primitives, which provide a more formal structure to specifications. The development of a domain-specific language, therefore, provides an easy interface between industry engineers knowledgeable in aircraft systems and the methods/tools used by computer scientists and software engineers. The rest of the paper is structured as follows: Section 2 explains how a given topology and components can be used in a domain-specific language, represented as "primitives." Section 3 explains the specification conversion tool, and Sections 4 and 5 provide a discussion/comparison of the synthesis results for different topologies, and extensions to current work. Figure 1 provides a flow diagram for the automatic specification generation procedure.

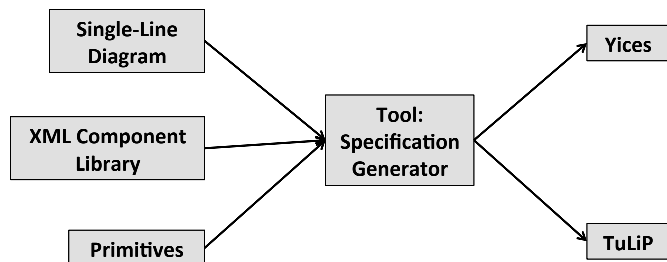

Figure 1: Architecture for the specification generator. The problem description includes three inputs: a single-line diagram, a component library, and a set of primitive specifications. The output is a set of formal specifications compatible with Yices (a SAT solver) or TuLiP (a reactive synthesis tool).

2. DOMAIN-SPECIFIC LANGUAGE

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
HSCC '13 Philadelphia, Pennsylvania USA
Copyright 20XX ACM X-XXXXX-XX-X/XX/XX ...$15.00.

This section introduces an aircraft electric power system and the set of primitives used in the domain-specific language.

2.1 Single-Line Diagram and Components

Figure 2 shows a single-line diagram for an electric power system on board a more-electric aircraft. The topology includes a combination of generators, contactors, rectifier units, transfomers, buses, and loads. The following is a brief description of the components referenced in the primary power distribution single-line diagram [8].

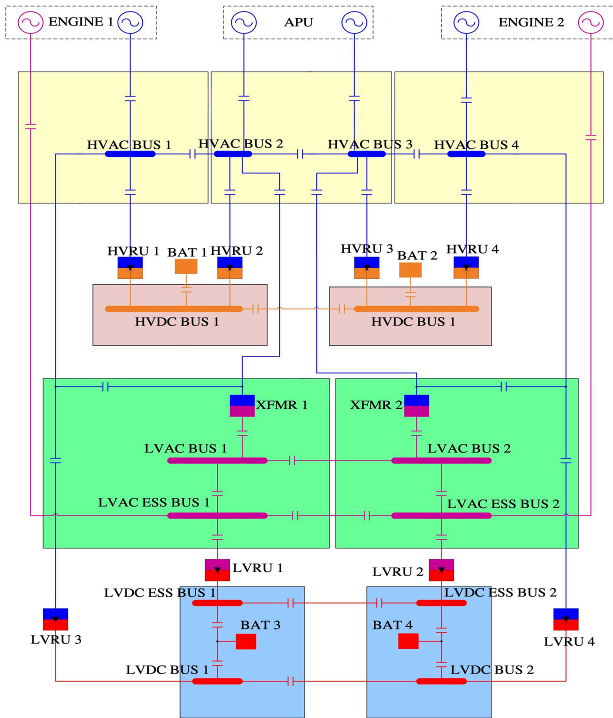

Figure 2: Single line diagram of an electric power system adapted from a Honeywell, Inc. patent [7]. Two high-voltage generators, APU generators, and low-voltage generators serve as power sources for the aircraft. Power can be routed from sources to buses through the contactors, rectifier units, and transformers. Buses are connected to subsystem loads.

AC generators provide high-voltage and low-voltage power to all components in the system. Main generators are powered by engines, while backup APU generators can be used in case of emergencies. *AC and DC buses* deliver power to high-voltage and low-voltage loads. Buses can be either essential or non-essential; essential buses supply power to loads that are safety-critical, while non-essential buses supply loads that may be shed in the case of a power failure. *Contactors* (represented in Figure 2 by ⊣⊢) are high-current electric switches that connect the flow of power from sources to buses and loads. They can reconfigure (i.e., switch between open and closed) through commands from one or multiple controllers. *Rectifier units* convert AC power to DC power, *transformers* step down a high voltage to a lower one, and a combination *transformer rectifier unit* both converts AC to DC power and lowers the voltage.

2.2 Input Files

Two sets of inputs must be provided from the information given by the diagram (connectivity) and components (attributes). First, the single-line diagram, a visual representation, can be converted into a graph data structure, where contactors are edges, and all other components represent nodes[1]. Let $G = (V, E)$ be a graph of the electric power system, with $V = \{v_1, v_2, \ldots, v_n\}$ containing all components consisting of generators, buses, and rectifier units. Loads, transformers and batteries are not implemented in our current formulation but can be easily integrated. The set of edges $E = \{e_1, e_2, \ldots, e_m\}$ then contains all contactors (as well as solid wire links between components). The adjacency matrix $\mathbf{A}_{ij}$ is a square adjacency matrix whose diagonal entries are zeros, and whose non-diagonal entries are ones or zeros depending on whether a contactor (or solid link) exists between vertices.

The second set of information is an XML file containing component attributes. Consider a simple case in which the XML file contains a listing for each type of component: contactor, generator, rectifier unit, and bus. Figure 3 depicts a example of an XML file for contactor and bus components. Each component has an attribute of *name* and *failure probability*, i.e., the probability each component has of failing over a certain number of operational hours. A failure probability of 10^{-3}, for example, means that the component may fail once over the course of 10^3 operating hours. Buses have an additional boolean attribute of *essential*, as well as an attribute *time* which states how long the bus may be unpowered for during a flight. In addition, contactors have attributes *opentime* and *closetime*, denoting the time it takes to physically open or close the contactor.

```
<contactor>              <bus>
      <failure>               <failure>
         1e-3                    1e-3
      </failure>              </failure>
      <opentime>              <essential>
         15                      true
      </opentime>             </essential>
      <closetime>        </bus>
         20
      </closetime>
</contactor>
```

Figure 3: A sample XML component library file for contactor and bus components that have attributes of *opentime*, *closetime*, and *essential*.

2.3 Specifications and Primitives

Given the topology of an electric power system and component attributes, the main design problem is determining all correct configuration of contactors for all flight conditions and faults that may occur. We now discuss some common or standard specifications relevant to the electric power system problem, and describe how these specifications may be written using a set of primitives.

Environment Assumptions: The overall system safety level determines the possible combinations of failures which

[1]Graphical tools exist which can convert visual diagrams into XML code. We begin with the assumption that such a conversion has been implemented and the XML file is parsed into an adjacency matrix.

may occur. Consider the case where generators and rectifier units are environment variables, i.e., uncontrolled. Because each component has an individual failure probability, we can determine how many components may fail at a single instance (while satisfying the system safety rating), and produce a set of valid environment assumptions. Let $\mathcal{G}$ and $\mathcal{R}$ be the sets of all generators and rectifier units, respectively. In the environment primitive (in which only generators and rectifier units are uncontrolled), the first input is a system safety level, followed by all subsets of components that are uncontrolled. This can be written as $\mathbf{env}(10^{-9}, \mathcal{G}_e, \mathcal{R}_e)$, where $\mathcal{G}_e \subseteq \mathcal{G}$ and $\mathcal{R}_e \subseteq \mathcal{R}$.

No-paralleling of AC sources: One common specification may be that no two asynchronous AC sources can power a bus simultaneously. A non-paralleling primitive thus has inputs of any subset of $\mathcal{G}$. This can be written as $\mathbf{noparallel}(\mathcal{G}_p)$, where $\mathcal{G}_p \subseteq \mathcal{G}$.

Essential buses: Essential buses supply power to safety-critical subsystems and loads, and thus must be powered at all times. Let the set of all buses be $\mathcal{B}$. An essential bus primitive can input any subset of $\mathcal{B}$. This is written as $\mathbf{essbus}(\mathcal{B}_e)$, where $\mathcal{B}_e \subseteq \mathcal{B}$.

Bus unpowered time: Non-essential buses supply power to loads and subsystems which can tolerate loss of power for up to a certain period of time. This time information is captured from the component library, and thus the primitive may be written $\mathbf{buspower}(\mathcal{B}_s)$, where $\mathcal{B}_s \subseteq \mathcal{B}$, and $\mathcal{B}_e \cap \mathcal{B}_s = \emptyset$.

Disconnect with unhealthy: When certain components (generators or rectifier units) become unhealthy, they must be disconnected from the system for safety reasons, i.e., the contactor connecting that component to other buses or components, needs to open. A disconnect primitive can take as input the union of subsets of $\mathcal{G}$ and $\mathcal{R}$. This primitive is written as $\mathbf{disconnect}(\mathcal{G}_d \cup \mathcal{R}_d)$, where $\mathcal{G}_d \subseteq \mathcal{G}$ and $\mathcal{R}_d \subseteq \mathcal{R}$.

3. TOOL INTEGRATION

The electric power system can be abstracted into different model views. We consider the following four views: untimed, discrete variables; discrete-time, discrete variables; continuous-time, discrete variables; and continuous-time, continuous variables. A domain-specific language can facilitate consistency between these views by providing a unifying framework for constituent elements. The following section discusses how the design problem can be automatically synthesized within the model view of discrete variables with no time or discrete-time. Our tool, which converts the above primitives into a set of specifications, is written using Python, with the additional use of the software package NetworkX to study the underlying graph structure. The sourcecode is included in TuLiP version 0.4a (and above) under tools/AES directory.[2]

3.1 SAT Solver

Consider the case in which timing specifications are ignored. Generators and rectifier units can either be healthy or unhealthy, contactors may either be open or closed, and buses can either be powered or unpowered. The synthesis problem reduces to a Boolean satisfiability problem. For each set of environment scenarios, a specific configuration of contactors satisfies all system requirements. Our current tool converts the set of primitives to a format compatible with the solver Yices [5][3].

[2]AES Directory

Based on the graph G derived from the single-line diagram, we automatically instantiate components, such that

$$(\mathbf{define}\ g :: \mathbf{bool})$$
$$(\mathbf{define}\ r :: \mathbf{bool})$$
$$(\mathbf{define}\ b :: \mathbf{bool})$$
$$(\mathbf{define}\ c :: \mathbf{bool})$$

for all $g \in \mathcal{G}$, $r \in \mathcal{R}$, $b \in \mathcal{B}$, and $c \in \mathcal{C}$, where $\mathcal{C}$ is the set of all contactors.

Because the SAT solver searches for a different solution for each configuration of environment behaviors, we generate all allowable environment sets, given the system safety level, and thus generate a set of environment assertions. Let the set of environment variables $\mathcal{P} \subseteq \mathcal{G} \times \mathcal{R}$. Environment assumptions can be written as $(\mathbf{assert}\ (=\ p\ [\mathbf{status}]))$ for all $p \in \mathcal{P}$, and $[\mathbf{status}]$ is either **true** or **false**, denoting a healthy or unhealthy component.

To avoid paralleling, the tool takes all pairs of generators input from the primitive and searches for all simple paths between items in each pair. For all simple paths between generator pairs, we disallow all contactors within each path to be closed at the same time. Consider, for example, a path that contains three contactors (c_1, c_2, and c_3) between AC sources g_1 and g_2, as seen in Figure 4. The non-paralleling specification output would be of the form

$$(\mathbf{assert}\ (\mathbf{not}\ (\mathbf{and}\ (=\ c_1\ \mathbf{true})\ (=\ c_2\ \mathbf{true})\ (=\ c_3\ \mathbf{true})))),$$

where **true** denotes a closed contactor.

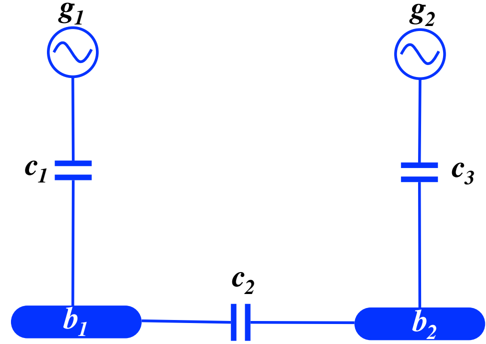

Figure 4: Simplified version of a the single-line diagram. Two AC generators connect to two buses via three contactors.

In order to assert that a bus must always remain powered, we first output a set of specifications which determine under what conditions a bus is powered or unpowered. We first search for all paths between each element input into the primitive, and output all path configurations that would cause the bus to be powered. This means all other buses, generators, and contactors in said path must be powered, healthy, and closed (respectively). If, in none of the paths, the conditions for a powered bus are met, then the bus is

[3]To be precise, Yices is an SMT solver which can also be used as a SAT solver.

unpowered. Consider again the simple example from Figure 4. The two output specifications for bus b_1 when it is powered would be

$$(\textbf{assert} \ (\rightarrow \ (\textbf{and} \ (= \ g_1 \ \textbf{true}) \\ (= \ c_1 \ \textbf{true})) \ (= \ b_1 \ \textbf{true}))),$$

and

$$(\textbf{assert} \ (\rightarrow \ (\textbf{and} \ (= \ g_2 \ \textbf{true}) \ (= \ c_2 \ \textbf{true}) \\ (= \ b_2 \ \textbf{true}) \ (= \ c_3 \ \textbf{true})) \ (= \ b_1 \ \textbf{true}))).$$

Then, b_1 is unpowered if neither of the two conditions above hold. This is written as

$$(\textbf{assert} \ (\rightarrow \ (\textbf{not} \ (\textbf{or} \ (\textbf{and} \ (= \ g_1 \ \textbf{true})(= c_1 \ \textbf{true})) \\ (\textbf{and}(= \ g_2 \ \textbf{true})(= \ c_2 \ \textbf{true})(= \ b_2 \ \textbf{true})(= \ c_3 \ \textbf{true})))) \\ (= \ b_1 \ \textbf{false}))).$$

These rules generalize to all AC and DC buses in the graph. Therefore, to assert that all buses are always powered, we write (**assert** $(= \ b$ **true**)), for all $b \in \mathcal{B}$.

To disconnect an unhealthy generator or rectifier unit, we search the graph for adjacent nodes, and assert an implication that if a component is unhealthy, the neighboring contactor must be open (take a value of **false**). This is written as (**assert** $(\rightarrow \ (= \ p$ **false**) $(= \ c_p$ **false**))), for all $p \in \mathcal{P}$, and $c_p \subseteq \mathcal{C}$ is the subset of contactors connecting component p to an adjacent component.

From the above set of specifications, Yices solves a satisfiability problem and determines the configuration for all contactors, for each environment configuration. Figure 5 shows an portion of the output from Yices. Thus a controller from Yices is a set of contactor configurations for each environment.

```
(= b5 true)      (= g2 false)
(= b6 true)      (= c27 false)
(= b7 true)      (= g3 true)
(= c56 true)     (= c38 false)
(= c67 true)     …
(= c78 true)     (= r10 true)
(= c89 true)     (= c510 true)
(= c1516 true)   (= r11 true)
(= c1617 true)   (= c611 true)
```

Figure 5: A sample output from Yices for a single environment configuration.

3.2 TuLiP

The benefits of an untimed model view is reducing the synthesis problem to a satisfiability problem, in which case a SAT solver may be used, the complexity of which is less than that for synthesis algorithms. More realistic design problems in the electric power system domain require timed specifications. We therefore incorporate formats compatible with TuLiP as well as Yices in the translation from primitives to specifications. TuLiP uses model view (II), which includes discrete-time and discrete variables; specifications are written in linear temporal logic (LTL). We assume the reader to have prior knowledge of LTL notation. For further details, see [11].

We visit the primitives described in Section 2.3, and begin by instantiating all variables (controlled and uncontrolled). Variables are again discrete and boolean. For all environment (uncontrolled) components, instantiations are written as **env_vars**$[p] = [0, 1]$, for all $p \in \mathcal{P}$. For all controlled variables, instantiations are written as **disc_sys_vars**$[s] = [0, 1]$, for all $s \in \mathcal{B} \cup \mathcal{C}$.

To specify the allowable environment assumptions, we again take all possible allowable sets of failures which can occur given the system failure probability. Assume the failure rate for each component is independent. Then, all combinations of failures that have a failure probability greater than the overall system level must be accounted for. The output specification, then, uses an always ($\Box$) operator alongside a string of disjunctions. Consider a simple example with two environment variables g_1 and g_2 that can take a value of 0 (unhealthy) or 1 (healthy). Suppose the overall system safety level is $1e-5$, and each generator has a failure probability of $1e-3$. The probability that both generators are unhealthy becomes $1e-6$, which is smaller than $1e-5$. So that acceptable environment behaviors include three possibilities: $g_1 = 1, g_2 = 1$; $g_1 = 1, g_2 = 0$; and $g_1 = 0, g_2 = 1$. One the tool calculates this set of environments, the TuLiP compatible specification output is

$$\textbf{assumptions} = \Box((g_1 = 1 \wedge g_2 = 1) \vee \\ (g_1 = 1 \wedge g_2 = 0) \vee (g_1 = 0 \wedge g_2 = 1)).$$

The non-paralleling specification disallows all contactors to be closed if they are within a path connecting two AC sources. In LTL, this is implemented using a never operator ($\Box\neg$). Using the example, from Figure 4, a non-paralleling specification would be of the form

$$\textbf{guarantees} = \Box\neg((c_1 = 1) \wedge (c_2 = 1) \wedge (c_3 = 1)).$$

The primitives for bus power and essential bus power first create a set of discrete properties that specify the conditions for when a bus is powered. Just as in the case using Yices, we find all paths from a bus to a generator, and list the component configurations needed for a bus to receive power. In Figure 4, for example, there are two properties for which bus b_1 can be powered. This is written as

$$\textbf{disc_props[d1]} = (g_1 = 1) \wedge (c_1 = 1), \\ \textbf{disc_props[d2]} = (g_2 = 1) \wedge (c_2 = 1) \wedge (b_2 = 1) \wedge (c_3 = 1).$$

Then, specifications output when bus b_1 is powered are

$$\textbf{guarantees} = \Box((d1) \rightarrow (b_1 = 1)), \\ \textbf{guarantees} = \Box((d2) \rightarrow (b_1 = 1)).$$

If neither proposition is true, b_1 is unpowered, written as

$$\textbf{guarantees} = \Box(\neg((d1) \vee (d1)) \rightarrow (b_1 = 0)).$$

These specifications are generalizable for all AC and DC buses within the topology.

Once these specifications are written, timing on buses can be introduced. If a bus is an essential bus, then another specification guarantees that the bus always remains powered. This is written as **guarantees** $= \Box(b = 1)$, for all $b \in \mathcal{B}_e$. For non-essential buses, we impose a maximum allowable time for which the bus may be unpowered. This value is taken from the XML component library file.

Note that LTL can be used to specify discrete-time properties for synchronous systems in which all processes (i.e., components) proceed in a lock-step manner. The next operator has a "time" measure so that, for a given property φ,

$\bigcirc\varphi$ signifies at the next time instant φ is true. To specify a property occurring at some point in the future, multiple next operators can be used, such that $\bigcirc^k\varphi \triangleq \bigcirc\bigcirc\cdots\bigcirc\varphi$ asserts that property φ holds k time instants in the future. To avoid the use of multiple next operators, which TuLiP cannot interpret, the time specifications in the electric power system uses a clock variable to define an equivalent property.

For each non-essential bus $b \in \mathcal{B}_s$, we introduce a unique counter t_k. We discretize each time step to take δ time. If a bus is unpowered, at the next step the counter will increment by δ. Counters are also bounded by a set maximum time limit. If the bus is powered, at the next step the counter will reset to 0. These specifications are output as

$$\textbf{guarantees} = \square((b_k = 0) \rightarrow (\bigcirc(t_k) = t_k + 1)),$$
$$\textbf{guarantees} = \square((b_k = 1) \rightarrow (\bigcirc(t_k) = 0)),$$

for all $b_k \in \mathcal{B}_s$. Then, we limit the number of "ticks" t_k can increment to $\frac{T}{\delta}$ steps. This specification is output as

$$\textbf{guarantees} = \square(t_k \leq \frac{T}{\delta}).$$

The final set of specifications involve removing unhealthy components from the overall system. To disconnect an unhealthy generator or rectifier unit, we use an implication. For all environment variables p_i, for $i \in \{1, \ldots, n_e\}$, if any component becomes unhealthy then the contactor connecting p_i to an adjacent component must open. This is written as **guarantees** $= \square((p_i = 0) \rightarrow (\wedge_{j\in N_i}(c_{ij}) = 0))$, where N_i denotes the set of vertices adjacent to vertex i.

These specifications are input into TuLiP, which interfaces with a digital design synthesis tool implemented in JTLV [10]. If the specification is realizable, TuLiP outputs a finite-state automaton that represents the control protocol. Figure 6 shows a portion of a sample automaton.

```
State 0 with rank 0 -> <g0:1, g1:1, ru4:1, ru5:1, c23:0,
c24:1, c67:0, b6:1, c13:1, b7:1, b2:1, b3:1, c35:1, c02:1>
    With successors : 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12,
13, 14, 15, 16
State 1 with rank 0 -> <g0:0, g1:1, ru4:0, ru5:1, c23:0,
c24:0, c67:1, b6:1, c13:1, b7:1, b2:0, b3:1, c35:1, c02:0>
    With successors : 17, 18, 19, 20, 21, 22, 23, 24, 25, 26,
27, 28, 29, 30, 31, 32
State 2 with rank 0 -> <g0:0, g1:1, ru4:1, ru5:0, c23:1,
c24:1, c67:1, b6:1, c13:1, b7:1, b2:1, b3:1, c35:0, c02:0>
    With successors : 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12,
13, 14, 15, 16
State 3 with rank 0 -> <g0:0, g1:1, ru4:1, ru5:1, c23:0,
c24:0, c67:1, b6:1, c13:1, b7:1, b2:0, b3:1, c35:1, c02:0>
    With successors : 17, 18, 19, 20, 21, 22, 23, 24, 25, 26,
27, 28, 29, 30, 31, 32
```

Figure 6: A sample finite-state automaton output from TuLiP that represents a control protocol.

4. RESULTS AND DISCUSSION

In this section we discuss some results for several electric power system topologies using both Yices and TuLiP. For ease of comparison, consider the base topology shown in Figure 7 that includes both AC and DC components. Each vertical set of components (generator, DC bus, rectifier unit, AC bus, and two contactors) form a base unit. Units may be connected together by contactors located between AC and DC buses. We examine the results for topologies with varying numbers of units.

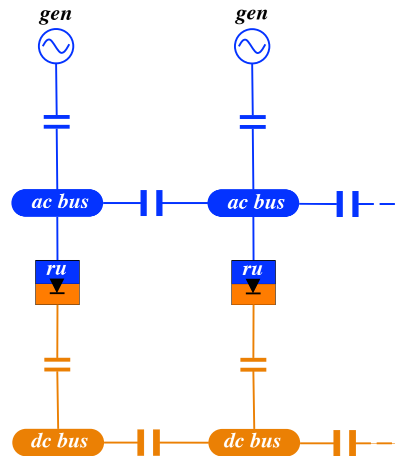

Figure 7: The base topology used to discuss the domain-specific language and conversion tool. Each base unit consists of a generator, DC bus, rectifier unit, and AC bus. Units are connected to each other by contactors between buses. More units are connected on the right (represented by the dotted wire/line.)

Table 1 lists the amount of time our tool takes to convert a set of primitives for a given base topology into formal specifications. Columns 2 and 3 show the size of the beginning graph, while column 4 compares the difference in times between converting specifications into a Yices or TuLiP-compatible format. The difference in conversion times is insignificant for smaller sized graphs. The Yices conversion takes more time due to the increase of allowable environment configurations. Because we solve a series of static problems, the tool must write a set of specifications for each of the environment scenarios. One thing to note is that the topologies we explore have many symmetries in the graph. Therefore, not all environment conditions need to be enumerated, e.g., an engine failure on the left side can be treated as similar to an engine failure on the right side.

Given the set of automatically generated specifications, Table 2 compares the time it takes for Yices and TuLiP to solve/synthesize a controller for a given topology. Column 2 lists the total number of environment configurations, i.e., the number of static problems Yices must solve. Then, Column 3 shows the time for Yices to solve a single environment configuration, as well as the time it takes for TuLiP to solve the full synthesis problem. Columns 3 and 4 show that solving a series of satisfiability problems is much time and memory efficient than using a synthesis tool. Increasing the topology from four to five base units dramatically increases the computation time. In addition, we applied the conversion tool to the single-line diagram topology from Figure 2. While the number of environment configurations is large, generation of all other primitives requires only 10 seconds. For one environment configuration, Yices takes 0.9 seconds and 39MB of memory to solve. This shows that the use of our conversion tool can be applicable to industrial-sized problems for untimed problems.

The size of the Yices controllers is the number of different

Table 1: Specification Conversion Time for Yices (Y) and TuLiP (T) [time in seconds]

Base Units	Nodes	Edges	Conversion Time (Y/T)
4	16	18	.13/.11
5	20	23	.25/.26
10	40	48	24/18
12	48	58	141/111
15	60	73	1634/1205

Table 2: Comparison of Synthesis Time for Yices (Y) and TuliP (T). [time in seconds]

Base Units	Yices Env.	Time(Y/T)	Mem. (Y/T)
4	25	.25/10.7	25MB/215MB
5	36	.82/1015	36MB/16GB
10	121	205.7/–	53MB/–
12	169	1410/–	158MB/–
15	256	62208/–	1.2GB/–

environment configurations. TuLiP synthesized controllers with four and five base units have 256 and 1024 states, respectively. While the use of a SAT solver is seemingly more advantageous than that of a synthesis tool, the range of problems which the SAT solver can handle is limited to those with untimed specifications. Alternatively, specifications written in linear temporal logic and synthesized using TuLiP can incorporate discrete-time specifications. Thus, we can automatically generate control protocols that can not only solve static configurations, but reason about how to transition between environment configurations through a series of contactor switches.

5. CONCLUSIONS AND EXTENSIONS

We have demonstrated techniques for synthesis of discrete-variable, untimed and discrete-time control protocols. TuLiP automata can also be converted into a continuous-time, continuous variable model view compatible with Simulink [12]. Future extensions will incorporate control protocols involving continuous-time and discrete-variables by converting synthesized protocols into a compatible format for a timed model checker such as UPPAAL [2]. We also plan to directly using the domain-specific language and tool to convert system requirements into specifications for a timed synthesis tool, such as UPPAAL-Tiga [4].

Further extensions include extending the domain-specific language to include user-specific requirements that may not be included in the high-level general specifications described in this paper. In addition, we are also exploring the use of this tool and language to distributed controller protocols. Namely, how to distribute a given topology among subsystems and generate interface specifications such that the overall system is realizable. Lastly, the problem of sensor placement and state estimation has been largely ignored in current problem formulations.

6. ACKNOWLEDGMENTS

Funding was provided by the MultiScale Systems Center (MuSyC). The authors would like to thank Rich Poisson from UTC Aerospace Systems, Ufuk Topcu, and Robert Rogersten for their helpful discussions.

7. REFERENCES

[1] K. An, A. Trewyn, A. Gokhale, and S. Sastry. Model-driven performance analysis of reconfigurable conveyor systems used in material handling applications. In *Cyber-Physical Systems (ICCPS), 2011 IEEE/ACM International Conference on*, pages 141 –150, April 2011.

[2] J. Bengtsson, K. Larsen, F. Larsson, P. Pettersson, and W. Yi. Uppaal: a tool suite for automatic verification of real-time systems. In R. Alur, T. Henzinger, and E. Sontag, editors, *Hybrid Systems III*, volume 1066 of *LNCS*, pages 232–243. Springer, 1996.

[3] A. Bhave, B. Krogh, D. Garlan, and B. Schmerl. View consistency in architectures for cyber-physical systems. In *Cyber-Physical Systems (ICCPS), 2011 IEEE/ACM International Conference on*, pages 151 –160, April 2011.

[4] F. Cassez, A. David, E. Fleury, K. G. Larsen, and D. Lime. Efficient on-the-fly algorithms for the analysis of timed games. In *IN CONCUR 05, LNCS 3653*, pages 66–80. Springer, 2005.

[5] B. Dutertre and L. D. Moura. The yices smt solver. Technical report, 2006.

[6] M. Mernik, J. Heering, and A. M. Sloane. When and how to develop domain-specific languages. *ACM Comput. Surv.*, 37(4):316–344, Dec. 2005.

[7] R. Michalko. Electrical starting, generation, conversion and distribution system architecture for a more electric vehicle, 10 2008.

[8] I. Moir and A. Seabridge. *Aircraft Systems: Mechanical, Electrical and Avionics Subsystems Integration*. Aerospace Series. John Wiley & Sons, 2011.

[9] N. Ozay, U. Topcu, and R. M. Murray. Distributed power allocation for vehicle management systems. In *CDC-ECE'11*, pages 4841–4848, 2011.

[10] N. Piterman and A. Pnueli. Synthesis of reactive(1) designs. In *In Proc. Verification, Model Checking, and Abstract Interpretation (VMCAI 06)*, pages 364–380. Springer, 2006.

[11] A. Pnueli. The temporal logic of programs. In *Foundations of Computer Science, 1977., 18th Annual Symposium on*, pages 46 –57, 31 1977-nov. 2 1977.

[12] R. Rogersten, H. Xu, N. Ozay, U. Topcu, , and R. M. Murray. An aircraft electric power testbed for validating automatically synthesized reactive control protocols. In *Proceedings of the 16th international conference on Hybrid systems: computation and control*, HSCC '13, submitted.

[13] T. Wongpiromsarn, U. Topcu, and R. M. Murray. Formal synthesis of embedded control software: Application to vehicle management systems.

[14] T. Wongpiromsarn, U. Topcu, N. Ozay, H. Xu, and R. M. Murray. Tulip: a software toolbox for receding horizon temporal logic planning. In *Proceedings of the 14th international conference on Hybrid systems: computation and control*, HSCC '11, pages 313–314, New York, NY, USA, 2011. ACM.

[15] H. Xu, U. Topcu, and R. M. Murray. Reactive protocols for aircraft electric power distribution. In *CDC, 2012 IEEE International Conference on*, 2012.