Factor Join Trees in Systems Exploration

Shahan Yang and John Baras

***Abstract*—The interaction of connected components creates complexity impeding the analysis of large systems. Thorough exploration, appearing as formal verification, optimization or constraint satisfaction typically has a complexity that is either a high order polynomial or an exponential function of system size. However, for a large class of systems the essential complexity is *linear* in system size and *exponential* in *treewidth* which makes the previous notion of exponential complexity in system size an accidental overestimate.**

In this paper, we demonstrate how this reduced complexity can be achieved using *summary propagation on factor graphs*. We describe summary propagation in terms of operations belonging to a *commutative semiring-weighted relational algebra*. We show how by appropriate selection of the commutative semiring, the same solution applies to propositional satisfiability, inference on Bayesian networks and a mixed integer linear optimization problem motivated by the power restoration problem for distributed, autonomous networks. This solution leads to a non-iterative distributed algorithm that operates on local data.

While weighted relational algebraic operators are used as the basic means of calculation, the algorithm presented works only on factor graphs that are trees. This requires the calculation of a non-unique structure called a *join tree*, which can be trivially generated from a unique structure called a *clique-separator graph*, which can be computed in linear time from a *SysML Parametric Diagram* description of the system for *chordal* systems[1]. Interesting artifacts arise from this structural analysis which may be interpreted as interfaces and components. The framework also contains mathematical artifacts that may be interpreted as hierarchy and abstraction.

We develop a tool to assist in the structural analysis described and implementations of the described algorithms.

I. Introduction

THE curse of dimensionality is a severe impediment to our ability to develop and reason about complex systems. As systems grow in size, the number of variables increase and this leads rapidly to computational intractability for questions of critical importance to the systems engineering process, for example, correctness and optimality. Under a given set of assumptions, does the system always operate in the intended manner? Is the system designed to attain its requirements at the lowest possible cost? Given a certain set of observations, how likely are certain unobserved events to have occurred? Although these questions appear different, they are mathematically very similar. They all require exploration of a high dimensional state or parameter space. In fact, each of the questions asked can be posed using a single common semiring equation, where the only difference is the specific semiring used.

S. Yang and J. Baras are with the Institute for Systems Research, University of Maryland, College Park, MD, 20742 USA. 2247 AV Williams Building, College Park, MD 20742-3271. Fax: 301-314-8486. Phone: 301-405-6606. E-mail: {syang, baras}@umd.edu.

[1]Systems that are not chordal are outside the scope of this paper but every system can be converted into a chordal one by adding fill-ins [1].

Naive exploration of high dimensional system spaces is computationally intractable. Systems engineering attempts to address this problem by partitioning a system into a graph of components. This promises the ability to understand the overall system by using a series of local analyses and a compositional technique to compose the results. However, most methods of partitioning are ad-hoc. Object-oriented analysis and design, for example, relies primarily upon general rules of thumb or the brilliance of an individual designer to determine the appropriate classes and interfaces. Furthermore, the ability to reason compositionally about a system depends on an appropriate partitioning. So we pose the two questions: is there a mathematical basis for deciding between different, but functionally equivalent, decompositions? And is there a locality respecting uniform framework for compositional analysis after the partitioning has been created?

For many NP-complete problems on graphs, including vertex cover, independent set, dominating set, graph k-colorability, hamiltonian circuit, network reliability [2], and dynamic programming [3], the complexity is exponential in *treewidth* and *linear* in problem size. We suggest that treewidth is closely tied to essential problem complexity. How can this be used within a systems engineering process?

A. Our Contribution

Our contribution is elucidating and advocating the use of chordality and the sum-product algorithm in the analysis of structured systems.

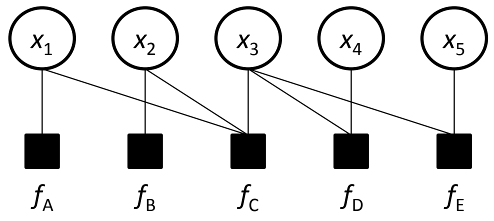
- We present an analysis method for optimization, logical and probabilistic inference on a certain class of systems where exploiting the logical system topology localizes the curse of dimensionality. The complexity of analysis using this framework grows *exponentially in the size of local neighborhoods* but only *linearly in overall system size* making it well suited to component based analysis.
- As an example application, we present a novel distributed algorithm that uses local information for optimal power restoration that also provides a space of possible computational and communication topologies that are consistent with the algorithm.
- We show how SysML Parametric Diagrams can be used to capture factor graphs and describe an algorithm for "refactoring" Parametric Diagrams into *factor join trees* for efficient summary propagation.

II. Factor Graphs

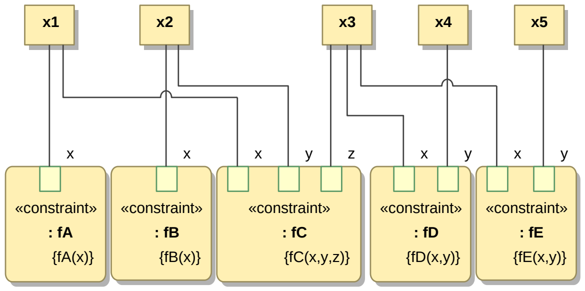
A factor graph[2] *is a bipartite graph that expresses how a "global" function of many variables factors into a product*

[2]See [4] for a good introduction to factor graphs.

of "local" functions [5]. The two independent sets of nodes in a factor graph are the *variable* nodes and *function* nodes. Figure 1(a) illustrates a factor graph. By observation, we note



(a) Example factor graph. The circles represent variables of the system and the black squares represent functions.



(b) An equivalent parametric diagram.

Fig. 1. The factor graph in 1(a), depicting the formula $f_A(x_1)f_B(x_2)f_C(x_1,x_2,x_3)f_D(x_3,x_4)f_E(x_3,x_5)$ is captured by the parametric diagram in 1(b). In contrast to the factor graph, the parametric diagram specifies which arguments of the constraints are mapped to which connected parameters.

that factor graphs can be captured by parametric diagrams as shown in Figure 1(b). Every variable node becomes a value property and every function node becomes a constraint block. A factor graph represents the decomposition of a function into the product ($\otimes$) of factors. In this paper, we restrict the domains of factors to finite, discrete sets. The codomain is a set forming the elements of a commutative semiring.

Let $\mathcal{X} = \{x_1 \dots x_N\}$ be a set of N variable symbols, with the corresponding variable having finite, discrete domain $\mathbb{X}_i$ for $i = 1 \dots N$. Let f be a function that can be represented as the product of M factors $f_1 \dots f_M$ such that

$$f(\mathcal{X}) = \bigotimes_{k=1}^{M} f_k(\mathcal{X}_k) \tag{1}$$

where $\mathcal{X}_k$ is the set of variables associated with f_k [3] .

The quintessential query on a factor graph is to determine the function resulting from summing (1).

$$\bigoplus_{x_1 \in \mathbb{X}_1} \cdots \bigoplus_{x_N \in \mathbb{X}_N} f(\mathcal{X}) \tag{2}$$

While this operation looks benign on the surface, there is an underlying combinatorical structure as shown in Figure 2. The set of terminal nodes of this tree can be mapped one to one onto the set of all combinations of variable assignments. This summation captures a full exploration of the complete variable space of the problem, which is what makes it useful for answering questions of interest in systems engineering. Roughly speaking, optimization, inference and satisfiability have in common this problem of searching over some parameter or state space. Since the summation in (2) implicitly evaluates every possible combination of state assignments, intuitively, it is connected to these problems. Section III has specific examples of problems that can be solved using (2). The factor graph provides a structure for computing this sum efficiently using summary propagation, as described in Section II-A.

A. Summary Propagation

Summary propagation (or interchangably, the sum-product algorithm) provides a way to compute (2) efficiently by reordering the sums $\oplus$ and products $\otimes$ using commutativity[4] and using the fact that multiplication distributes over addition to factor out invariant terms. In this section, we assume that the factor graphs are trees and address loopy graphs in Section II-C.

Property 1. *Every pair of nodes in a tree is connected by a unique simple path.*

Summary propagation works by first selecting a root node which induces a hierarchy on the tree as shown in Figure 4. Computing (2) can then be described as the recursion listed in Figure 3 on the tree starting at the root node.

Theorem 1. *The program listed in Figure 3 computes* (2). *(See Appendix A for proof.)*

Property 2. *The complexity of applying a summation to a function $f : \mathbb{X}_1 \times \mathbb{X}_2 \times \ldots \times \mathbb{X}_k \to \mathbb{R}$, over any number of its variables, has an upper bound given by the size of the domain of the function $\mathcal{O}(|Dom(f)|)$ where $|Dom(f)| = |\mathbb{X}_1 \times \ldots \times \mathbb{X}_k| = |\mathbb{X}_1| \cdot \ldots \cdot |\mathbb{X}_k|$. (See Appendix B for proof.)*

Property 3. *The complexity of performing a multiplication of two functions $f : \mathbb{X}_1 \times \ldots \times \mathbb{X}_k \times \mathbb{Y}_1 \times \ldots \times \mathbb{Y}_m \to \mathbb{R}$ and $g : \mathbb{X}_1 \times \ldots \times \mathbb{X}_k \times \mathbb{Z}_1 \times \ldots \times \mathbb{Z}_n \to \mathbb{R}$ has an upper bound of $|\mathbb{X}_1| \cdot \ldots \cdot |\mathbb{X}_k| \cdot |\mathbb{Y}_1| \cdot \ldots \cdot |\mathbb{Y}_m| \cdot |\mathbb{Z}_1| \cdot \ldots \cdot |\mathbb{Z}_n|$. (See Appendix C for proof.)*

Theorem 2. *The program listed in Figure 3 has an upper bound on complexity of*

$$\mathcal{O}(n \cdot \max_{k \in \{1 \dots M\}} |Dom(f_k)|)$$

where n is the number of nodes, and f_k are the factors for $k = 1 \dots M$. (See Appendix D for proof.)

Theorem 2 is the benefit of using the *Sum-Product* program. By Property 2, the complexity of performing (2) is equal to $\prod_{i=1}^{N} |\mathbb{X}_i|$ which grows exponentially with the number of

[3]We denote sets of symbols by script capital letters. For brevity, we will abuse notation and permit the set of symbols $\mathcal{X} = \{x_1, \dots, x_N\}$ to stand in for a tuple of symbols $(x_1, \dots, x_N)$. This is ill-defined, in a sense, because sets do not imply an ordering of the symbols and the ordering of the symbols plays a role in how the symbols are interpreted by the function, for example $f(\mathcal{X})$ could mean $f(x_1, \dots, x_N)$ or it could conceivably mean $f(x_N, \dots, x_1)$. The intended interpretation should be clear from the context.

[4]By definition of semirings the $\oplus$ operator is commutative and recall we have assumed a commutative semiring which means the $\otimes$ operator is also commutative.

$$
\bigoplus_{x_1 \in \mathbb{X}_1} \cdots \bigoplus_{x_N \in \mathbb{X}_N} f(\mathcal{X}) =
\begin{array}{c}
\bigoplus_{x_1 \in \mathbb{X}_1} \cdots \bigoplus_{x_{N-1} \in \mathbb{X}_{N-1}} f(\mathcal{X})|_{x_N = \mathbb{X}_N[1]} \quad = \quad
\begin{array}{c}
\bigoplus_{x_1 \in \mathbb{X}_1} \cdots \bigoplus_{x_{N-2} \in \mathbb{X}_{N-2}} f(\mathcal{X})|_{\substack{x_N = \mathbb{X}_N[1], \\ x_{N-1} = \mathbb{X}_{N-1}[1]}} \\
\oplus \\
\vdots \\
\oplus \\
\bigoplus_{x_1 \in \mathbb{X}_1} \cdots \bigoplus_{x_{N-2} \in \mathbb{X}_{N-2}} f(\mathcal{X})|_{\substack{x_N = \mathbb{X}_N[1], \\ x_{N-1} = \mathbb{X}_{N-1}[|\mathbb{X}_{N-1}|]}}
\end{array} \\
\oplus \\
\vdots \\
\oplus \\
\bigoplus_{x_1 \in \mathbb{X}_1} \cdots \bigoplus_{x_{N-1} \in \mathbb{X}_{N-1}} f(\mathcal{X})|_{x_N = \mathbb{X}_N[|\mathbb{X}_N|]} \quad = \quad
\begin{array}{c}
\bigoplus_{x_1 \in \mathbb{X}_1} \cdots \bigoplus_{x_{N-2} \in \mathbb{X}_{N-2}} f(\mathcal{X})|_{\substack{x_N = \mathbb{X}_N[|\mathbb{X}_N|], \\ x_{N-1} = \mathbb{X}_{N-1}[1]}} \\
\oplus \\
\vdots \\
\oplus \\
\bigoplus_{x_1 \in \mathbb{X}_1} \cdots \bigoplus_{x_{N-2} \in \mathbb{X}_{N-2}} f(\mathcal{X})|_{\substack{x_N = \mathbb{X}_N[\mathbb{X}_N], \\ x_{N-1} = \mathbb{X}_{N-1}[|\mathbb{X}_{N-1}|]}}
\end{array}
\end{array}
$$

Fig. 2. This figure shows the branching, combinatorical structure of evaluating the sum in (2). The tree continues to expand towards the right with the total number of layers equal to the number of variables and the number of terminal nodes in the tree is equal to $\prod_{i=1}^{N} |\mathbb{X}_i|$ which means that the size of this tree and the number of operations needed to perform the summation is roughly exponential in the number of variables. The indexing notation, $\mathbb{X}_i[k]$ assumes that domain $\mathbb{X}_i$, for $i = 1 \dots N$, has a default ordering of its elements such that they can be indexed $\mathbb{X}_i[1] \dots \mathbb{X}_i[|\mathbb{X}_i|]$.

variables. The expression $\max_{k \in \{1 \dots M\}} |Dom(f_k)|$ also grows exponentially with the number of nodes, but is restricted to local neighborhoods and the complexity is linear in the size of the graph. Thus systems questions that can be answered by (2) behave well for composition. The complexity of analysis is very sensitive to the density of local interaction, but then grows only linearly in overall system size. As defined in Figure 3, the sum product algorithm only works on trees, but we will show how it may be applied to a much larger class of systems in Section II-C.

B. Abstraction and Induced Hierarchy

For any graph that is a tree, every node can induce a hierarchy on the graph as shown in Figure 4. The hierarchy induced is based on paths leading away from the selected node. So for any tree, there are as many hierarchies as there are nodes. Interpreting Figure 4 in light of the Sum-Product algorithm (Figure 3), we see that the head node receives a summary of all the information in the rest of the tree projected onto its variables of interest. Abstraction can be interpreted precisely in this framework as the summation operation, summing over variables that are not directly related to the current node. This is a projection of the overall behavior into a lower dimensional space. In this sense, nodes that are higher in the hierarchy receive an abstract view of what lies lower in the hierarchy. However, what is higher or lower depends entirely upon a particular perspective or choice of head node. The traditional view of hierarchy is biased towards user level requirements. This idea is interesting because it indicates that when we are working on a low level subsystem, we must work with an abstract view of the overall system. Within this framework, many different abstract views are generated as projections.

Another note about hierarchy is that it is possible to obtain *multiple levels of resolution* by keeping certain variables exposed. For the lowest level of resolution, we might only keep variables pertaining to the root node. We could also choose to keep variables from multiple nodes. Figure 5 shows an example of multi-resolution abstractions for a particular choice of head node using the same factor graph as Figures 1 and 4. Sum-Product provides a means to expand and collapse different branches to provide different levels of resolution.

C. Join Trees

As presented, the Sum-Product algorithm performs exact computations, and has predictable complexity with the restriction that the factor graph is a tree. This topological limitation can be eased by using join trees (see [6] for an introduction). A join tree uses variable aggregation to create trees out of graphs that have loops. As an example of variable aggregation, let x_1 and x_2 be two variables with domains $\mathbb{X}_1$ and $\mathbb{X}_2$ respectively. Then we may define a new variable $x_{1\times2}$ that has domain $\mathbb{X}_{1\times2} = \mathbb{X}_1 \times \mathbb{X}_2$ where $\times$ denotes the Cartesian product of sets. x_1 and x_2 are aggregated into the single variable $x_{1\times2}$. Figure 6 shows a simple example of how this can be used to transform a loopy graph into a tree. In Figure 6, the variables x_1 and x_3 are aggregated, but it is also valid to aggregate variables x_2 and x_4 instead. This is an example of a general property which is that the join tree is seldom unique. The join tree chosen has an affect on the computational complexity of evaluating (2). Computing an optimal one is generally an NP-complete problem [7]. Here, rather than the general case, we will consider only graphs that are *chordal*.

```
function sumProduct(node) {
  rv = recurse(node);
  for (v ∈ node.variables) {
    rv = $\bigoplus_v$ rv;
  }
  return rv;
}

function recurse(node) {
  if (isVariable(node)) {
    rv = 1;
    for (c ∈ node.children) {
      projection = recurse(c);
      for (v ∈ c.variables \ node.variables ) {
        projection = $\bigoplus_v$ projection;
      }
      rv = rv $\bigotimes$ projection;
    }
    return rv;
  } else { // node is a function
    rv = node.function;
    for (c ∈ node.children) {
      rv = rv $\bigotimes$ recurse(c);
    }
    return rv;
  }
}
```

Fig. 3. The listing here shows pseudocode for the recursive function *sumProduct*. It takes as a parameter a *node*, the head of a factor hierarchy. The output of *sumProduct* is a value from the semiring. The output of *recurse* is a function mapping a set of variables to the semiring. A *node* can be either a variable node or a function node, consistent with the factor graphs in Figure 1. Since the graph is bipartite, every variable node has only function nodes as children and every function node has only variable nodes as children. *node.children* refers to the child nodes via the induced hierarchy and *node.variables* refers to the set of variables relevant to the node. For a function node, *node.variables* is the input variables to the function and for variable nodes, *node.variables* is the set of variables associated with the node (for simple factor graphs, there is only one).

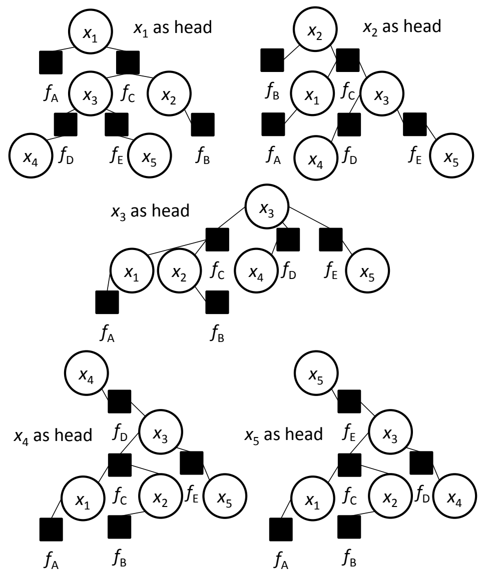


Fig. 4. Multiple hierarchies induced by selecting different head nodes for the factor graph shown in Figure 1. Note that the underlying trees are the same in each of these hierarchies.

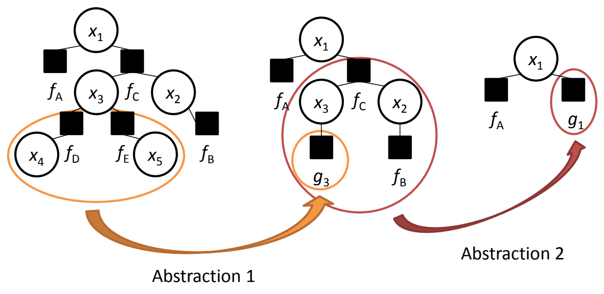


Fig. 5. This figure shows multiple abstract representations of the same function, by varying which variables are hidden. Abstraction 1 is achieved by hiding variables x_4, x_5 and defining $g_3(x_3) \equiv \left[\bigoplus_{x_4 \in \mathbb{X}_4} f_D(x_3, x_4)\right] \otimes \left[\bigoplus_{x_5 \in \mathbb{X}_5} f_E(x_3, x_5)\right]$. Abstraction 2 is achieved by further hiding variables x_2, x_3 and defining $g_1(x_1) \equiv \bigoplus_{x_2 \in \mathbb{X}_2} \bigoplus_{x_3 \in \mathbb{X}_3} (f_B(x_2) \otimes f_C(x_1, x_2, x_3) \otimes g_3(x_3))$. Both of these functions are the result of performing summary propagation on the collapsed nodes. The resulting functions are entirely equivalent to the original function, with certain information hidden.

Definition 1. *A **chordal**[5] graph is one in which every loop of length greater than or equal to 4 has an edge joining two non-adjacent nodes in the loop.*

Theorem 3. *For chordal graphs, computing a join tree requires $\mathcal{O}(n + m)$ operations where n is the number of nodes and m is the number of links [8].*

Theorem 3 is the reason we consider only chordal graphs here. The join tree is computed in a straightforward manner for graphs that are chordal, unlike the general case which is NP-complete.

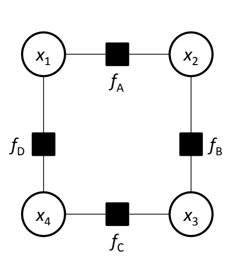
Even though it is computed in linear time, the join tree is not unique. Within a systems context, where behavior will be mapped to a physical structure, the logical structure of the join tree may have an impact on the design. This means that the particular choice of join tree may be a design decision so it is best to present the engineer with a means of choosing the best join tree. Every join tree can be generated from a unique structure called the *clique-separator graph* [9]. We first compute the clique-separator graph then assume that the user provides further input to convert the clique-separator graph into a join tree.

Figure 7 shows how a loopy, chordal graph can be transformed into a clique-separator graph. We call the undirected graph in Figure 7(a) a *functional dependence* graph. It is derived from a factor graph by removing the functional nodes but keeping track of connectivity. Each node corresponds to a variable node in the initial factor graph. The links represent the connectivity between the variables where two variables are connected in this graph iff there is a function node that has both of those variables as inputs.

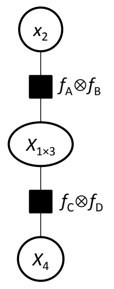
Definition 2. *Formally, the **functional dependence graph**, $G_S = (V_S, E_S)$, represents the structure of a function $f(\mathcal{X}) = \bigotimes_{i=1}^{M} f_i(\mathcal{X}_i)$ with $\mathcal{X}_i \subseteq \mathcal{X}$ for $i = 1 \ldots M$. The nodes of the graph are given by $V_S = \mathcal{X}$, the variables of the function. The edges are given by $E_S = \{(x, y) | \exists i \in \{1 \ldots M\} : x, y \in \mathcal{X}_i\}$.*

In this section, we will describe the construction of the join tree from the functional dependence graph of a factor graph. Section II-D will describe transforming the resulting join tree

[5]For an excellent introduction to chordal graphs, see [6].

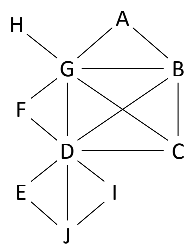


(a) Looping factor graph.

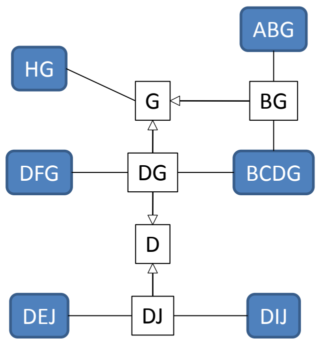


(b) Factor join tree.

Fig. 6. Variable aggregation can be used to transform a loopy graph into a tree. Variables x_1 and x_3 in the loopy graph (a) are aggregated into a single variable $x_{1\times 3}$ as shown in (b). The factor graph in (b) represents the same function as (a), but is a tree, so summary propagation is readily applicable. We assume that the values of the variables $x_{1\times 3}$ can be expressed as tuples (a, b) where $a \in \mathbb{X}_1$ and $b \in \mathbb{X}_2$ so applying the original or product aggregated functions is straightforward.



(a) The functional dependence graph. This graph is chordal.



(b) An equivalent clique-separator graph. The rounded rectangles are the cliques and the separators are the plain rectangles.

Fig. 7. This diagram shows a chordal functional dependence graph and its corresponding clique-separator graph. Every maximal clique of the chordal graph in 7(a) has a corresponding clique in 7(b). Also, every minimal vertex separator in the chordal graph 7(a) is represented as a rectangle in 7(b).

back into a factor graph. To handle the non-uniqueness of the join tree, we will use a unique generalization of join trees defined in [9] called the *clique-separator* graph which we review here.

Definition 3. *A* ***maximal clique*** *is a clique that cannot be extended to a larger clique by adding any node belonging to the graph.*

Definition 4. *A* ***minimal vertex separator*** *is a set of nodes that when removed from the graph creates ≥ 2 partitions separating some given nodes u, v. The set is minimal in the sense that by removing any node from the set would leave u, v in the same component when the set is removed.*

Definition 5. *The* ***clique-separator*** *graph consists of two types of nodes. There is one* ***clique*** *node for each maximal clique of G_S. There is one* ***separator*** *node for each minimal vertex separator of G_S.*

Both the clique and separator nodes in the clique-separator graph $\mathfrak{G} = (\mathfrak{V}, \mathfrak{E})$ correspond to sets of nodes in the original functional dependence graph G_S. We will refer to the sets associated with elements $\mathcal{V} \in \mathfrak{V}$ using the same identifier $\mathcal{V}$.

Definition 6. *The* ***clique-separator*** *graph also has two types of links. An* ***arc*** *connects two separator nodes. An arc occurs if a separator $\mathcal{S} \subset \mathcal{S}''$ and there is no separator $\mathcal{S}'$ such that $\mathcal{S} \subset \mathcal{S}' \subset \mathcal{S}''$. An* ***edge*** *connects a separator node $\mathcal{S}$ and a clique node $\mathcal{K}$ if $\mathcal{S} \subset \mathcal{K}$ and there is no separator node $\mathcal{S}'$ such that $\mathcal{S} \subset \mathcal{S}' \subset \mathcal{K}$.*

We can see by this definition of arcs that they are directed. The relationship between separators is by subset, and it is very similar behaviorally to an inheritence relationship. Any separator that is a superset of another separator can be thought of as a specialization. We will use the standard UML generalization arrow to depict arcs in the direction indicated as shown in Figure 7(b). The arcs are shown using generalization links and the edges are shown using undirected association edges.

Theorem 4. *The clique-separator graph is computed from the functional dependence graph in $\mathcal{O}(N^3)$ time [9].*

Maximum cardinality search can be used to find all the maximal cliques and minimal vertex separators in a graph [8] so the clique-separator graph can be constructed efficiently.

Theorem 5. *Every join tree of a given functional dependence graph can be determined from the clique-separator graph.*

Proof: Kumar [10] provides an algorithm for finding every join tree from an object he defines called the reduced clique hypergraph (rch). The rch consists exactly of nodes that are the maximal cliques and hyperedges that are the minimal vertex separators. A hyperedge connects two nodes iff the the associated separator is a subset of the associated clique. The rch can be obtained from the clique-separator graph by simply observing that the clique-separator graph also consists of the maximal cliques and the minimal vertex seprataors and includes the relationships between them. ■

Since every join tree can be found from the clique-separator graph, it will serve the purpose of providing a representation that allows the user to select a particular join tree for his application. The algorithm in [10] has steps that require an arbitrary choice to be made. This decision can be made by the engineer.

D. Factor Join Tree

The join tree, which is derived from the clique-separator graph in our approach, creates a tree topology from what may be a loopy topology, but does not contain the functions in the original factor graph it is derived from. In order to apply summary propagation, it is necessary to establish how these functions are related to the join tree so that (2) can be computed.

Theorem 6. *For every factor f in the original formula there exists at least one maximal clique in the clique-separator graph that contains all variables in $Dom(f)$.*

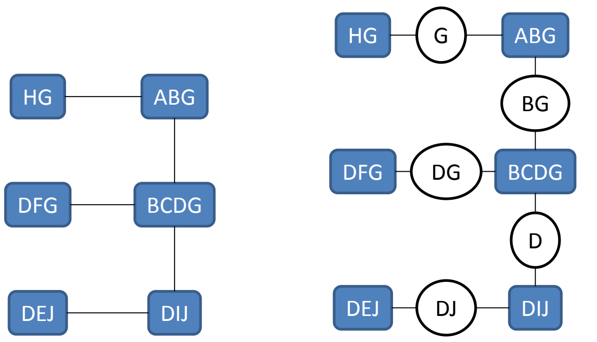
Proof: Every factor f induces a clique V_f on the variables. V_f is either a maximal clique itself or a subset of k maximal cliques $C_1, \ldots, C_k$. In either of these two cases, there will be at least one maximal clique that contains all the variables in $Dom(f)$ and since the maximal cliques are the nodes of the clique-separator graph, the result follows. ■

Because of Theorem 6, it is always possible to assign every factor in the original factor graph to at least one of the maximal cliques in the the clique-separator graph. These same function assignments can be carried over to the join tree, which has the same nodes as the clique-separator graph. When multiple functions are assigned to the same maximal clique, we use their product for the associated function.

The only remaining detail is the interpretation of variable nodes in the join tree. In the join tree, variables span connected subtrees. This is called the *running intersection property*, and is well known, see [10] for references.

Property 4. *Let $\mathcal{U}$ and $\mathcal{V}$ be any two vertices in a join tree and let $\mathcal{R} = \mathcal{U} \cap \mathcal{V}$. Since the graph is a tree, there is a unique path $\mathcal{U}, \mathcal{P}_1, \ldots \mathcal{P}_k, \mathcal{V}$ connecting $\mathcal{U}$ and $\mathcal{V}$. $\mathcal{R} \subset \mathcal{P}_i$ for $i = 1 \ldots k$.*

To turn the join tree into a factor join tree for summary propagation, we replace each link between nodes with a link that passes through an intermediate variable node. The variable node has as its variables the intersection of the variables of the linked nodes. See Figue 8 for an example. To apply



(a) A join tree derived from Figure 7.

(b) A factor join tree resulting from substituting links for variables.

Fig. 8. This figure shows how variable nodes can be inserted into the join tree to create a factor join tree. Observe that the running intersection property (Property 4) holds on this join tree. The maximal clique nodes have not been assigned functions here, since in this example, no functions were defined, but if they were, then Theorem 6 could be used to assign them.

summary propagation to this as given in Figure 3, no changes are necessary. Only the following corollary to Theorem 1.

Corollary 1. *The code listing in Figure 3 generalizes for computing* (2) *over factor join trees.*

Proof: Function nodes do not differ in factor join trees and regular join trees. The difference is in the variable nodes. We need only show that variables are not summed out before they are multiplied. This follows easily from Property 4. ■

E. Queries

In addition to being able to ask questions like (2), it is also useful to be able to ask what happens when variables are fixed to certain values. Fixing a variable to a particular value alters the topology of the factor graph by elimination of the corresponding variable nodes. Referring back to the example given in Figure 1, suppose we wish to examine the system fixed at the value of $x_3 = c$. Figure 9 shows the resulting factor graph. The topology shown in Figure 9 is partitioned,

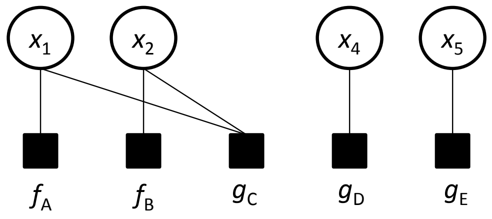


Fig. 9. This factor graph depicts the function shown in Figure 1 evaluated at $x_3 = c$, $g(x_1, x_2, x_4, x_5) \equiv f(x_1, x_2, x_3, x_4, x_5)|_{x_3=c}$, with $g_C(x_1, x_2) \equiv f_C(x_1, x_2, c)$, $g_D(x_4) \equiv f_D(c, x_4)$ and $g_E(x_5) \equiv f_E(c, x_5)$. Fixing the value of x_3 causes the resulting factor graph to be partitioned into three distinct graphs.

which means that the equation has completely independent factors. Evaluating (2) on g as defined in Figure 9 results in

$$\begin{aligned} &\bigoplus_{x_1 \in \mathbb{X}_1} \bigoplus_{x_2 \in \mathbb{X}_2} \bigoplus_{x_4 \in \mathbb{X}_4} \bigoplus_{x_5 \in \mathbb{X}_5} g(x_1, x_2, x_4, x_5) \\ &= \left[\bigoplus_{x_1 \in \mathbb{X}_1} \bigoplus_{x_2 \in \mathbb{X}_2} f_A(x_1) f_B(x_2) g_C(x_1, x_2) \right] \\ &\otimes \left[\bigoplus_{x_4 \in \mathbb{X}_4} g_D(x_4) \right] \otimes \left[\bigoplus_{x_5 \in \mathbb{X}_5} g_E(x_5) \right]. \end{aligned} \tag{3}$$

The *sum* itself decomposes into three independent factors in this case so the three trees can be evaluated independently of one another and then combined by multiplication. Since the resulting factor graph is simpler than the original factor graph, the complexity of evaluation is still bounded by Theorem 2.

Another possible query is to compute an entire function of several variables that are a subset of the original variables. By merely iterating the above over all possible variable value combinations we have the following.

Theorem 7. *Let $\mathcal{Z} = \{z_1, \ldots, z_q\} \subset \mathcal{X}$ be a subset of the input variables of a function $f(\mathcal{X})$. Computing $g(\mathcal{Z}) \equiv \bigoplus_{\mathcal{X} \setminus \mathcal{Z}} f(\mathcal{X})$ is bounded above in complexity by*

$$\mathcal{O}(\left[\prod_{i=1}^{q} |\mathbb{Z}_i| \right] \cdot n \cdot \max_{k \in \{1 \ldots M\}} |Dom(f_k)|).$$

This upper bound is fairly loose, but it is generally expensive to compute functions like this because it requires computing $\prod_{i=1}^{q} |\mathbb{Z}_i|$ individual sums. It can be useful because sometimes the topology of the graph does not allow a computational reduction.

The effect of the computation indicated by Theorem 7 is defining an objective function $h(\mathcal{Z})$ and adding it to the factor graph. Adding $h(\mathcal{Z})$ to the factor graph proceeds by adding the function node h and linking it to the variable nodes associated with each of the input variables in $\mathcal{Z}$. This changes the topology of the factor graph. Recall that the Sum-Product algorithm as defined with predictable complexity is only for trees. Adding a node may add a loop to the original factor graph. Section II-C will describe how loops are handled in greater detail, but they are not always reducible and sometimes can add complexity to the computation as indicated in Theorem 7.

F. Weighted Relational Algebra (WRA)

The sum-product rule maps easily to database computations. This was also observed in [11] in the context of Boolean logic, but we generalize in this presentation. There are two basic operations that are required for performing summary propagation, the sum and the product of two functions.

Definition 7. *We define the database representation, R, of a function $f(x_1, \dots, x_k)$ as follows. Let R have columns $x_1, x_2, \dots, x_k$, and w. There will be one row for R for every possible combination of $x_1, \dots, x_k$ values meaning $\prod_{i=1}^{k} |\mathbb{X}_i|$ rows. Column w will be assigned the value $f(x_1, \dots, x_k)$ corresponding to the x values of that row.*

The sum over a function, as indicated in the proof of Property 2, can be executed starting with a tabular representation of the function. Let R be as described in Definition 7. Then the following SQL command computes the summation over x_1.

```
SELECT $x_2, \dots, x_k$ , $\oplus w$ FROM $R$ GROUP BY $x_2, \dots, x_k$
```

The $\oplus$ operator could be SUM, MIN, MAX or any other database aggregator that corresponds to the semiring addition operation. The `GROUP BY` keyword ensures that the summation occurs over groups of common values. To modify this query for sums other than x_1, either add or remove variables from the lists given to the `SELECT` and `GROUP BY` statements.

The product of two functions also fits well within a database context. Let $f(x_1, \dots, x_m, z_1, \dots, z_s)$ and $g(y_1, \dots, y_n, z_1, \dots, z_s)$ be two functions sharing variables $z_1, \dots, z_s$. Let R and S be the database representations of f and g respectively. Then the product of the two functions can be computed using the following statement.

```
SELECT $x_1, \dots, x_m, y_1, \dots, y_n, z_1, \dots, z_s, w_R \otimes w_S$
FROM $R$ INNER JOIN $S$ ON $R.z_1 = S.z_1, \dots, R.z_s = S.z_s$
```

This operation can be thought of as a weighted natural join since the `ON` condition corresponds merely to the shared variables. The $\otimes$ operator corresponds with the semiring product operation.

Databases are a natural fit for performing computations of this form because the input models are relational models, which can be provided either parametrically, or as factor graphs. Databases also support the creation of different abstract views on the fly by the `CREATE VIEW` operation which is useful if these abstract views will be used in other parts of the system design.

III. Examples

A. Propositional Satisfiability

Amir [11] uses partitioning to reduce the complexity of solving propositional satisfiability problems and provides a compositional method. Here, we show how the same problem is solved using the Sum-Product algorithm over the Boolean semiring. Figure 10 shows a Boolean function as a loopy factor

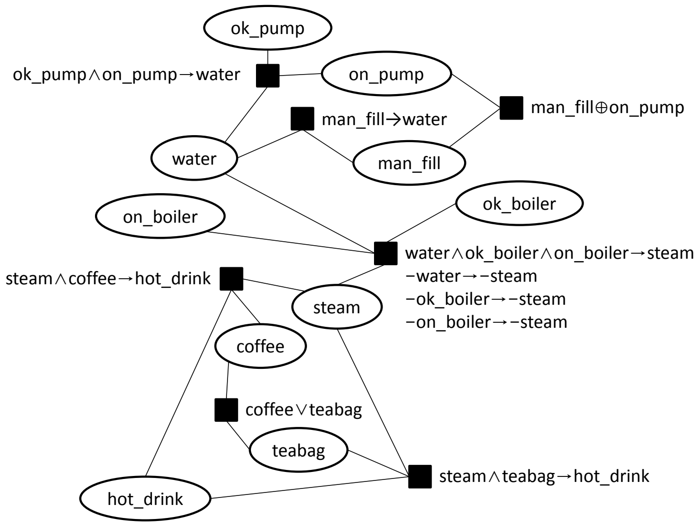


Fig. 10. This diagram shows a factor graph representing a Boolean function. We would like to know whether there is a satisfying assignment of variables for this Boolean function.

graph. Determining whether there is a satisfying assignment is the same as evaluating (2) over the conjunction or product of all the formulas. The Boolean semiring has as its domain $\{0, 1\}$ and its addition operator is $\vee$ while its multiplication operator is $\wedge$. Since this factor graph is loopy, we will not be able to perform summary propagation, so we try to form the factor join tree. The next step in this process is to find the functional dependence diagram, which is shown in Figure 11. Since the functional dependence diagram in Figure 11 is

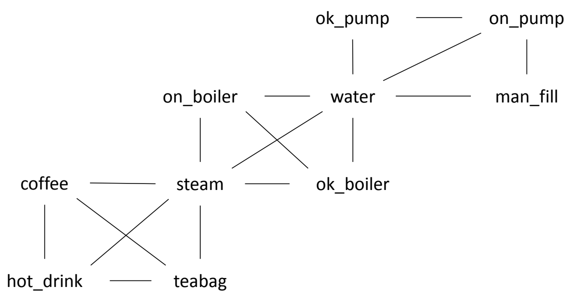


Fig. 11. This diagram shows the functional dependence diagram of the Boolean function depicted in Figure 10. Note that it is chordal. The maximal cliques are also fairly apparent in this diagram.

chordal, the clique-separator graph as described in Definition 5 can be found using maximum cardinality search. Figure 12 shows the result. From the clique-separator graph, we can determine the join tree. In this case, there are only two possible join trees. One in which water is connected to `ok_pump, on_pump, water` and the other case which is shown in Figure 13. Applying Theorem 6 allows us to determine a factor join tree from the join tree by assigning the formulas to the maximal cliques that have the associated variables.

The result of this transformation is that we reduce a Boolean satisfiability problem that has 10 variables and $2^{10} = 1024$

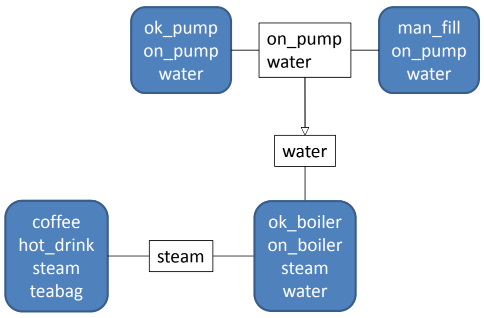


Fig. 12. The clique-separator graph for the Boolean function shown in Figure 10. Note that the cliques form logical groupings of the variables and the separators can be interpreted as shared variable interfaces. We can determine the treewidth at this point as the maximal sized clique, which is 4.

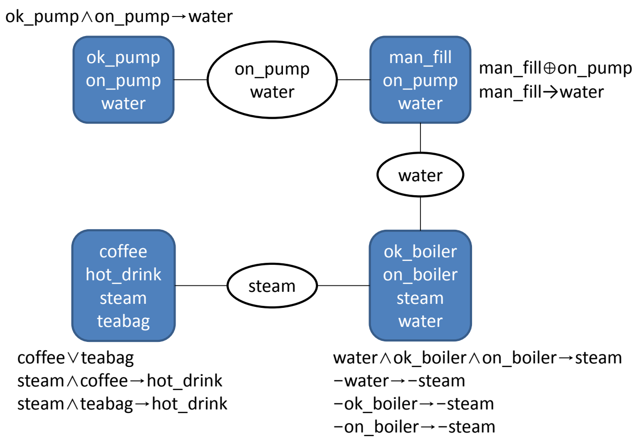


Fig. 13. One possible factor join tree of Figure 10 is shown in this diagram. Observe that the running intersection property holds.

possible assignments to investigate down to 4 Boolean satisfiability problems, each with less than or equal to $2^4 = 16$ possibilities to investigate. This is a significant computational reduction.

B. Bayesian Networks

This is the canonical application of join trees and inference using summary propagation. This is reviewed in the book of Pearl [12], but does not include clique-separators which give more flexibility than finding join trees directly. In Bayesian networks, the relevant semiring is the domain of $[0, 1] \subset \mathcal{R}$ with addition being addition on real numbers and product being the product of real numbers.

C. Power Restoration

The power restoration problem can be described with the following parameters and (time varying) variables.

N	number of generators
M	number of loads
K	number of buses
P	number of load priority levels
$C_i(t) \in \mathbb{R} \geq 0$	for $i = 1 \ldots N$ generator capacities
$D_i(t) \in \mathbb{R} \geq 0$	for $i = 1 \ldots M$ demands
$W_i(t) \in \mathbb{R}^P$	for $i = 1 \ldots M$ load weights
$A_i(t) \in \{0, 1\}$	for $i = 1 \ldots K$ bus availabilities

The load weights are vectors that are lexicographically ordered, which captures different priority levels for the loads. Additionally, there are connectivity constraints that describe the topology of the network.

$\mathcal{G}_i \subseteq \{1 \ldots K\}$	for $i = 1 \ldots N$ buses that may connect to generator i
$\mathcal{B}_j \subseteq \{1 \ldots M\}$	for $j = 1 \ldots M$ loads that may connect to bus j

The above variables and parameters are given inputs to the optimization problem. The decision variables are the generator assignments for the buses and the bus assignments for the loads.

$g_j \in \{i : j \in \mathcal{G}_i\}$	for $j = 1 \ldots K$ generator connected to bus j
$b_k \in \{j : k \in \mathcal{B}_j\}$	for $k = 1 \ldots M$ bus connected to load k

There is one constraint equation that must be satisfied for each generator $i = 1 \ldots N$. It says that the capacity of every generator must exceed the sum of the demands of the connected loads.

$$C_i(t) \geq \sum_{j \in \mathcal{G}_i} A_j(t) \mathbf{1}_0(g_j - i) \left[\sum_{k \in \mathcal{B}_j} \mathbf{1}_0(b_k - j) D_k(t) \right] \quad (4)$$

Given the above constraints [6], the score is the sum of the weights of the powered loads. For $i = 1 \ldots N$ generator i contributes S_i to the score:

$$S_i = \sum_{j \in \mathcal{G}_i} A_j(t) \mathbf{1}_0(g_j - i) \left[\sum_{k \in \mathcal{B}_j} \mathbf{1}_0(b_k - j) W_k(t) \right] . \quad (5)$$

The objective is to maximize the sum:

$$\max_{g_1 \ldots g_K, b_1 \ldots b_M} \sum_{i \in \{1 \ldots N\}} S_i . \quad (6)$$

It is clear from the description and well known [13] that this problem is a mixed integer linear program (MILP). While MILP can determine an optimal solution, the computational complexity is high, leading to the development of heuristic methods, for example [14].

There is one constraint equation per generator associated with its capacity and it has the exact same structure as the score equation. This implies that all of the variables associated with

[6] $\mathbf{1}_0(k) \equiv \begin{cases} 1 & \text{if k = 0} \\ 0 & \text{o.w.} \end{cases}$

each generator i, the potentially connected buses $\mathcal{G}_i$ and the potentially connected loads to those buses $\bigcup_{j\in\mathcal{G}_i}\mathcal{B}_j$, become part of a single clique. Let $B_i \equiv \mathcal{G}_i \cup \bigcup_{j\in\mathcal{G}_i}\mathcal{B}_j$, the set of variables in each clique. Figure 14 shows a higher order functional dependency diagram for this set. The individual variables are hidden and a link is shown if one clique is related to another. Treating the variables as aggregates with additional

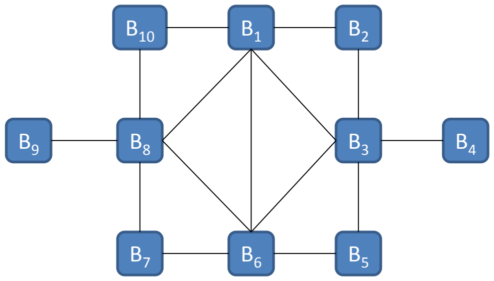


Fig. 14. This diagram shows the higher order functional dependency structure of the cliques associated with the generators. The underlying variables are not in a chordal relationship with one another, but this graph is chordal. If we treat the sets of variables B_i as aggregate values, functions are introduced so that consistency of the individual values can be maintained. The functional dependencies for these new consistency functions are the same as those shown in this graph.

functions to model the consistency between the individual variables allows us to create the higher-order clique-separator graph as shown in Figure 15 by maximum cardinality search. Some of the separators occur only between two cliques. Any

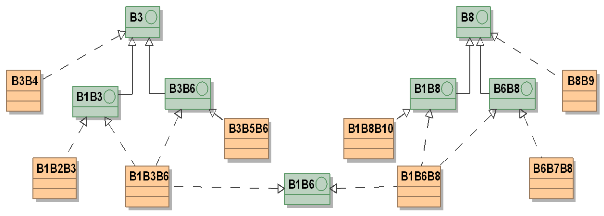


Fig. 15. A clique-separator graph for functional dependency structure shown in Figure 14. A slightly different notation is used here, that is consistent with SysML block diagrams. Separators are rendered as interfaces and cliques are rendered as blocks.

resulting join trees will necessarily have a link between those cliques. It is possible to reduce these and produce a reduced clique-separator graph as shown in Figure 16. The semiring

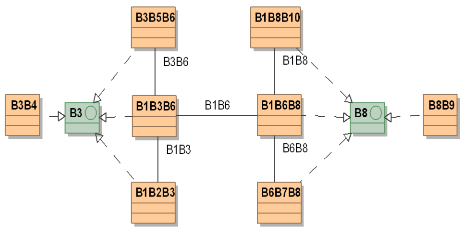


Fig. 16. The *reduced* clique-separator graph for the functional dependences shown in Figure 14. This clearly shows that there are only 9 possible join trees.

used to solve this particular example is max-plus, which consists of a real-valued domain, with max as the addition operator and real addition as the multiplication operation. The functions at each node are constructed to respect constraints (4) and the weights are given by (5), or the product (real addition) of scores in case of aggregation.

The interesting thing about this example is that the underlying topology of the variables was not chordal, but chordality at a higher level of the system design is still useful in creating a structure suitable for summary propagation. We have also introduced here the notion of a reduced clique-separator graph, which is easily generated from the clique-separator graph.

1) Computational and Communication Topology: Another interesting aspect of using factor join trees is that locality is respected so that distributed algorithms are easy to create. It also provides an answer to where to place a local computational resource. This scheme has flexibility in that different join trees are possible and aggregations of join tree nodes also result in topologies that respect the structure of the computation.

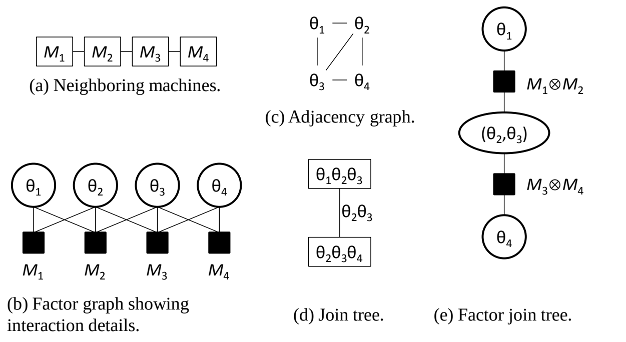
IV. Discussion

Systems development is coupled with the process of eliciting requirements. As development proceeds, unforseen interactions or scenarios are often encountered, requiring a refinement or modification of the requirements. A design will usually start simple, because the designer wants the system to be as simple as possible but it seldom ends this way because reality is seldom simple. Refactoring, or behavior performing transformations of the system are typically used to reduce perceived complexity as the system evolves. However, there are no quantitative metrics for measuring the complexity and the refactoring is performed in an ad-hoc manner. In this work, we identified *treewidth* as a metric for measuring the complexity of systems. Treewidth is based entirely on the logical structure of the system, which means it can be computed from non-rigorous SysML diagrams. SysML is often criticized for lacking semantics, but that is also a strength, because it is less costly to produce and thus frees a designer to reasoning at a higher level thus preventing a kind of deep tunnel vision that a more formal analysis method would require.

We make the suggestion that design tools should measure the treewidth of the system because it provides insight into the essential system complexity. While this paper focuses on measuring treewidth of parametric diagrams, treewidth decomposes along clique separators, which means that we can reason compositionally about treewidth. A clique separator would be interpreted in this sense as an interface that partitions the system. The treewidth of the composed system is equal to the maximum treewidth over the components. This also means that complexity hotspots with higher treewidth can be identified and analyzed independently.

Having a tool that makes treewidth calculations of the system provides a way for the designer to understand the implications of design decisions on system complexity. System complexity is a source of bugs and unforseen system behavior. If the system is too complex to analyze completely, then unforseen behavior will occur by definition. An awareness of system complexity at design time and making choices that tradeoff complexity can therefore reduce overall system costs.

As shown in our example with the power restoration problem, the join tree provides an interesting partitioning of the system from a behavior description of the system. There are many places where the question of how to allocate behavior or computation to a graph of computational resources. The join trees provide structures that are in a sense compatible with the underlying computational and logical topology. We would expect that generally speaking, variables that are logically related correspond to physically proximate system elements and so there is a match between the join tree, computational needs and underlying physical system structure. Also, as shown in the power distribution network example, summary propagation also provides an algorithm for performing distributed inference and optimization on such structures. For these reasons, we suggest that aiming to design a system as a join tree could be a useful design principle.



M_1 M_2 M_3 M_4
(a) Neighboring machines.
θ_1 θ_2 θ_3 θ_4
(b) Factor graph showing interaction details.
(c) Adjacency graph.
$\theta_1\theta_2\theta_3$
$\theta_2\theta_3$
$\theta_2\theta_3\theta_4$
(d) Join tree.
$M_1 \otimes M_2$
(θ_2,θ_3)
$M_3 \otimes M_4$
(e) Factor join tree.

Fig. 17. Step by step transformation of simple communicating finite state machines to a factor join tree. Analysis of this graph can be performed in two components. It is interesting that even though there are 4 machines, analysis only permits 2 major blocks. This is because, as shown in (c), M_2 and M_3 connect all the blocks in their neighborhoods.

V. Future Work

A. Non-Chordal Graphs

While computing join trees for chordal graphs is linear in the number of nodes and links, the general problem is NP-complete. A great deal of research has been conducted on computing join trees for non-chordal graphs. Vertex separators occur in non-chordal graphs and can be exploited for factorization and recursive decomposition.

B. Symbolic Tables for Infinite Sets

One of the limitations of the algorithms as described is that they are applicable to only finite sets. We give an example where that constraint can be lifted using communicating finite state machines. One interesting direction for future research would be to extend this to other classes of infinite sets using symbolic representations.

A state machine is a tuple $M =< Q, \Sigma, \Theta, \delta, \lambda >$. Q is a set of states, Σ is an input alphabet, Θ is a set of labels, $\delta : Q \times \Sigma \rightarrow Q$ is a transition function and $\lambda : Q \rightarrow \Theta$ is a labeling function. Given an initial state $q_0 \in Q$, every machine M_{q_0}, from an input/output perspective can be seen as a mapping $M_{q_0} : \Sigma^* \rightarrow \Theta^*$. Note that this set of acceptable input / output strings is infinite.

It is possible to extend this definition to build networks of communicating state machines as follows. The input alphabet of machine M, Σ can be defined as the Cartesian product of the output alphabets of a set of machines $M_1, \ldots, M_k$, $\Sigma = \Theta_1 \times \ldots \times \Theta_k$ where the machines $M_1, \ldots, M_k$ could also have various relationships between their inputs and outputs. We may construct a factor graph with variable nodes representing output traces from the domains Θ_i for $i = 1 \ldots N$ where there are N machines. The relational nodes of this graph will be the machines themselves. See Figure 17 for an example.

To define the factor graph, the variable nodes become the machines themselves and the constraints reflect the connectivity of the machines. The summary operator is interpreted as a behavior preserving projection over a particular label and the multiplication operator is merely composition of the machines. Summary would likely use Nerode equivalence classes using Hopcroft's state minimization algorithm [15]. This allows us to perform analysis using factor graphs and to enjoy the computational benefits. It would be interesting to create a model checker based on these principles.

Note that in Figure 17, because of the way the system is defined, the state update equations end up creating large coupled neighborhoods. An alternate way of defining a communicating state machine would be to define interactions between machines and express the behaviors of the machines themselves as products of their interactions. This would reduce the amount of coupling between systems and allow further factorization, but it is unclear how expressive such a formalism would be.

C. Loopy Belief Propagation and Approximation Schemes

Loopy belief propagation has been used as an approximate method for summary propagation in Bayesian networks [12]. It would be interesting to see how well it approximates solutions to optimization and constraint satisfaction problems.

D. Tool Creation

There are many tools that we envision being created around this framework. A modeling tool using essentially the database operations described in Section II-F could be very beneficial. It would come with a built in mechanism for creating many different, mathematically justified (see Figure 5), abstracted views of a system. Other tools would be necessary for measuring treewidth and the interactive process described for transforming from the loopy factor graph to a factor join tree (see Section II-C) would be helpful.

VI. Conclusions

We have presented an objective framework for determing system structure. Rather than using heuristics or subjective measurements, structure is defined in a way that is dependent on the logical configuration of the system that minimizes the complexity of answering certain optimizations or inferences on a large class of systems. Within this framework, structure has a certain degree of flexibility, and a unique representation, the clique-separator graph is presented that generates every possible realization. The framework induces mathematical notions of hierarchy and abstraction that coincides well with our intuition.

Complexity is at the root of many problems that are encountered in systems today. A system that cannot be analyzed rigorously will have unanticipated behavior by definition, and such behaviors are often undesirable. It would be interesting to see whether developing tools that make a designer aware of the complexity of analyzing the system improves system quality. In this framework, systems may grow to arbitrary size, provided relationships have strong locality properties. The complexity of analyzing a system is dominated by the size of its maximal clique and only linear in overall size. We suggest that such metrics be incorporated into tools.

Appendix A
Proof of Theorem 1

Proof: Observe that the *recurse* performs a full depth first recursion of the tree starting at the input node. Ignoring lines 14-16 in Figure 3, observe that the result of the recursion is a product of all the function nodes in the tree. This holds because every variable node returns a product of the return values of its children, and every function node returns a product of its own function with the return of every child. By commutativity of multiplication, the resulting product is equivalent to the product in (2).

By commutativity of addition, we can freely rearrange the ordering of the summations occurring in (2). By distributivity of multiplication over addition, if a factor is invariant over a sum, then it is possible to factor it out of the summation, for example, $\oplus_a f(a)f(b)f(c) = f(b)f(c) \oplus_a f(a)$. Thus to show equivalence to (2), it is sufficient to show the following.

1) Every time the summation operation is applied, it captures every occurrence of the summed variable.
2) Every sum occurring in (2) occurs in *Sum-Product*.

(1) The summation operator on line 4 is applied to the entire resulting formula, so it indeed captures any instance. The only other summation operator is on line 15. Line 14 restricts summation to only variables occurring only in the child function. This implies that the variable occurs in a function somewhere under the current node's hierarchy. Using Property 1, we know that the variable cannot occur anywhere else in the hierarchy.
(2) It suffices to show that every variable associated with every variable node gets summed out. By inspecting lines 3-5, we know that if the head node is a variable node, then that particular variable gets summed out. Likewise, if it is a function node, then all of its child node variables get summed out. It remains to be shown that every variable occurring at a *variable* node that is neither the head node nor a child of the head node gets summed out. Let x be such a node. Since x is neither the child of the head node nor the head node itself, it has a grandparent that is a variable node and a parent that is a function node. The variable associated with x is part of its parent node's domain, by the definition of a factor graph, thus it will be summed out by line 15 operating on its grandparent. ■

Appendix B
Proof of Property 2

Proof: We may convert the function f into a tabular representation with one row per possible value of $x_1, \ldots, x_k$ as shown in the table

x_1	$\ldots$	x_k	$f(x_1, \ldots, x_k)$
$\mathbb{X}_1[1]$	$\ldots$	$\mathbb{X}_k[1]$	w_1
$\mathbb{X}_1[1]$	$\ldots$	$\mathbb{X}_k[2]$	w_2
$\ldots$	$\ldots$	$\ldots$	$\ldots$
$\mathbb{X}_1[\lvert\mathbb{X}_1\rvert]$	$\ldots$	$\mathbb{X}_k[\lvert\mathbb{X}_k\rvert]$	w_P

letting $P = |\mathbb{X}_1| \cdot \ldots \cdot |\mathbb{X}_k|$. This operation has $\mathcal{O}(P)$ complexity. To perform the summation, we construct a second table. WLOG, assume we are summing out x_k. This table has as its columns $x_1, \ldots, x_{k-1}, \bigoplus_{x_k} f(x_1, \ldots, x_k)$ and has $P/|\mathbb{X}_k|$ rows. We initialize the function value column to zero. To compute the table weights, we loop through every row of the first table and add the function value to the function value of the row that matches on the first $k-1$ variables in the second table. It is clear by construction that the second table represents the summation function and that the complexity of performing this computation is $\mathcal{O}(|\mathbb{X}_1| \cdot \ldots \cdot |\mathbb{X}_k|)$. The complexity is generated by evaluating all the values in the initial table and it does not matter how many variables we are summing out. Summing out multiple variables has the same complexity but results in a table with fewer columns. ■

Appendix C
Proof of Property 3

Proof: Note that the resulting function $h(x_1, \ldots, x_k, y_1, \ldots, y_m, z_1, \ldots, z_n)$ can be expressed as a table having columns $x_1, \ldots, x_k, y_1, \ldots, y_m, z_1, \ldots, z_n$ which has $P = |\mathbb{X}_1| \cdot \ldots \cdot |\mathbb{X}_k| \cdot |\mathbb{Y}_1| \cdot \ldots \cdot |\mathbb{Y}_m| \cdot |\mathbb{Z}_1| \cdot \ldots \cdot |\mathbb{Z}_n|$ rows. Each row can be filled in by querying functions f and g with the matching values and multiplying the results. This operation is performed P times so the overall complexity is bounded above by P.

■

Appendix D
Proof of Theorem 2

Proof: Line 15 of Figure 3 is executed for every child of a variable node, which means for every function node, and by Property 2, has complexity $|Dom(f)|$ for each function f. Since Property 2 is invariant with respect to the number of summed out variables, the enclosing loop formed by lines 14 and 16 does not affect this complexity. This has an upper bound on complexity of $\max_{k=1\ldots M} |Dom(f_k)|$ for each function node. Line 17 has complexity $|\mathbb{X}|$ and is executed once each time line 14 is. But $\max_{k=1\ldots M} |Dom(f_k)| > |\mathbb{X}|$, so the complexity of lines 13-17 can be bounded by $\max_{k=1\ldots M} |Dom(f_k)|$ times the number of function nodes. Line 23 is executed once per variable node and by Property 3 has complexity $|Dom(f)|$, which again is bounded above by $\max_{k=1\ldots M} |Dom(f_k)|$. Thus the total complexity is bounded above by $n \cdot \max_{k=1\ldots M} |Dom(f_k)|$.

■

ACKNOWLEDGMENT

Research supported in part by NSF grant No. 10092651 “CPS Large: Science of Integration for Cyber Physical Systems”.

REFERENCES

[1] D. Rose and R. Tarjan, “Algorithmic aspects of vertex elimination,” in *Proceedings of seventh annual ACM symposium on Theory of computing*. ACM, 1975, pp. 245–254.

[2] S. Arnborg and A. Proskurowski, “Linear time algorithms for NP-hard problems restricted to partial k-trees,” *Discrete Applied Mathematics*, vol. 23, no. 1, pp. 11–24, 1989.

[3] H. Bodlaender, “Dynamic programming on graphs with bounded treewidth,” *Automata, Languages and Programming*, pp. 105–118, 1988.

[4] H. Loeliger, “An introduction to factor graphs,” *Signal Processing Magazine, IEEE*, vol. 21, no. 1, pp. 28–41, 2004.

[5] F. R. Kschischang, B. J. Frey, and H.-A. Loeliger, “Factor graphs and the sum-product algorithm,” *IEEE Transactions on Information Theory*, vol. 47, pp. 498–519, 1998.

[6] J. Blair and B. Peyton, “An introduction to chordal graphs and clique trees,” *Graph Theory and Sparse Matrix Computations*, vol. 56, 1993.

[7] S. Arnborg, D. G. Corneil, and A. Proskurowski, “Complexity of finding embeddings in a k-tree,” *SIAM J. Algebraic Discrete Methods*, vol. 8, pp. 277–284, April 1987. [Online]. Available: http://dx.doi.org/10.1137/0608024

[8] R. Tarjan and M. Yannakakis, “Simple linear-time algorithms to test chordality of graphs, test acyclicity of hypergraphs, and selectively reduce acyclic hypergraphs,” *SIAM Journal on Computing*, vol. 13, p. 566, 1984.

[9] L. Ibarra, “The clique-separator graph for chordal graphs,” *Discrete Appl. Math.*, vol. 157, pp. 1737–1749, April 2009. [Online]. Available: http://portal.acm.org/citation.cfm?id=1530895.1531045

[10] P. S. Kumar and C. E. V. Madhavan, “Clique tree generalization and new subclasses of chordal graphs,” *Discrete Applied Mathematics*, vol. 117, no. 1-3, pp. 109 – 131, 2002. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0166218X0000336X

[11] E. Amir and S. McIlraith, “Partition-based logical reasoning,” in *Principles of Knowledge Representation and Reasoning: Proceedings of the 7th International Conference*, 2000, pp. 389–400.

[12] J. Pearl, *Probabilistic reasoning in intelligent systems: networks of plausible inference*. Morgan Kaufmann, 1988.

[13] T. Nagata, H. Sasaki, and R. Yokoyama, “Power system restoration by joint usage of expert system and mathematical programming approach,” *Power Systems, IEEE Transactions on*, vol. 10, no. 3, pp. 1473–1479, 1995.

[14] T. Nagata and H. Sasaki, “A multi-agent approach to power system restoration,” *Power Systems, IEEE Transactions on*, vol. 17, no. 2, pp. 457–462, 2002.

[15] J. E. Hopcroft, “An n log n algorithm for minimizing states in a finite automaton,” Stanford, CA, USA, Tech. Rep., 1971.