Implementation of a generic Architecture for Local Computation[1]

Marc Pouly[2]

Abstract. Valuation algebras are the foundation of a widespread theory unifying different fields of modern research and culminating in the power of local computation. Although this theory rejoys in great interest from a pure theoretical perspective, many questions were originally motivated by practical experiences. The aim of this paper is to return to this practical starting point by illustrating how these results can be used to design generic software for local computation. Thereby, the novelty consists in passing the high level of abstraction given by the mathematical theory to the implementation itself. Different architectures of local computation are realized independently to valuation algebra instances and join tree construction, which leads to a modular construction system inspired software project, available for anyone being interested in local computation.

1 INTRODUCTION

A valuation algebra [1] represents an algebraic structure unifying various instances from different fields of application. Famous examples of such instances are known from logic, probability theory, relational databases and many more. All of them have in common that they represent somehow knowledge and information, so that inference is a central topic when dealing with valuation algebras. This is the point where local computation comes into play because it provides an efficient tool for solving inference problems based on a graphical structure named join tree. Local computation is realized in different architectures, each one taking advantage of additional properties of the underlying valuation algebra.

The beauty of the theory around valuation algebras and local computation is the high level of abstraction that unifies this large number of applications. Many software projects have been realized in this domain, and they implement parts of this theory in a very elaborated and high-performance way. But each of these implementations covers only some specific aspects of this theory. It is either restricted to a specific valuation algebra instance or to one architecture of local computation. Therefore, our interest concerns the realization of a software architecture [12] preserving this high level of abstraction. Such a piece of software must comply many requirements. Typically, each user can implement the own valuation algebra instance and integrate it to perform local computation. Furthermore, each architecture of local computation can be combined with every valuation algebra instance satisfying the mathematical restrictions. Often, we are keen to compare different join tree construction algorithms and of course they should be unattached by the different architectures of local computation. Meeting all these requirements leads to a modular construction system inspired architecture, allowing the user to customize it for the current purposes.

The first part of this paper gives a formal definition of a valuation algebra together with some properties being important in the context of local computation. It follows a short description of the inference problem and introduces the four most popular architectures for local computation. The second part is dedicated to the software architecture itself. Starting with a generic implementation of a valuation algebra, we will introduce a suitable data structure to represent join trees. Local computation can then be implemented recursively on the nodes of these join trees and the different architectures of local computations are reduced to different node types of a join tree. At last, we will take a closer look at the implementation of join tree construction. Construction algorithms must be independent of the join tree node architecture, and we will present an implementation strategy allowing to interchange construction algorithms.

2 VALUATION ALGEBRAS

The basic elements of a valuation algebra are so-called *valuations*. Intuitively, a valuation can be regarded as a representation of knowledge about the possible values of a set of variables. Thus it can be said that each valuation ϕ refers to a finite set of variables $d(\phi)$, called its *domain*. For an arbitrary set s of variables, Φ_s denotes the set of valuations ϕ with $d(\phi) = s$. With this notation, the set of all possible valuations corresponding to a finite set of variables r can be defined as

$$\Phi = \bigcup_{s \subseteq r} \Phi_s. \tag{1}$$

Furthermore, let D be the lattice of subsets (the powerset) of r. For a single variable X, Ω_X denotes the set of all its possible values. We call Ω_X the *frame* of the variable X. In an analogous way, we define the frame of a non-empty variable set s by the Cartesian product of frames Ω_X of each variable $X \in s$,

$$\Omega_s = \prod_{X \in s} \Omega_X. \tag{2}$$

The elements of Ω_s are called *configurations* of s. The frame of the empty variable set is defined by convention as $\Omega_\emptyset = \{\diamond\}$. This summarizes the most important notations and allows to define formally a *valuation algebra* by three fundamental operations and a set of axioms.

[1] Research supported by grant No. 20-67996.02 of the Swiss National Foundation for Research.

[2] University of Fribourg, Switzerland, email: marc.pouly@unifr.ch

2.1 A formal definition

Let Φ be a set of valuations with their domains in D. We assume the following operations defined on Φ and D:

1. *Labeling:* $\Phi \to D$; $\phi \mapsto d(\phi)$,
2. *Combination:* $\Phi \times \Phi \to \Phi$; $(\phi, \psi) \mapsto \phi \otimes \psi$,
3. *Marginalization:* $\Phi \times D \to \Phi$; $(\phi, x) \mapsto \phi^{\downarrow x}$.

These are the three basic operations of a valuation algebra. If we interpret valuations as pieces of knowledge, the labeling operation tells us the questions to which this knowledge refers. Combination can be understood as aggregation of knowledge and marginalization as the focussing of some knowledge onto a smaller set of questions.

We impose now the following set of axioms on Φ and D:

1. *Associativity and Commutativity:* Φ is associative and commutative under combination.
2. *Labeling:* For $\phi, \psi \in \Phi$,

$$d(\phi \otimes \psi) = d(\phi) \cup d(\psi). \tag{3}$$

3. *Marginalization:* For $\phi \in \Phi$, $x \in D$, $x \subseteq d(\phi)$,

$$d(\phi^{\downarrow x}) = x. \tag{4}$$

4. *Transitivity:* For $\phi \in \Phi$ and $x \subseteq y \subseteq d(\phi)$,

$$(\phi^{\downarrow y})^{\downarrow x} = \phi^{\downarrow x}. \tag{5}$$

5. *Combination:* For $\phi, \psi \in \Phi$ with $d(\phi) = x$, $d(\psi) = y$,

$$(\phi \otimes \psi)^{\downarrow x} = \phi \otimes \psi^{\downarrow x \cap y}. \tag{6}$$

Axiom (1) indicates that Φ is commutative and associative. The labeling axiom tells us that the domain of a combination is the union of both valuations' domains. The marginalization axiom says that the resulting valuation of a marginalization on a domain x is itself a valuation on this domain x. Transitivity means that marginalization can be performed in two or more steps. At last, the combination axiom says that, instead of combining two valuations and marginalizing the result on the first valuation's domain, we can marginalize first the second valuation on the intersection of both valuations' domains and perform the combination in a second step.

Definition 1 *A system (Φ, D) together with the operations of labeling, marginalization and combination satisfying the above set of axioms is called a (labeled) valuation algebra.*

This formal definition of a valuation algebra opens the floodgates to a widespread and profound theory. At this place we will restrict our efforts to a handful of further definitions which constitute additional properties a valuation algebra may have. In the context of local computation, these properties are exploited to improve the corresponding computations.

2.2 Properties of valuation algebras

We start by a very appealing property which links valuation algebras to the notion of information.

Definition 2 *A valuation algebra is said to be idempotent if for every valuation ϕ and $t \subseteq d(\phi)$,*

$$\phi \otimes \phi^{\downarrow t} = \phi. \tag{7}$$

Loosely spoken, idempotency says that combining a piece of information with a part of itself gives nothing new. This is a very natural property in the context of information and therefore idempotent valuation algebras are called *information algebras*.

Another property being important for local computation is *division*, although in general no division for valuation algebras exists. Nevertheless, we distinguish two concepts of division in the context of valuation algebras. Because valuation algebras form a semi-group, we know that if a certain instance is a union of a family of disjoint groups, then inverse elements exist. Such an instance is called *regular*. Other instances which are not regular can be embedded in a semi-group being such a union. In this case, we call the valuation algebra *separative*. We refer to [1] for an extensive study of division in the context of valuation algebras.

3 LOCAL COMPUTATION

The formulation of the *projection problem* motivates our interest in local computation.

3.1 The projection problem

We have seen that a valuation can be interpreted as some kind of knowledge container. Hence, a *knowledge-base* is represented by a set of such valuations $\{\phi_1, \phi_2, \ldots, \phi_m\}$. Our basic point of interest consists in using this knowledge-base to answer one or more *queries*. For this purpose, we need to aggregate all knowledge in the knowledge-base to focus afterwards the result on the domain of interest given by the query. This is summarized formally by the following equation:

$$\phi^{\downarrow s} = (\phi_1 \otimes \phi_2 \otimes \cdots \otimes \phi_n)^{\downarrow s} \tag{8}$$

with

$$s \subseteq \bigcup_{i=1}^{n} d(\phi_i) \tag{9}$$

being the query of interest. This problem is often called *projection problem* or *inference problem*.

Solving the projection problem in a straightforward way by first computing the combination of all elements in the knowledge-base and then in a second step performing a single marginalization on the result ϕ is hardly a feasible way. Although this naive computational strategy would lead to the answer of the query, its efficiency has to be scrutinized. It is a well-known phenomenon that the complexity of combination and marginalization tends to increase exponentially with the size of the domains. By first computing the combinations, the size of the domain will increase with each step due to the labeling-axiom.

3.2 Architectures of local computation

A way out of this dilemma is given by a computational scheme called *local computation*. The remarkable idea behind local computation is that it organizes the computations in such a way that each operation takes place in a domain any larger than the domains appearing in the factors of the combination. That is why the algorithms are called *local*. More concretely, local computation consists in eliminating successively all variables not contained in the query and this is possible

due the the combination-axiom. So, if t is the domain of the combined factors,

$$\phi^{\downarrow t-\{X\}} = (\bigotimes_{i:X\in d(\phi_i)} \phi_i)^{\downarrow u-\{X\}} \otimes (\bigotimes_{i:X\notin d(\phi_i)} \phi_i). \quad (10)$$

u is the joint domain of all factors containing the variable X and the formula shows that the elimination of X is restricted to the domain u, which is in general much smaller than t. All other valuations need not yet to be combined. Repeating this procedure of *variable elimination* until only the variables of the query s remain, leads naturally to the solution of the projection problem. This algorithm has been introduced by [4] and carries the name *fusion algorithm*. [6] proposed a visualization of this fusion process, which results in a graphical structure named *join tree*.

Definition 3 *A tree whose nodes are domains s is called a join tree, if for any pair of nodes s' and s'', if $X \in s' \cap s''$, then $X \in s$ for all nodes on the path between s' and s''.*

This is the natural order of introducing the above mentioned concepts. Nevertheless, an alternative and more sophisticated way to solve the projection problem can be derived by reversing this order. Instead of regarding join trees just as a representation of the fusion algorithm, they will become the center of the further development of this theory. To start with, we consider a *join tree cover* of the factors in the projection problem.

Definition 4 *A join tree is covering a combination*

$$\psi_1 \otimes \psi_2 \otimes \cdots \otimes \psi_n \quad (11)$$

if for all ψ_i there is a node v such that $d(\psi_i) \subseteq d(v)$.

For every factorization of ϕ there exists a join tree such that each factor ϕ_i can be associated to a node v of this join tree respecting $d(\phi_i) \subseteq d(v)$. This follows from the fusion algorithm. If multiple factors are associated to the same node, they will first be combined. For computational reasons, we interdict nodes staying empty. Therefore, we adjoin an empty element e to the underlying valuation algebra with $\phi \otimes e = \phi$ for all $\phi \in \Phi$ and $d(e) = \emptyset$. One can show that this extended version is still a valuation algebra. This element e is then associated with every empty node. Next, we number the nodes of the join tree by $i = 1, \ldots, m$, such that $i < j$ if j lies on the path from node i to node m. Node m will be called the join tree's *root*. The above factor distribution results then in a join tree cover with the additional property that each factor refers to exactly one node.

$$\phi = \phi_1 \otimes \cdots \otimes \phi_n = \psi_1 \otimes \cdots \otimes \psi_m, \quad (12)$$

with ψ_i being the content of node i. This kind of join tree cover is called a *join tree factorization*. We remark that such a join tree factorization can always be found and refer to [1] for further details on join tree construction and factorizations. In the sequel, we restrict our considerations to join tree factorizations, and the projection problem of a join tree factorization to the root node m of the join tree will attract our interest.

3.2.1 The Shenoy-Shafer architecture

Complementary to the fusion algorithm, we will describe a message-passing algorithm to solve the above mentioned problem working on the nodes of the join tree. To facilitate notations, we denote the unique neighbor of node i towards the root by $ch(i)$ (child of node i) and the set of all other neighbors $pa(i)$ (parents of node i). Nodes with an empty set of parent nodes are called *leaves*. The domain of node i is referred by d_i. The algorithm itself consists of a step-wise procedure, which changes the content of node i at step $i = 1, \ldots, m$. The content of node j at step i is denoted by $\psi_j^{(i)}$, initially $\psi_j^{(1)} = \psi_j$ and its domain $s_i = d(\psi_i^{(i)})$. Then, at step i, node i computes the message

$$\mu_{i\to ch(i)} = \psi_i^{(i)\downarrow s_i \cap d_{ch(i)}}. \quad (13)$$

This message is sent to the unique child of node i which updates its node content to

$$\psi_{ch(i)}^{(i+1)} = \psi_{ch(i)}^{(i)} \otimes \mu_{i\to ch(i)}. \quad (14)$$

The storages of all other nodes do not change at step i, $\psi_j^{(i+1)} = \psi_j^{(i)}$ for all $j \neq ch(i)$. This procedure of inward message passing is called *collect algorithm*.

Theorem 1 *At the end of the collect algorithm applied to a join tree factorization, the root node contains the solution of the projection problem,*

$$\psi_m^{(m)} = \left(\bigotimes_{i=1}^{m} \psi_i\right)^{\downarrow s_m} = \phi^{\downarrow s_m}. \quad (15)$$

A detailled proof of this theorem can be found in[1] and [2].

The collect algorithm provides an efficient way to solve the projection problem relative to the root domain of the join tree. Often, the projection of the same factorization has to be computed but to another node of the join tree. In this case, we only have to take the corresponding node as the new root node and reorient all edges towards this root. Obviously, only the edges on the path between the old and the new root are concerned. This implies that most of the foregoing computations can be reused. In fact, we can organize the computations in such a way that the projections on all node domains of the join tree can be computed essentially only by the double effort of the collect algorithm. For this purpose, we install mailboxes between two neighboring nodes i and j of the join tree storing the messages $\mu_{i\to j}$ and $\mu_{j\to i}$. Then, the message $\mu_{i\to j}$ is defined as,

$$\mu_{i\to j} = \left(\psi_i \otimes \bigotimes_{k\in ne(i), k\neq j} \mu_{k\to i}\right)^{\downarrow s_i \cap s_j}. \quad (16)$$

with $ne(i)$ denoting the set of neighbors of node i. This computational scheme is often referred as *distribute algorithm*.

Theorem 2 *At the end of the distribute algorithm*

$$\phi^{\downarrow s_i} = \psi_i \otimes \left(\bigotimes_{k\in ne(i)} \mu_{k\to i}\right), \quad (17)$$

for each node i of the join tree.

A detailled proof of this theorem can be found in [1].

This organization of the computations is called the *Shenoy-Shafer architecture* [3]. We foreshadow at this place that the Shenoy-Shafer architecture presents a lack of efficiency when dealing with nodes with more than three neighbors. In this case, some combinations are

executed multiple times. [5] showed that these redundant computations can be anticipated by dealing with so-called *binary join trees*. A binary join tree is a join tree with no node having more than three neighbors. Being confronted with valuation algebras having further properties, one can apply more sophisticated architectures taking advantage of them.

3.2.2 The idempotent architecture

When the underlying valuation algebra is idempotent, the distribute algorithm can be simplified in an essential way. We first execute the collect algorithm without any changes. Then we pass messages outward in the inverted direction of the edges. If node $j = ch(i)$ contains valuation χ_j after this node has received the outward message from its child, then it passes message

$$\mu_{ch(i)\to i} = \chi_{ch(i)}^{\downarrow s_i \cap s_{ch(i)}} \tag{18}$$

to node i and this message will be combined there with the node content presented from the collect phase, we get

$$\chi_i = \psi_i^{(i)} \otimes \mu_{ch(i)\to i}. \tag{19}$$

[1] and [2] shows that this modified distribute algorithm leads indeed to the same result as the Shenoy-Shafer architecture.

3.2.3 Architectures with division

If the underlying valuation algebra presents some concept of division, two further architectures can be applied, namely the *Lauritzen-Spiegelhalter architecture* [7] and the *HUGIN architecture* [8].

In the Lauritzen-Spiegelhalter architecture we first execute the collect algorithm with the extension that in node i, the node content $\psi_i^{(i)}$ is divided by the outgoing message to $ch(i)$. So, we update the content of node i to

$$\psi_i \otimes \bigotimes_{j \in pa(i)} \mu_{j\to i} \otimes \left(\mu_{i\to ch(i)}\right)^{-1}. \tag{20}$$

After the collect algorithm, the distribute algorithm follows exactly as in the idempotent architecture. A proof of the correctness of this architecture is shown in [1].

The idea of the HUGIN architecture is quite similar to the Lauritzen-Spiegelhalter architecture except that the division is performed in the distribute phase. The collect algorithm is executed as originally defined. However, we introduce an additional node between all nodes i and $ch(i)$, the so-called separator, which stores the message $\mu_{i\to ch(i)}$ sent during the collect phase. In the following distribute phase, each node sends its messages as in the idempotent architecture. When a message reaches the separator, it is first divided by the inverse of the separator's content,

$$\mu_{ch(i)\to i} \otimes \left(\mu_{i\to ch(i)}\right)^{-1}. \tag{21}$$

The resulting message arrives then at the node and will be combined with its content $\psi_i^{(i)}$. We refer again to [1] for a detailed proof of correctness of this architecture.

3.3 Scaling in local computation

In many applications, we are interested in scaled valuations for semantic reasons. Probability potentials are perhaps the most famous example and show clearly that this aspect is not negligible in the context of local computation. So, for a given join tree factorization

$$\phi = \psi_1 \otimes \cdots \psi_m, \tag{22}$$

we are not only interested in the marginal $\phi^{\downarrow s_i}$ to some domain s_i, but rather in the scaled version of this marginal, denoted by $(\phi^{\downarrow s_i})^{\downarrow} = \phi^{\downarrow s_i} \otimes (\phi^{\downarrow \emptyset})^{-1}$. We refer to [1] of more details.

The most important result shown in [1] is that scaling can be reached in the Shenoy-Shafer, Lauritzen-Spiegelhalter and HUGIN architecture by performing one single division in the root node after the collect algorithm. Of course, we consider only valuation algebras having some concept of division. More concretely, we replace the valuation ψ_m in the root node by

$$\psi_m^{\downarrow} = \psi_m \otimes (\phi^{\downarrow \emptyset})^{-1} \tag{23}$$

Then, the corrsponding distribute phase is executed as usually. In the case of idempotent valuation algebras scaling is performed implicitly, and no changes to the algorithms are necessary.

3.4 A word about building join trees

Going back to the beginning of this theory, we remark that the architectures of local computation demand a join tree factorization as starting point. This can be achieved by constructing a join tree directly from the initial set of valuation (knowledge-base), and a procedure is given implicitly by the fusion algorithm. We modify the fusion algorithm in such a way that we leave out all real computations and work only on the domains of the initial valuation set. We refer to [9] for more informations about join tree construction in general.

A central point when dealing with join tree construction is the choice of the elimination sequence. The fusion algorithms assumes a given variable elimination sequence and each elimination sequence, leads to a different join tree. Generally, we are interested in join trees whose nodes have small domains because local computation on such join tree is more efficient. Therefore, the problem of finding optimal join trees arises, and this problem turned out to be NP-hard [10]. Nevertheless, there exists a number of heuristics to find "good" elimination sequences, especially [9].

4 A GENERIC FRAMEWORK FOR LOCAL COMPUTATION

The interest in a generic software architecture for local computation has already been discussed in the introductory chapter of this paper, and after the former abridgment of the theory we are familiar with all necessary components which have to be part of such an architecture. This second part will therefore be dedicated to the implementation process itself and, to do so, we will give a requirement-based description of the underlying implementation strategies without going into programming details. The requirements for this architecture are derived from the theoretical background and will serve as motivation for the design strategies subsequently presented.

From the developer's point of view it has been decided to use a programming language offering facilities in the development of generic programs such as object oriented languages. More concretely, we chose the JAVA programming language for this realization.

4.1 Representing valuation algebras

The starting point is the representation of a valuation algebra based on its mathematical components, namely variables, domains and valuations. Precisely, this will be done by a framework of interfaces, representing the generic nature of these components. This skeleton allows then to create and embed new valuation algebra instances by implementing the corresponding interfaces. Figure 1 shows the class diagram defining a valuation algebra. The arrows within this diagram express a relationship of aggregation ("has a"-relationship).

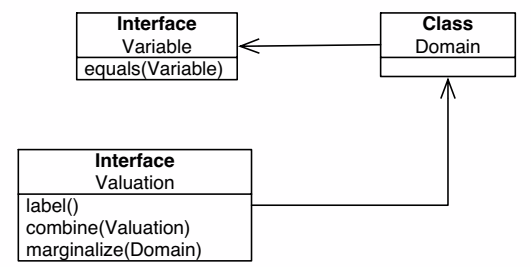

Figure 1. Generic representation of a valuation algebra.

Variables are represented by an interface which prescribes just an identity check. Based on this abstract representation of variables, we can give a concrete pre-implementation of domains. The class `Domain` implements a common set of variables offering all fundamental set operations. Following exactly the mathematical definition, we specify next a valuation by the three basic operations of labeling, combination and marginalization. Therefore, a new valuation algebra instance is fully determined by an implementation of the `Variable` and `Valuation` interfaces.

4.1.1 Properties of a valuation algebras

In the theory part we showed that additional properties of valuation algebras play an important role in the context of local computation. So we are obliged to extend the class diagram to take these properties into consideration. Two requirements in accordance with this extension are central. First, a valuation algebra instance can theoretically have any possible combination of properties. Hence, no restriction on the number of properties or on their combination is acceptable. The second requirement concerns extensibility. We presented the four most famous architectures of local computation but this does not imply that there are no others. Such new architectures are perhaps based on other properties of the underlying valuation algebra. This is only one scenario showing the importance of an extensible property management. It is clear that adding new properties should not influence existing code. Figure 2 shows an extension of the class diagram taking properties into consideration. Note that the arrows whose tip is an empty triangle stand for inheritance ("is a"-relationship). This class hierarchy fulfills the formerly mentioned requirements.

4.2 Representing join trees

It is very common to represent trees by linked set of tree nodes, and this is exactly the approach that has been chosen in the case of join trees. So, a join tree is basically a pointer to its root node, and the nodes are linked among each other to effectuate the tree structure.

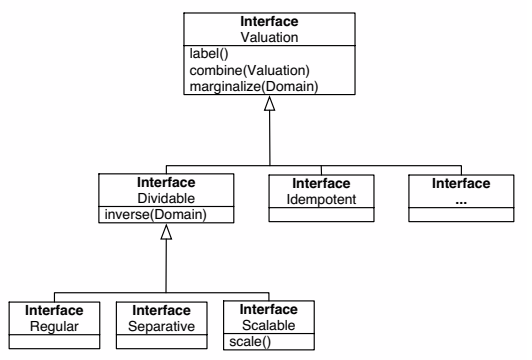

Figure 2. Extended class diagram representing a valuation algebra.

Local computation, independently of the architecture, is defined by message passing algorithms working on the nodes of a join tree. With the above representation of join trees by linked sets of nodes, one can reduce the different architectures of local computation to different node types. The idea is that each node type carries out the algorithms corresponding to its architecture of local computation. The message passing algorithms themselves are implemented recursively for each node type. Typically, the pattern of the collect algorithm is given by the following steps:

1. Node i waits until it has received messages from all parent nodes.
2. Node i updates its storage (according to the architecture).
3. Node i computes the new message (according to the architecture).
4. Node i sends the message to its unique child.

Proceed in the inverse order for the distribute algorithm. During the collect phase we perform a bottom-up tree traverse and in the distribute phase a top-down traverse respectively. To guarantee that each node can communicate with all neighbors, the join tree nodes are implemented as doubly linked nodes. Figure 3 summarizes the relationship between join trees, join tree nodes and architectures of local computation on design level.

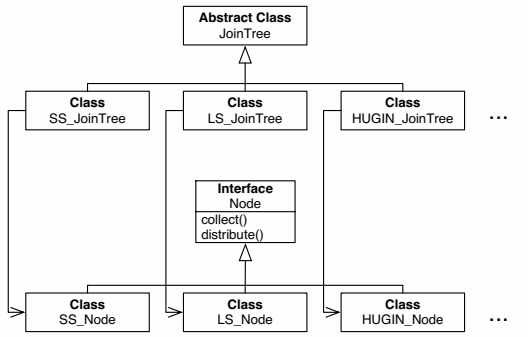

Figure 3. Different architectures of local computation are reduced to different join tree node types.

4.3 Join tree construction

Join tree construction is implemented by a modified version of the fusion algorithm working only on the domains of the initial knowledgebase. But the fusion algorithm demands a certain variable elimination

sequence, and we know that each elimination sequence will construct a different join tree. A good degree of optimality of join trees is the size of the largest node domain in the join tree, but we have seen that finding optimal join trees is known to be NP-hard. There exists a large number of heuristics for this task, and each heuristic defines another join tree construction algorithm. This explains our interest in a system allowing to swap construction algorithms.

Another challenge is given by the implementation of local computation. The architecture of local computation is determined by the appropriate node type of the join tree. But the construction algorithm should be independent of the join tree's node type. In other words: whenever the construction algorithms attaches a new node to the join tree, we must choose the node type corresponding to the join tree's architecture.

All these requirements are handled by the so-called *join tree factory*, whose main task is to organize the join tree construction. Figure 4 illustrates how construction algorithms and join tree architectures are controlled by the join tree factory. Construction algorithms are

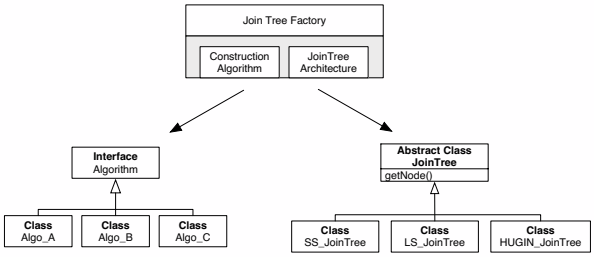

Figure 4. The join tree factory handles construction algorithms and architectures of local computation to build join trees.

abstracted by a common interface and the join tree factory uses this interface to construct join tree. The user can change the construction algorithm by setting another `Algorithm` object to the factory. This is an implementation of the STRATEGY design pattern [11]. We explained briefly before the one-to-one relationship between join trees types and node types. This relationship is implemented by the method `getNode()` in the `JoinTree` class, returning the node type corresponding to this architecture. Whenever the construction algorithm creates a new node, this method will be called and delivers a `Node` object according to the current architecture. This is again an implementation of the STRATEGY design pattern and we refer to [11] for detailed description. The join tree construction ex-

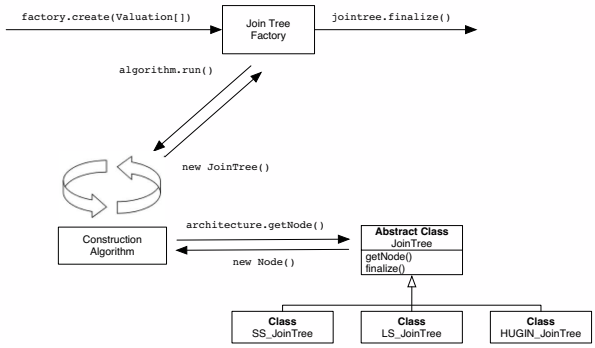

Figure 5. Join tree construction - the big picture.

ecutes without any user interaction, but at the end of the construction algorithm, just before returning the join tree object, the factory calls the `finalize()` method of the class `JoinTree`. By default, this method has an empty pre-implementation and allows the user to perform last actions on the almost completed `JoinTree` object by overwriting this method. This dynamical loading of the appropriate node type is a realization of the BUILDER design pattern [11]. At the end, one can execute collect and distribute on the returned join tree by calling `jointree.startPropagation()`. The process of join tree construction organized by the join tree factory is summarized in figure 5.

4.4 A graphical user interface

Originally, the implementation of this generic architecture for local computation was motivated by a pure academic interest. But during the development process it came out that such a piece of software could also be viable for educational purposes. These thoughts were the birth of a graphical user interface (GUI) to control the underlying architecture and to display join tree structures graphically. Figure 6 shows a join tree image exported from this GUI.

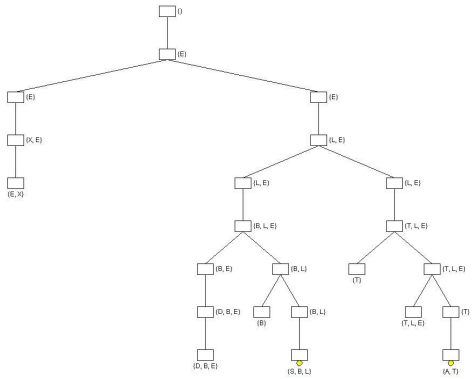

Figure 6. An image of a join tree displayed in the graphical user interface.

5 CONCLUSION

This generic architecture for local computations specifies valuation algebras by a set of interfaces, and each user can create own instances by implementing these interfaces. The basic hierarchy is extended by a system of interfaces taking additional properties of a valuation algebra into consideration. This implementation is very close to the abstract mathematical framework of a valuation algebra. Furthermore, local computation is reduced to different kinds of join tree node types and join trees are created in such a way that the creational algorithm is independent of the join tree's architecture. The architecture itself and even the construction algorithm are exchangeable. This summarizes the two-fold interpretation of the label "generic".

In its current state, the software contains a generic implementation of the four architectures of Shenoy-Shafer, Lauritzen-Spiegelhalter, HUGIN and the idempotent architecture. Instances exist so far for probability potentials [12], propositional logic and Gaussian hints

[13].

The field of research dealing with valuation algebras is very far reaching, and this architecture is only a first step in the direction of realizing a software tool based on an abstract mathematical framework. Nevertheless, it shows the practical benefit of this abstract theory and puts a unifying, general framework at the disposition of a developer who wants to apply local computation to a particular instance of a valuation algebra.

REFERENCES

[1] J. Kohlas, *Information Algebras: Generic Structures for Inference*, Springer 2003.

[2] J. Kohlas, *Valuation Algebras induced by Semirings*, Internal working paper no. 04-03, April 2004.

[3] P. P. Shenoy and G. Shafer, *Axioms of probability and belief-function propagation.*, In R. D. Shachter, T. S. Levitt, L. N. Kanal, and J.F. Lemmer, editors, *Uncertainty in Artificial Intelligence 4*, pages 169–198, Amsterdam, 1990. North-Holland.

[4] P.P. Shenoy, *Valuation-Based Systems: A Framework for Managing Uncertainty in Expert Systems.*, Pages 83-104 of: Zadeh, L.A., Kacprzyk, (eds), Fuzzy Logic for the Management of Uncertainty. John Wiley & Sons.

[5] P.P. Shenoy, *Binary Join Trees for Computing Marginals in the Shenoy-Shafer Architecture.*, Int. J. of Approximate Reasoning, 17, 239-263, 1997 b).

[6] G. Shafer, *Probabilistic Expert Systems.* Society for Industrial and Applied Mathematics, Philadelphia, 1996.

[7] S.L. Lauritzen & D.J. Spiegelhalter, *Local Computations with Probabilities on Graphical Structures and their Application to Expert Systems.*, J. of Royal Stat. Soc., 50(2), 157-224, 1988.

[8] F.V. Jensen & S.L. Lauritzen & K.G. Olesen, *Bayesian Updating in Causal probabilistic Networks by Local Computation.*, Computational Statistics Quarterly, 4, 269-282, 1990.

[9] N. Lehmann., *Argumentation Systems and Belief Functions.*, PhD thesis, Department of Informatics, University of Fribourg, 2001.

[10] S.Arnborg, D.Corneil, and A.Proskurowski, *Complexity of finding embeddings in a k-tree.*, Technical report, 1987.

[11] E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns, Elements of reusable object-oriented Software.*, Addison-Wesley 1995.

[12] M. Pouly, *A generic Architecture for Local Computation*, Master Thesis, University of Fribourg, Institute for Informatics, Switzerland.

[13] Ch. Eichenberger, *Implementing Gaussian Hints*, Master Thesis, University of Fribourg, Institute for Informatics, Switzerland.