8. KressArray Design Space Exploration

The KressArray architecture family is defined by the design space described in section 8. This design space allows a large variety of KressArray architectures, which differ in their complexity and their performance. For a given purpose, there may be one specific architecture, which resembles the best trade-off between performance and implementation complexity. The process of finding this architecture is done by a design space exploration process, comparable to those described in section 5. This section introduces the concepts and tools of the KressArray Xplorer design space exploration framework for KressArray architectures.

The Xplorer framework employs the MA-DPSS described in sectio n7, which can be used to map applications onto a given KressArray architecture. As the MA-DPSS is used mostly in a status evaluation step, as described in the general discussion in section 5.1, additional tools are employed to perform the other tasks of the exploration process. These tasks include gathering of statistic data, generation of suggestions for improvement, and interactive control of the whole process by the designer.

In the following section, the concepts of the Xplorer framework are presented. Then, an overview on the system is given, with the relevant tools for the exploration process being described afterwards. In the next section, the analyzing of the mapping result is discussed, followed by a detailed description of the technique used for the generation of design suggestions. Finally, an overview on the Xplorer graphical user interface is given.

8.1 Concepts

The previous work done on design space exploration described in section 5.2 shows a variety of approaches to meet the problem of finding a suitable hardware structure for a given set of requirements. For the KressArray Xplorer, the task consists in finding the best trade-off between the complexity of the hardware and an optimization objective, as e.g. the performance. The optimal architecture in respect to this trade-off is expected to depend on the application to be executed on the KressArray. However, as being reconfigurable, a given KressArray architecture instance is supposed to execute several different applications, so the exploration process is capable to consider a set of applications rather than a single one. On the other hand, the requirements may be different for each intended

application. For example, the KressArray could be aimed as a unit to execute mainly one task requiring real-time performance, while other applications are executed in a different mode of the device, which does not require hard timing limits. Those other applications are in that respect of lesser importance for the exploration process.

Furthermore, the KressArray design space is defined at a high abstraction level, which does not deal with the physical implementation of ALUs or communication resources. The actual implementation may imply additional constraints or measures for the quality of an architecture in regard to the optimization objective.

Due to the variety of factors which influence the exploration, a fully automated approach seems not promising. Instead, an interactive strategy has been selected, which is based on an iterative refinement of a start architecture. In each iteration, the designer is assisted with a suggestion how to improve the architecture in respect to the optimization objective, which may be heeded or rejected, so the control over the exploration process is retained by the user.

Status Evaluation. Another issue for the design of the Xplorer framework are the realization of the two main operations of design space exploration described in section 5.1, the status evaluation and the status generation. For the status evaluation, different techniques are used by existing design space exploration systems, like analytical or abstract models as in the systems described in section 5.2.1 and sectio n5.2.4.

Evaluating a given KressArray architecture has to happen with regard to a given application or application set. Thus, it is reasonable to actually map the application onto the architecture in order to get usable performance estimations and other statistic data, like the percentage of idle connections. As a consequence, each iteration requires a synthesis step. This can be tolerated, as the computation time for the synthesis lies in the range of minutes due to the relative low cell count, and a relative low number of iterations is expected to be necessary.

However, this kind of exploration strategy works for a single application, while the Xplorer framework is designed for application sets. Thus, one target application has to be selected from the domain for the exploration process, such that the optimized architecture for this application will also satisfy the requirements of the other applications of the domain. This selection process has to be supported by the framework by an according complexity estimation. Also, the assumption that the optimized architecture satisfies the other applications has to be verified at the end.

Status Generation. For the generation of new architectures based on the knowledge of the evaluation step, a widely-used method consists in employing a kind of expert system or generally a rule-based decision methodology [PaMc90], [HaKa92], [KnPa91], [GPR98]. As this approach gives a natural way of expressing human knowledge on how the design could be improved, this concept is also being employed in the Xplorer framework. Furthermore, the decision methodology is based on fuzzy reasoning [Gain76] with the relevant measures modeled as fuzzy variables [Pedr96], which allows imprecise estimation data to be processed. Based on fuzzy rule systems, suggestions are created, which are presented to the designer, who may then heed them, or perform changes on the architecture on his own. After applying any changes, the process will typically iterate by mapping the application again on the new architecture. However, certain modifications may affect the complexity of the application in regard to the exploration objective, making a re-selection of the application to be explored necessary. The concepts discussed above lead to a general exploration strategy pictured in figure 8-1.

Basic Strategy. The general Xplorer strategy shown in figure 8-1 consists of three phases. In the first phase, the exploration process is prepared by compiling all applications of the domain into an intermediate format suitable for further processing, which is described in section A.1. An architecture estimation will create an initial architecture for the following exploration process, taking into account the minimal requirements of all the applications given. These requirements are properties like the number of operators needed or the minimal number of connections on each rDPU. One application is then selected for the following exploration phase. The selection may be based on the tightest requirements or on other preferences of the designer.

In the exploration phase, several architectures are examined by mapping the application onto them, followed by an analysis of the mapping result. The data gathered in the analysis is then used by a fuzzy rule based inference engine to derive suggestions for the modification of the architecture. These suggestions consist in a ranking of possible actions for architecture changes, thus giving the user an idea which action would lead to an architecture enhancement. The process iterates by mapping the application again onto the new architecture, until it satisfies the exploration objective.

In the final phase, the other applications of the domain have to be mapped onto the optimized KressArray in order to verify the solution. If it turns out, that the architecture does not satisfy one or more applications of the domain, the exploration process is repeated with one of the failing applications, until finally a suitable architecture has been found.

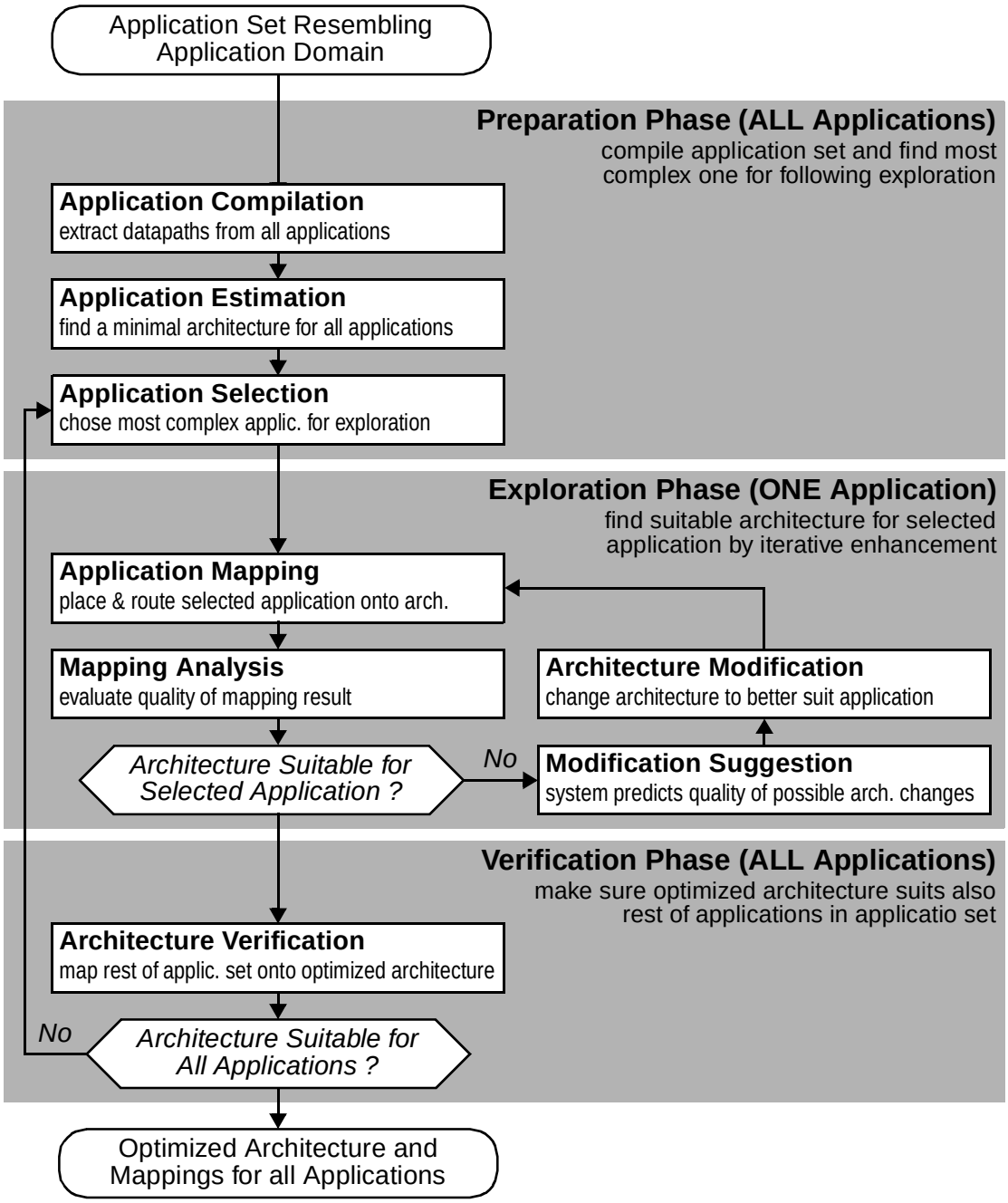


Figure 8-1: General Xplorer strategy

Relations to Previous Work. Existing design space exploration and design guidance systems show techniques similar to the concepts of the Xplorer framework. The interactive design guidance by Guerra et al. described in [GPR98] bears the closest similarity to the global approach using iterative

refinement and design suggestions. However, while the system by Guerra is meant to optimize an architecture for a single application, the Xplorer framework introduces an extension for multiple applications. Furthermore, the efficiency of design suggestions has been enhanced by employing fuzzy reasoning instead of conventional reasoning as a new feature to compensate for imprecise information gathered during the status evaluation. In addition, the architecture estimator resembles an analytical component in the process, which complements the interactive approach.

Two other approaches, the design space exploration for Raw processors (see section 5.2.4) and the exploration for multimedia processors by Kin et al. (see section 5.2.6) have a similar base as the Xplorer approach, as the design space is also constructed from a combination of different units making up the target architecture to be designed. However, the general approaches are quite different, as the Raw exploration relies on analytical models for applications and design space, which can be difficult to generate for arbitrary applications. The system by Kin et al. employs simulation for reliable state evaluation, but uses a vast amount of benchmark scenarios. The Xplorer tries to combine the advantages of both approaches by performing the core exploration cycle on one application, which is selected from the application set in a suitable way. The exploration for the single application is done by using actual synthesis.

The use of fuzzy logic in design space exploration can also be found in the ICOS system (see section 5.2.5). ICOS uses fuzzy values for fixed assessments of design alternatives and relies on the existence of a complete set of these assessments. The Xplorer introduces the application of fuzzy reasoning for design space exploration, which, together with the interactive approach, allows expert knowledge to complement the user experience, thus resembling a flexible aid.

8.2 System Overview

In order to perform the general exploration approach described in section 8.1, an appropriate framework has been developed, making use of the MA-DPSS for the mapping and part of the analysis. Additional tools are added for further mapping analysis, generation of suggestions, and interactive control. The implementation of the approach discussed in section 8.1 will be discussed in the following. An overview of the KressArray Xplorer framework showing its components is pictured in figure 8-2.

The exploration process starts with the designer specifying a set of applications, which resembles the application domain, in the ALE-X language [Kres96], the input language for the MA-DPSS. In the preparation phase according to section 8.1, each application is compiled by the ALE-X compiler (see section 7.1)

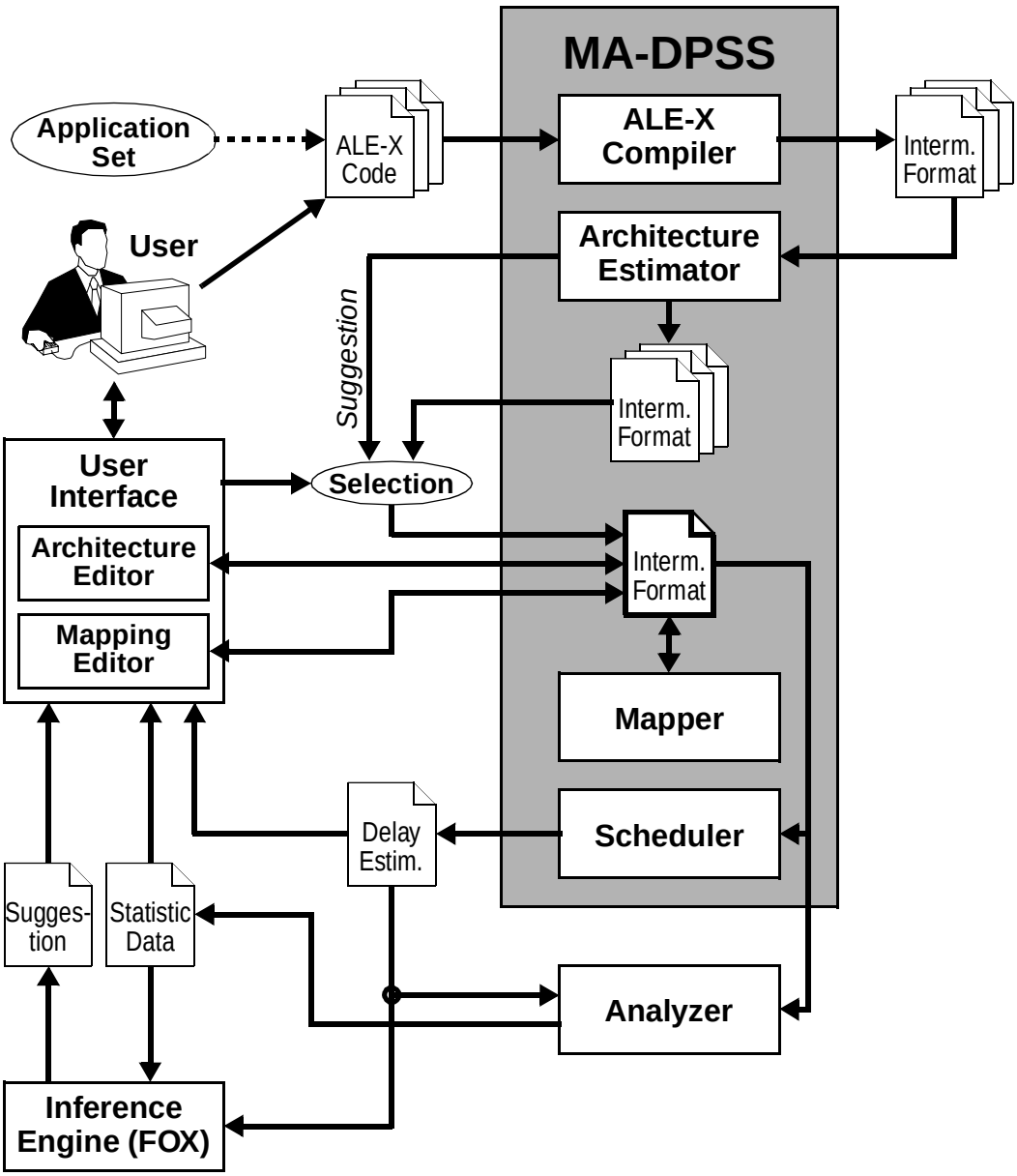


Figure 8-2: KressArray Xplorer system overview

into a first intermediate form, containing the datapath. Afterwards, the minimal architecture requirements for each application are determined by an architecture estimator (see section 7.2). These estimations are merged to gain the minimal requirements for all applications, determining also the application with the most requirements.

For the following exploration phase, the designer selects one application from the domain. He or she may base the selection on a suggestion from the estimator, or on data produced by an additional run of the mapper and the analyzer, if

necessary. The selected application is then used to iteratively refine the architecture. The datapath is synthesized on the current architecture by the mapper (see section 7.3). The mapper output is then analyzed, gathering different kinds of data, including a performance estimation of the datapath, which is generated by a scheduler (see section 7.4), and other statistics, like e.g. the usage of communication resources or the complexity of the datapath. Those data are generated by an analyzer (see below in section 8.3). Based on the extracted statistics, suggestions for the improvement of the architecture are produced based on inference rules. These rules are evaluated by invoking an extern inference engine [HLNL96], which produces suggestions for improving the architecture by ranking possible actions for architecture changes. The suggestions are presented to the designer via the interactive user interface, which contains also an architecture editor and a mapping editor.

The designer changes the architecture, either heeding the suggestions or using own plans. While changes to the number of the communication resources or the array capabilities can be made using the architecture editor, a change of the operator set is done by the designer in the datapath. This operation may change the complexity of the applications in the domain, making a new selection of the application for the exploration process necessary. In any case, the exploration phase is iterated until a suitable architecture has been found.

The final verification phase is realized by mapping the other applications of the domain onto the optimized architecture to make sure it satisfies their requirements also. In case this fails, the exploration process may be repeated with one of those applications turning out to perform bad. When a suitable architecture is finally found, the configuration data for all applications are available through the exploration approach. If necessary, the mapping of a single application can be tuned using the mapping editor incorporated in the user interface.

8.3 Mapping Analysis

In order to create suggestions for the improvement of the architecture, a base for the decisions leading to these suggestions has to be provided by gathering statistic data about the mapping of the exploration datapath onto the current architecture. For the collection of the statistics, a dedicated analyzer [Hung00], [HHHN00b] is employed. An overview on the analyzer subsystem for statistics retrieval is depicted in figure 8-3.

For the analyzer, a plug-in-based concept has been used to allow an easy extension of the subsystem capabilities by adding additional plug-ins. The statistics are kept in a central database, which is shared between all plug-ins

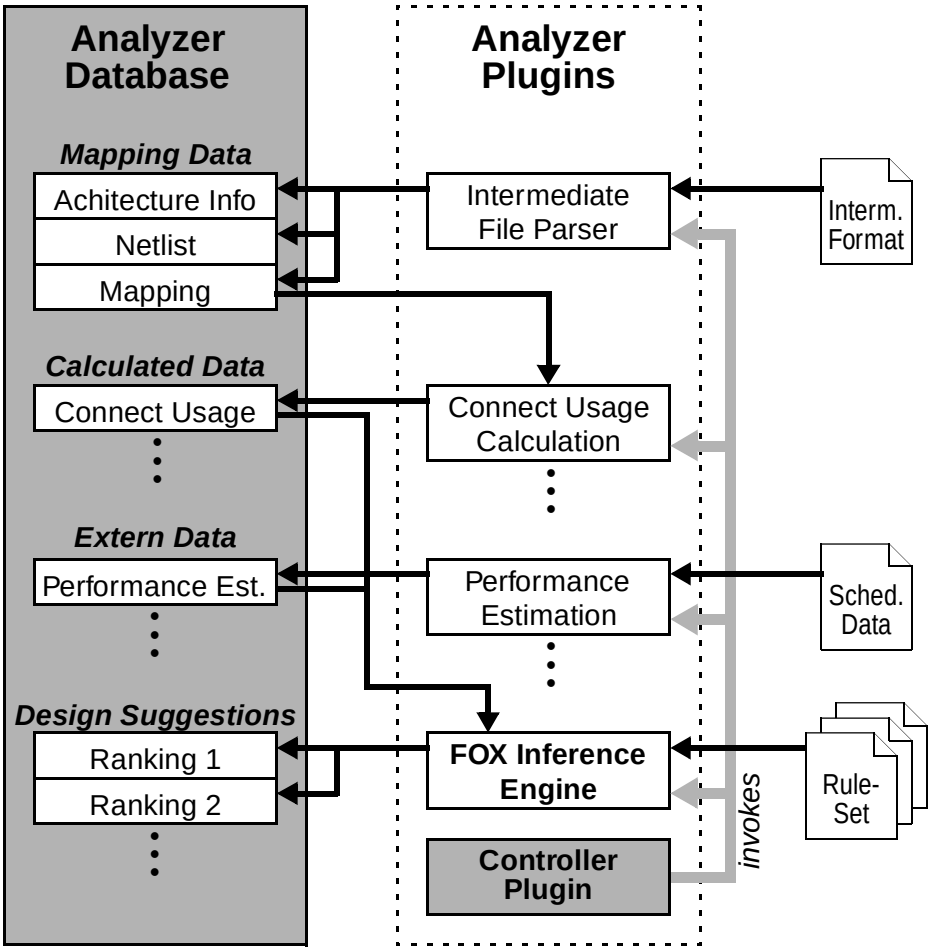


Figure 8-3: Xplorer analyzer subsystem overview

adding new statistics they produce. To do this, the plug-ins may also make use of existing data, by invoking other plug-ins. To avoid unnecessary invocations, all data are cached in the database. The process of data collection is steered by a controller plug-in, which is called first and invokes the other plug-ins to collect the data. There are three types of statistics, which are produced by the analyzer, namely mapping data, calculated data, and extern data.

- Mapping data are statistics directly contained in the intermediate file. These values are extracted by a parser plug-in and are typically used by other plug-ins to create more complex statistics.
- Calculated data is produced by a specific algorithmic step executed by the according plug-in. The calculation relies typically on data already made available in the database by other plug-ins.

- Extern data is produced by other tools like the MA-DPSS scheduler described in section 7.4. If necessary, the according plug-in calls the extern tool on operating system level and acts as an interface between the tool output and the analyzer database.

When all necessary statistics are collected, a reasoning process is invoked to generate design suggestions. This is done by activating an inference engine, which uses predefined rule sets to produce ranks for possible design actions like described in section 8.1. The inference engine is represented by the FOX fuzzy executor [NLLH95], which is integrated into the analyzer as a special plug-in, as shown in figure 8-3.

A collection of example measures produced by the analyzer is given in section A.3. While the calculation of most of those values is quite obvious or described elsewhere, the technique for the generation of the datapath complexity is described in more detail in the following subsection.

8.3.1 Datapath Complexity

A coarse measure for the complexity of a given datapath is known in literature as the so-called Rent coefficient, derived from a relation known Rent's rule [LaRu71]. The Xplorer analyzer subsystem provides a simplified calculation of the Rent coefficient, producing the according parameter for use in the reasoning process. In the following, the basics of Rent's rule are described, and two different methods of simplified complexity computation used in the Xplorer analyzer are presented.

8.3.1.1 Rent's Rule

The original relation known as Rent's Rule [LaRu71] applies to logic designs consisting of a certain number of blocks, which are connected to each other according to a netlist. The design is assumed to be partitioned, i.e. the blocks are grouped into several subdesigns, dubbed modules in the following. Let P denote the number of external pins of a given module, i.e. those connections, which cross module boundaries, B the number of blocks comprising a module, and T the average number of connections (terminals) of these blocks to other blocks. These definitions are illustrated in figure 8-4.

In the figure, it is assumed that the input ports (*a*, *b*, *c* and *d*) as well as the output ports (*x* and *y*) appear as kind of blocks, which add to the complexity of the datapath. This assumption suits a general KressArray architecture with edge ports for data I/O, as these ports appear as pseudo-rDPUs and are thus included in the routing process.

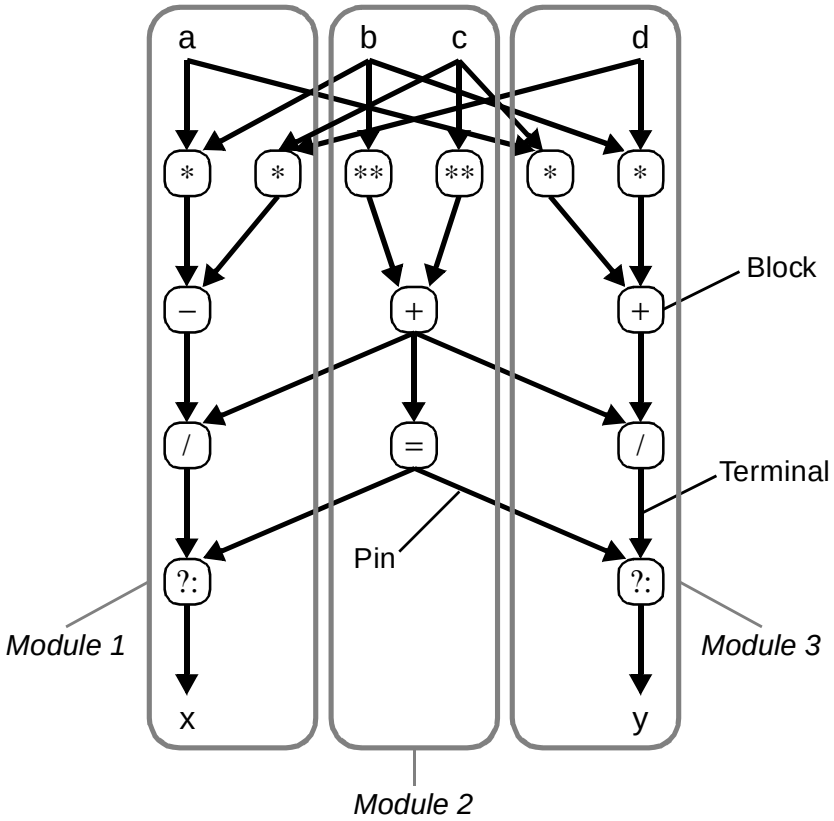


Figure 8-4: Example datapath showing Rent's rule terminology

Rent's rule describes a relation between the number of blocks in a module and the number of pins. The most common form of the rule is given by:

$$P = T \cdot B^{r}$$

The exponent r is known as the Rent coefficient. Its value depends on the complexity of the datapath. Typical values range from about 0.4 for regular designs up to about 0.8 for irregular structures. Obviously, the Rent coefficient relates directly to the requirements for routing resources of the specific datapath. It can be calculated by the following equation:

$$r = \frac{log\left(\frac{P}{T}\right)}{log(B)}$$

To illustrate this relation, the values for *P*, *T*, *B* and *r* are given in tabl e8-1 for the example datapath consisting of four modules shown in figure 8-4.

Module	P	B	T	r
Module 1	6	7	2.57	0.44
Module 2	8	6	3	0.55
Module 3	6	7	2.57	0.44
Average	r = 0.47			

Table 8-1: Rent coefficients for example datapath

According to the table, the average Rent coefficient for the datapath calculates to 0.51, which can be considered a moderate complexity.

8.3.1.2 Xplorer Complexity Computation from Mapping

The Rent coefficient is highly dependent on the specific partitioning of the datapath. As the three modules of the example in figure 8-4 are chosen to contain mostly connected blocks, the resulting coefficients are low to moderate. On the other hand, a bad partitioning of the blocks would lead to quite high values. In the literature, other efforts are described calculating the Rent coefficient by performing a partitioning on the blocks in advance [HMD79], [Stro96]. However, as the MA-DPSS provides a complete placement, a novel simplified calculation method using this placement is employed in the Xplorer framework.

The technique used is based on the idea, that the placement procedure aims to arrange connected rDPUs close to each other. Thus, a simple partitioning of the mapped datapath consists of dividing the mapping into rows and columns of rDPUs. For each row or column module, a Rent coefficient can be calculated using the equation given in section 8.3.1.1. The total datapath complexity is then calculated by the average of all these coefficients. This method adds a considerable amount of imprecision to the complexity calculation, as connected rDPUs, which are in proximity of each other, need not necessary be neighbors. However, the technique has shown to supply usable values for the fuzzy reasoning. The method for estimating the datapath complexity is illustrated in figure 8-5.

The figure shows a mapping of the datapath in figure 8-4. Although the resulting measure for the datapath complexity does not differ much from the one given in table 8-1, the simplified calculation used in the Xplorer framework can be expected to produce higher values than those calculated from an optimized

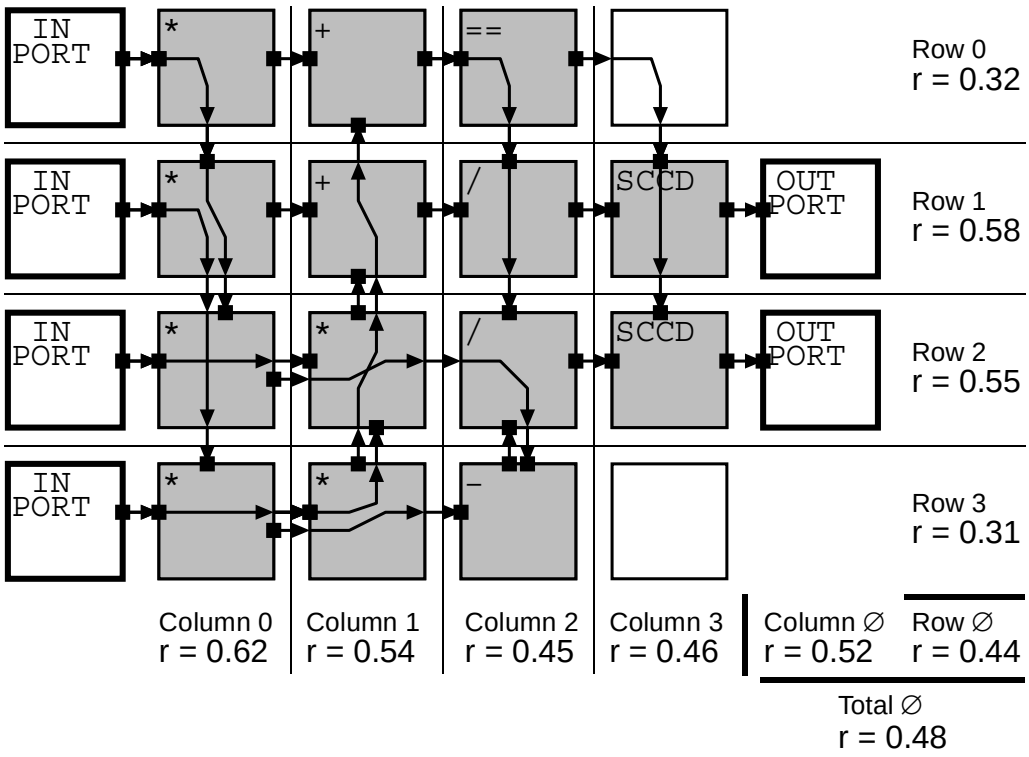


Figure 8-5: Example calculation of datapath complexity

partitioning. This is caused by the additional consideration of the routing connections passing between rows and columns. Also, the Xplorer method is more dependent on the quality of the mapping.

8.3.1.3 Xplorer Complexity Computation from Datapath

To overcome this strong dependency, another method is implemented in the Xplorer framework to get a complexity measure, which is completely independent of the mapping. This method calculates the Rent coefficient by simply assuming the whole datapath as one single partition. The pins are then the I/O connections of the datapath. The values gained by this method can be expected to be higher and more imprecise than the other two methods. However, as neither a partitioning nor a mapping is needed for this method, the complexity is useful for the application selection at the beginning of the exploration process (see section 8.1).

8.4 Generation of Suggestions

After statistic data has been derived from a mapping, this data has to be interpreted in order to derive a possible improvement. This interpretation requires expert knowledge. In order to make human knowledge about the evaluation of statistics available to users, an approach representing a simple expert system [PaMc90] has been used, which generates suggestions how to enhance the architecture. These suggestions are made by providing a set of possible actions to change the architecture, and a rating about the quality of each action considering the current architecture analysis. The resulting set of ratings are then presented to the user, so an appropriate action can be taken. This concept is illustrated in figure 8-6. In this section, the method of how these ratings are derived, will be discussed.

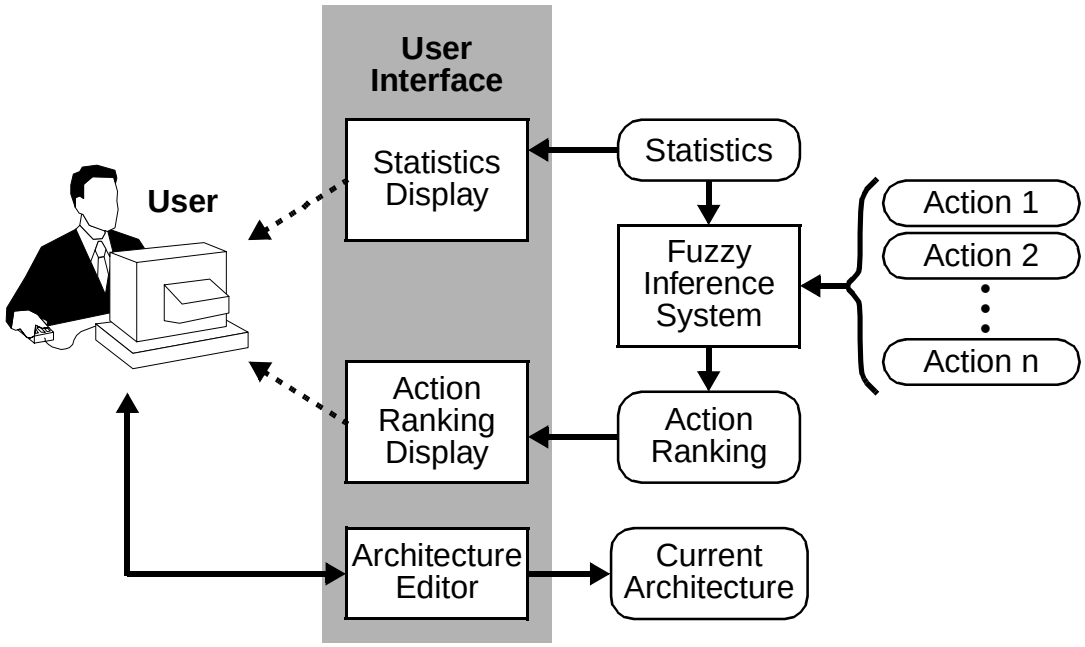


Figure 8-6: Generation of architecture improvement suggestions and interactive architecture enhancement

For the representation of knowledge, several methods exist and are discussed in the literature [HaKa92]. The Xplorer framework uses a fuzzy expert system approach [Kand92] employing rules for knowledge representation. This system is implemented using the extern fuzzy logic system FOOL / FOX developed elsewhere [HLNL96], [NLLH95]. As this topic is discussed extensively in the literature, the description of the concepts are restricted to those relevant for the Xplorer suggestion generation.

The general approach used to generate a rating for a possible architecture change is similar to a fuzzy controller [NLLH95], [MMSW93] using an inference engine. The fuzzy system used in the KressArray Xplorer is depicted in figu re8-7.

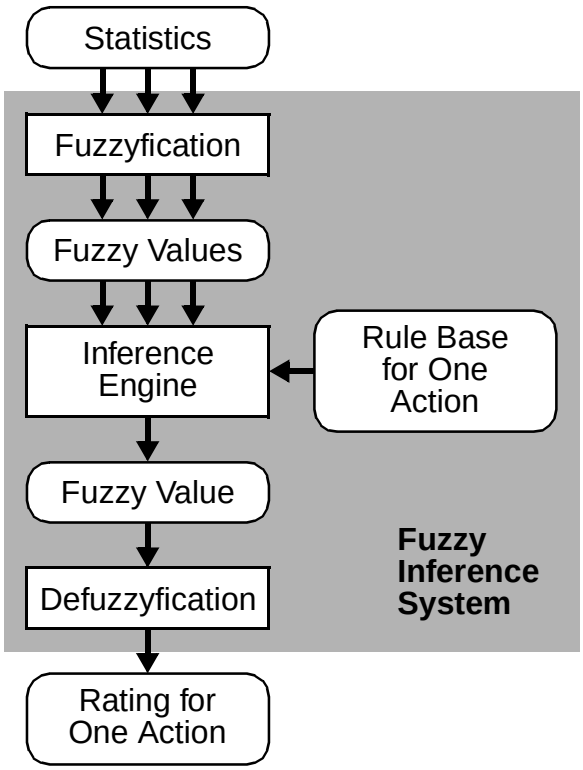


Figure 8-7: Fuzzy inference system used in the Xplorer framework

The basic task of this system is to generate a rating for one specific action at a time. This rating is based on statistics gathered by the analyzer (see section 8.3). The rating process has to be performed for each possible action sequentially.

The first phase of the process is a fuzzyfication of the crisp statistic data so they can be processed further. Based on the resulting fuzzy values, an inference engine evaluates the statistics and generates a rating for the appropriateness of one specific action. The expert knowledge, which is needed to do the rating based on the observed statistics is represented by a rule base for the action. The resulting rating is again a fuzzy value, which has to be transformed into crisp data in order to be processed further. This is done in a defuzzyfication step. Finally, the actions are sorted by their ratings, thus providing a suggestion for a suitable architecture change.

In the next subsection, the basics of fuzzy sets and linguistic variables are sketched. Then, common operations on fuzzy sets, as being used in the Xplorer rules, are reviewed. After that, the concept of approximate reasoning using fuzzy

rules is explained, followed by a brief description of the defuzzyfication process used to generate a crisp value from fuzzy sets. Finally, the FOOL / FOX toolkit used for implementation of the reasoning is introduced briefly.

8.4.1 Fuzzy Sets and Linguistic Variables

The Xplorer framework employs a technique called fuzzy reasoning to generate design suggestions, which is based on fuzzy logic. In the following, fuzzy sets and linguistic variables will be defined briefly. A more thorough discussion of these concepts can be found in [KPMK96].

The concept of fuzzy sets as introduced by L. Zadeh in #[Zade65] is a generalization of crisp sets done by extending the concept of membership to a set. Let X be a set of objects. Then a fuzzy set A over X is a set of ordered pairs defined by the following relation:

$$A = \{ (x, \mu_A(x)) \mid x \in X \wedge \mu_A(x) \rightarrow [0, 1] \}$$

In this context, $\mu_A(x)$ is the membership function, which defines, to which degree element x is a member to the fuzzy set A.

For the Xplorer expert system, a certain type of fuzzy sets called linguistic variables is employed, which allow a more natural and human readable definition of knowledge than crisp variables. A linguistic variable L features a set $T(L) = \{t_1, ... t_n\}$ of natural language terms t_i also called adjectives, which are fuzzy sets over a crisp variable U. A formal definition found in [Zade75a], [Zade75b] generalizes this concept by allowing compound adjectives, defining a linguistic variable as a five-tuple $(L, T(L), U, G, M)$. In addition to L, $T(L)$ and U, which are defined above, G is a syntactic rule to generate the $t_i \in T(L)$, and M is a semantic function assigning each $t_i \in T(L)$ a fuzzy set $M(t_i)$ over U. These concepts are illustrated by an example in figure 8-8.

Let "hc_usage" be a variable with values in the range [0..100], which is generated by a mapping analysis and indicates the usage of horizontal communication resources in a particular mapping. Using this variable, a linguistic variable $(L, T(L), U, G, M)$ can be constructed following the definition above. For the sake of simplicity, the syntax rule G is omitted. The name L for the linguistic variable is chosen as L = "Horizontal_NN_Usage", with U = [0..100] as the range of values for the crisp base variable. An example choice for $T(L) = \{t_1, t_2\}$ contains two adjectives t_1 = "low" and t_1 = "high", which can be used to describe the usage of the horizontal communication resources. To each of the adjectives, a fuzzy set over U is defined by M, resulting in $M(t_1) = \{ (u, \mu_{low}(u)) \mid u \in U \}$ and $M(t_1) = \{ (u, \mu_{high}(u)) \mid u \in U \}$. The membership functions μ_{low} in figure 8-8a and μ_{high} in figure 8-8b describe how much a given percentage of usage can be considered "low" or "high", by determining the degree of membership to the according fuzzy set. For example, a value of 50% could be interpreted as "rather

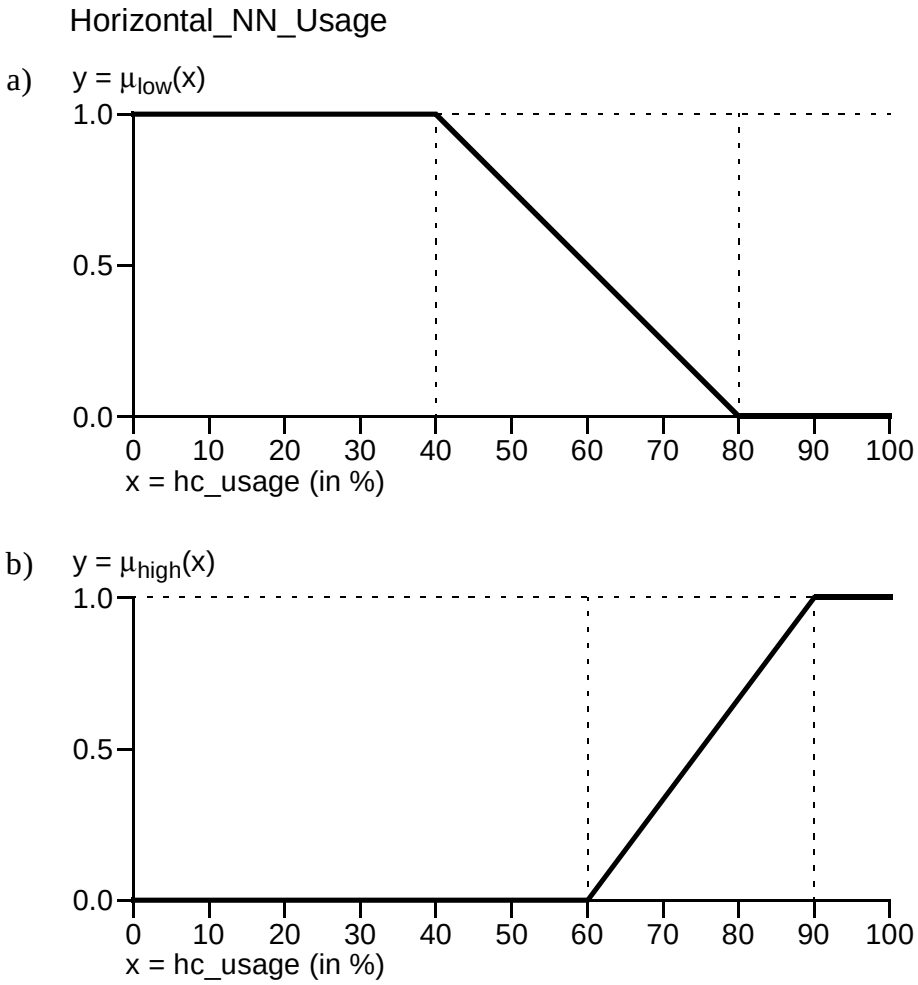


Figure 8-8: Example linguistic variable "Horizontal_NN_Usage" defined over the crisp value "hc_usage":
a) Membership function for adjective "low"
b) Membership function for adjective "high"

low", as $\mu_{low}(50\%) = 0.75$ and at the same time "not at all high", as $\mu_{high}(50\%) = 0$. In the example, the membership functions for the adjectives "low" and "high" can be expressed as follows:

$$\mu_{low}(x) = \begin{cases} 1.0 & 0 \le x \le 40 \\ \frac{80 - x}{40} & 40 \le x \le 80 \\ 0.0 & 80 \le x \le 100 \end{cases} \qquad \mu_{high}(x) = \begin{cases} 0.0 & 0 \le x \le 60 \\ \frac{x - 60}{30} & 60 \le x \le 90 \\ 1.0 & 90 \le x \le 100 \end{cases}$$

8.4.2 Operations on Fuzzy Sets

The union and intersection operations for crisp sets have equivalents for fuzzy sets. These operations play an important role for the reasoning, as union and intersection of fuzzy sets are equivalent to boolean OR and AND operations when applied to premises in rules. Accordingly, the terms OR and AND will be used as synonyms to "union" and "intersection" in the following discussion.

Generally spoken, an operation "•" on fuzzy sets *A*, *B* and *C* with $C = A \bullet B$ is defined by the according membership functions as $\mu_C(x) = f_\bullet(\mu_A(x), \mu_B(x))$. In this context, $f_\bullet(u,v)$ denotes a function specific to the operation "•", which defines the membership function for *C* from those of *A* and *B*.

While the intersection and union for crisp sets are defined in an unambiguous way, there are a large variety of corresponding operators for fuzzy sets [Zimm93], [MMSW93], [NLLH95]. While the intersection operation (AND) is typically implemented by functions known as t-norms, the union (OR) is realized by functions dubbed t-conorms. In addition to this, average operations exist, which implement a semantic lying between AND and OR. In the following, the concepts of these operations are sketched briefly and illustrated by an example function for each operation.

8.4.2.1 T-Norms and T-Conorms

Prior to the discussion of the semantics of t-norms and t-conorms, formal definitions are given for both, according to [Zimm93], [MMSW93], [NLLH95].

A function $t : [0, 1] \times [0, 1] \rightarrow [0, 1]$ is called a t-norm if all of the following conditions are met for $x,y,z,u,v \in [0, 1]$:

- $t(0, 0) = 0$; $t(x, 1) = t(1, x) = x$;
- $t(x, y) \leq t(u, v)$; if $x \leq u$ and $y \leq v$
- $t(x, y) = t(y, x)$;
- $t(x, t(y, z)) = t(t(x, y), z)$;

A function $s : [0, 1] \times [0, 1] \rightarrow [0, 1]$ is called a t-conorm if all of the following conditions are met for $x,y,z,u,v \in [0, 1]$:

- $s(1, 1) = 1$; $s(x, 0) = s(0, x) = x$;
- $s(x, y) \leq s(u, v)$; if $x \leq u$ and $y \leq v$
- $s(x, y) = s(y, x)$;
- $s(x, s(y, z)) = s(s(x, y), z)$;

It should be noted, that every t-norm $t(x, y)$ defines an according t-conorm $s(x, y)$ and vice-versa by the following equations:

- $s(x, y) = 1 - t(1 - x, 1 - y)$;

- $t(x, y) = 1 - s(1 - x, 1 - y)$;

There are different t-norms and t-conorms described in literature, which are used to implement the union and intersection of fuzzy sets, and thus AND and OR operations. For illustration of the semantics, the functions proposed by Zadeh [Zade65] are used, resembling the minimum operator as t-norm and the maximum operator as t-conorm:

$t_{min}(x, y) = \min(x, y)$;

$t_{max}(x, y) = \max(x, y)$;

In contrast to these functions, which resemble unparameterized operators, there are several parameterized t-norms and t-conorms [MMSW93], [Zimm93] of the form $t(p, x, y)$, where the parameter p affects the result of the according operator.

Let L and H be fuzzy sets over $X = [0..100]$ defined by the membership functions μ_{low} and μ_{high} as described in section 8.4.1 and depicted in figure 8-8a and figure 8-8b respectively. Then, using the minimum and maximum operators as t-norm and t-conorm, the intersection and union $L \cap H$ and $L \cup H$ are given by:

$L \cap H = \{ (x, \mu_{L \cap H}(x)) \mid x \in X \wedge \mu_{L \cap H}(x) = \min(\mu_{low}(x), \mu_{high}(x)) \}$;

$L \cup H = \{ (x, \mu_{L \cup H}(x)) \mid x \in X \wedge \mu_{L \cup H}(x) = \max(\mu_{low}(x), \mu_{high}(x)) \}$;

Following the example in section 8.4.1, the resulting fuzzy set $L \cap H$ can be interpreted as the values of communication resource usage, which are considered "low" AND "high", while $L \cup H$ represents the set of values which are "low" OR "high". The resulting membership functions are depicted in figure 8-9a and figure 8-9b respectively.

8.4.2.2 Average Operators

Average operators resemble a class of operators on fuzzy sets, which complement the possibilities given by the t-norms and t-conorms by the addition of a compensatory factor to operators. This compensation is sometimes useful to better model the semantic of a natural language description. For example, in the statement "If the nearest neighbor connection usage is high, and the number of global bus connections is high, then the number of neighbor connections should be increased.", the semantics of the "and" is not exactly covered by the AND implemented by a t-norm. Basically, both prepositions should be true in order to reach the conclusion. However, if e.g. the number of global bus connections is extremely high, then it might be advisable to increase the number of neighbor connections, even if their usage happens to be not too high. In this case, the extraordinary degree of one preposition compensates for the lower degree of the second one. To implement this compensation, a number of average operators have been defined in literature [MMSW93], [Zimm93]. Like t-norms and t-conorms, average operators can be defined in a formal way.

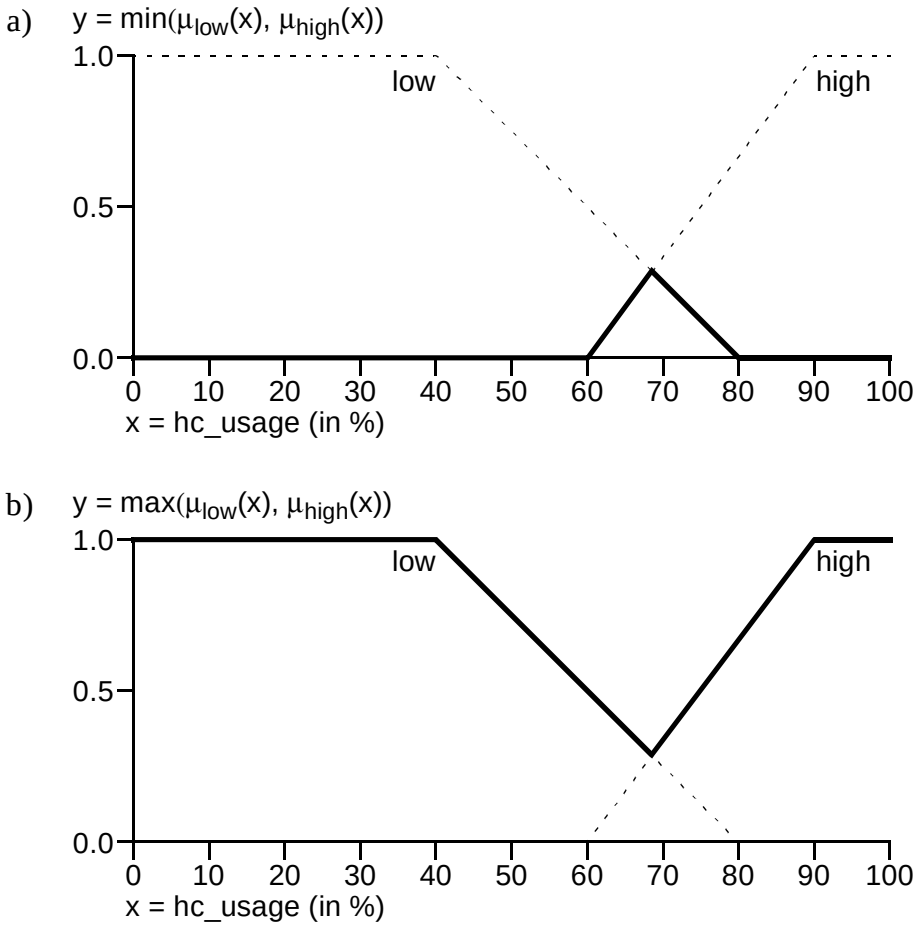


Figure 8-9: Operations on fuzzy sets:
a) Intersection
b) Union

A function $m : [0, 1] \times [0, 1] \rightarrow [0, 1]$ is called an average operator, if all of the following conditions are met for $x,y,z,u,v \in [0, 1]$:

- $\min(x, y) \leq m(x, y) \leq \max(x, y)$; $m \notin \{\min, \max\}$
- $m(x, y) = m(y, x)$;
- $t(0, 0) = 0$; $t(1, 1) = 1$;
- $m(x, y) \leq m(u, v)$; if $x \leq u$ and $y \leq v$
- m is continuous
- $m(x, x) = x$;

The values of the average operators lie between those of the t-norms and t-conorms. Examples for functions qualifying as average operators are the harmonic-mean m_{HM}, the arithmetic-mean m_{AM}, and the dual-of-harmonic-mean m_{DHM}. They are given by:

$$m_{HM}(x, y) = \frac{2xy}{x + y}$$

$$m_{AM}(x, y) = \frac{x + y}{2}$$

$$m_{DHM}(x, y) = \frac{x + y - 2xy}{2 - x - y}$$

The harmonic-mean can be used to implement an AND, while the dual-of-harmonic-mean is appropriate for an OR. The semantic of the arithmetic-mean lies in-between these of AND and OR. These three operators are illustrated by an example application to the μ_{low} and μ_{high} fuzzy sets described in section 8.4.1. The fuzzy sets resulting from the operations are depicted in figure 8-10.

8.4.3 Approximate Reasoning

Based on the foundations given in the previous sections, the expert knowledge to decide, if a specific action is reasonable with a given set of analyzer data, can be expressed in a set of rules. Let there be m rules R_1, ...,R_m to represent the knowledge about the action, and let R_j, j ∈ {1, ..., m) be one of those rules. Let further d_{j1}, ..., d_{jn} denote a set of n statistic data values, which describe the degree of matching for an according set of architecture properties P_{j1}, ..., P_{jn}. Let r_A denote the resulting ranking for the action. Then R_j has the following form:

IF $d_{j1} = P_{j1}$ **&** $d_{j2} = P_{j2}$ **&** ... **&** $d_{jn} = P_{jn}$ **THEN** $r_A = A_j$ **WITH** c_j ;

In this form, the ranking adjective A_j denotes the output value, which should be assigned if the preconditions hold true. Furthermore, the value c_j expresses a degree, in how important this rule is for the generation of the final output value from all rules. While A_j is an adjective of the output value, and thus a fuzzy set, c_j is supposed to be a number in the interval [0, 1].

With this representation, r_{jA} can be determined using a generalized form of the modus ponens [KPMK96]. In contrast to the reasoning found in non-fuzzy expert systems [PaMc90], this generalized modus ponens allows for approximate reasoning [KPMK96], [Kand92]. Approximate reasoning is the process of inferring imprecise conclusions from imprecise premises, thus being appropriate for the aim of generating design suggestions rather than directives.

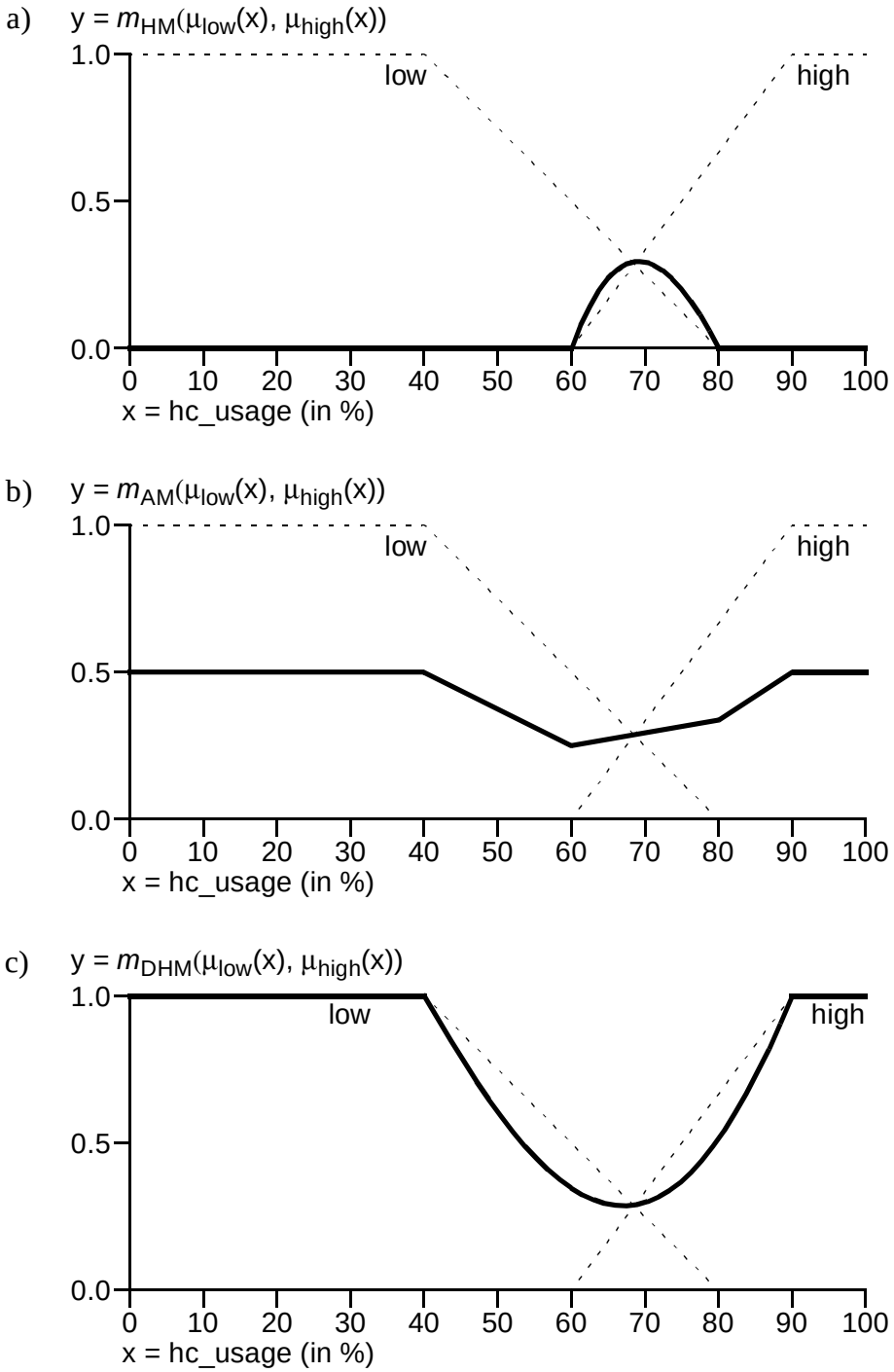


Figure 8-10: Fuzzy average operators:
a) Harmonic-mean
b) Arithmetic-mean
c) Dual-of-harmonic-mean

The use of fuzzy values and sets in the rules further implies several differences to non-fuzzy reasoning due to additional degrees of flexibility. In the following, the inference process will be explained in more detail. The steps to be performed in this process are compatibility computation, aggregation of compatibilities, evaluation of certainty, inference, and accumulation of all rule outputs into one final fuzzy value representing the ranking for the action. For the Xplorer suggestion process as depicted in figure 8-7, this fuzzy value has to be converted into a numeric value by a defuzzyfication process. This process is described separately in section 8.4.4. The other steps of the approximate reasoning will be described briefly in the following.

8.4.3.1 Compatibility Computation

The compatibility computation determines for each clause "$d_{ji} = P_{ji}$", to what degree it is fulfilled. The degree of fulfillment is dubbed a_{ji} . While in non-fuzzy logic this would be a binary decision, the output of this step here is a numeric value in the interval [0, 1]. Depending on the type of the data d_{ji}, there are different techniques to determine the fulfillment a_{ji}. As the Xplorer framework uses crisp values for the d_{ji}, the according a_{ji} is simply the membership function of P_{ji} at the point d_{ji}. Thus, the a_{ji} are given by:

$a_{ji} = \mu_{Pji}(d_{ji})$, for $i = 1, ..., n$;

8.4.3.2 Aggregation of Compatibilities

As the precondition of the rule may consist of several clauses "$d_{ji} = P_{ji}$" connected by operators ("&"), the degree of fulfillment of the total precondition is calculated by combining the a_{ji} to a single value a_j. The combination is done by applying fuzzy operations like those discussed in section 8.4.2 to the a_{ji}, where the selection of the operation depends on the semantic to be expressed. If the operation for rule R_j is denoted as $aggOp_j$, the degree of fulfillment is given by:

$a_j = aggOp_j(a_{j1}, ..., a_{jn})$;

8.4.3.3 Evaluation of Certainty

A further degree of flexibility in contrast to non-fuzzy reasoning is provided by the possibility to assign a different level of trust to each rule R_j, denoted as the certainty c_j. The certainty affects the level of fulfillment a_j by being applied to it using an appropriate operation $cerOP_j$. The general idea is to ensure, that the fulfillment of the precondition of the rule a_j should not exceed the certainty. Thus, a t-norm (see section 8.4.2.1) is used for $cerOP_j$. The resulting final degree of fulfillment a_j* of the precondition calculates to:

$a_j^* = cerOp(a_j, c_j)$;

8.4.3.4 Inference

The classic inference step produces the conclusion, if the precondition of the rule is fulfilled. This is once more a binary decision for non-fuzzy inference. In fuzzy reasoning, the conclusion A_j is fulfilled to a certain degree, which depends on the fulfillment of the precondition a_j*. It seems appropriate, that the fulfillment of A_j should not exceed the fulfillment of a_j*. Thus, the resulting conclusion A_j* is produced by combining A_j and a_j* using an inference operator *infOp*, which should be a t-norm. Thus, the final conclusion A_j* calculates to:

$A_j^* = infOp(a_j^*, A_j)$;

The result of this operation consists in limiting the fuzzy set A_j to the degree given by a_j*. This effect is depicted by an example shown in figure 8-11, showing the resulting A_j* for a conclusion A_j = "low" as used in section 8.4.1, a_j* = 0.7, and $infOp(x, y) = \min(x, y)$.

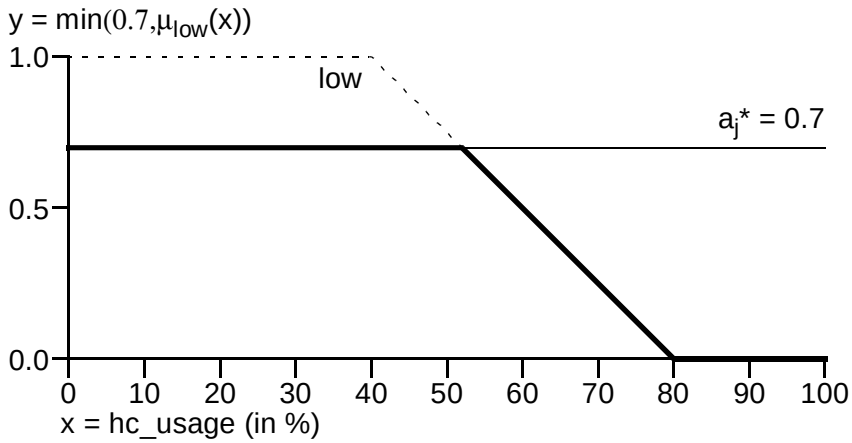


Figure 8-11: Example showing effect of inference operation, with variable "low" limited by a_j*=0.7, resulting in a new fuzzy set

8.4.3.5 Accumulation

To generate the final ranking r_A from the evaluated ruleset, the conclusions A_j* from all rules have to be combined again by an operator *accOp* to be applied to all A_j*. For a set of rules, the natural understanding is, that every rule represents by itself a satisfying condition for the calculation of the ranking. Therefore, for the combination of the conclusion, a t-conorm is used for *accOp*. Thus, for a ruleset of m rules, r_A is calculated by:

$r_A = \text{accOp}(A_1^*, ..., A_m^*)$;

For the requirements of the Xplorer framework, the resulting fuzzy set r_A is finally transformed back to a crisp value using a defuzzyfication step.

8.4.4 Defuzzyfication

The output of the approximate reasoning implemented by the inference engine is once more a fuzzy value. For the generation of a required crisp ranking from a fuzzy value, a defuzzyfication step is needed. If A is a fuzzy set over a set X of objects, then the aim of the defuzzyfication consists in producing the crisp value x where the membership $\mu_A(x)$ is maximal. To produce a unique crisp value, different methods exist [NLLH95]. One of the most popular ones is the method of the center of gravity. This method calculates the crisp value x' from a fuzzy set A as follows:

$$x' = \frac{\int_{x_{min}}^{x_{max}} x \cdot \mu_A(x)\, dx}{\int_{x_{min}}^{x_{max}} \mu_A(x)\, dx}$$

For the example fuzzy set "low" depicted in figure 8-8a, the value x' calculated by the center of gravity method results to $x' \approx 31.1$ %.

8.4.5 The FOOL / FOX Toolkit

To implement the approximate reasoning described in the previous sections, the FOOL / FOX toolkit [HLNL96], [NLLH95] is employed. This toolkit consists of the fuzzy inference engine FOX and a graphic user interface FOOL. A user can comfortably design a rule set with FOOL, which is then executed by FOX. A screenshot of the graphical user interface FOOL is shown in figure 8-12 for illustration. The picture shows the editor for adjectives of linguistic variables. A detailed description of this interface is given in [NLLH95]. In the Xplorer environment, the FOX engine is integrated into the framework as a plug-in for the analyzer as described in section8.3.

8.5 The User Interface

The Xplorer framework provides a graphical user interface [Wern00], which serves two main tasks. First, it allows a comfortable control of the exploration flow described in section 8.1 above, as well as a proper project management. Second, it implements an architecture editor, which allows the change of the architecture during the exploration, and a mapping editor for fine-tuning a mapping result. The user interface is based on the XMDS framework [HHHN99b], [HHNS99], which implements a client-server approach for the user interface. In the following, the XMDS framework will be reviewed briefly. Then, the concepts of features of the three main parts of the user interface, project

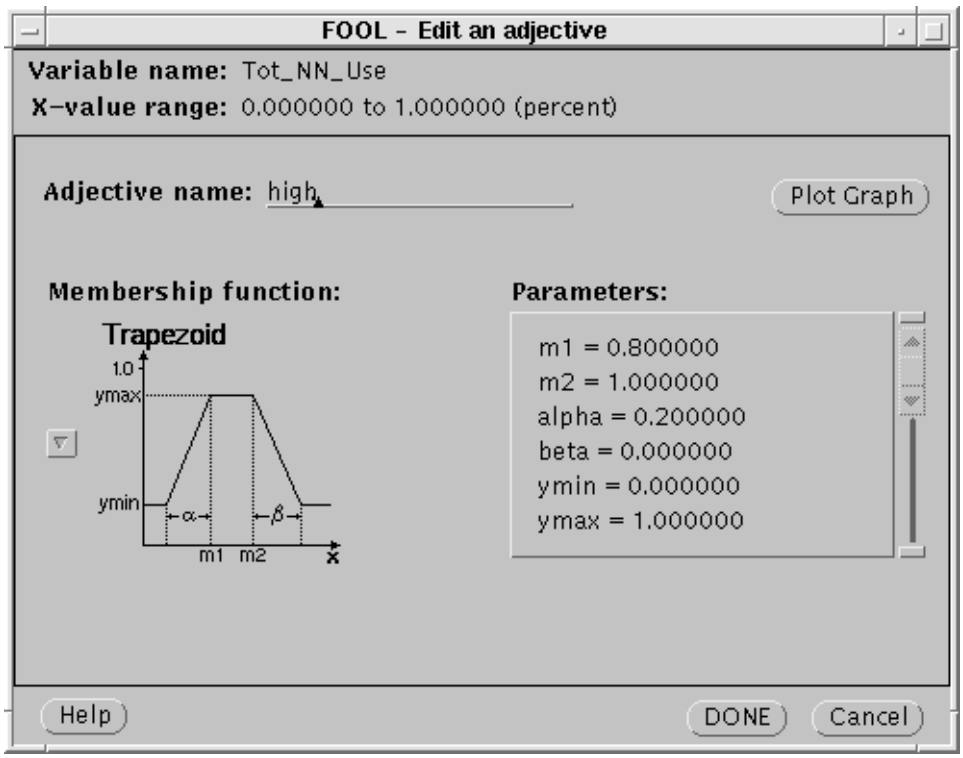


Figure 8-12: FOOL graphical rule designer
(adjective editor for linguistic variables)

management, architecture editor, and mapping editor, are outlined, accompanied by screenshots of selected elements of the user interface. For a detailed description of all features, the reader is referred to [Wern00].

8.5.1 The XMDS Framework

The XMDS framework [HHNS99] is a client-server based development system. The tools are running on the server, which keeps also the user data on according user accounts. On the client side, the user controls the development process by a user interface applet running on a Java-enabled web-browser. The user interface is downloaded from the server upon user login, so there is no further software required on the user side. The communication between client and server is done using standard web services. The Xplorer tools are invoked on the server side by a web server, which is enhanced by XMDS Java servlets to implement the transport services. These servlets are also responsible for invoking the appropriate Xplorer tools. An overview on the XMDS concept adapted to the Xplorer framework is given in figure 8-13.

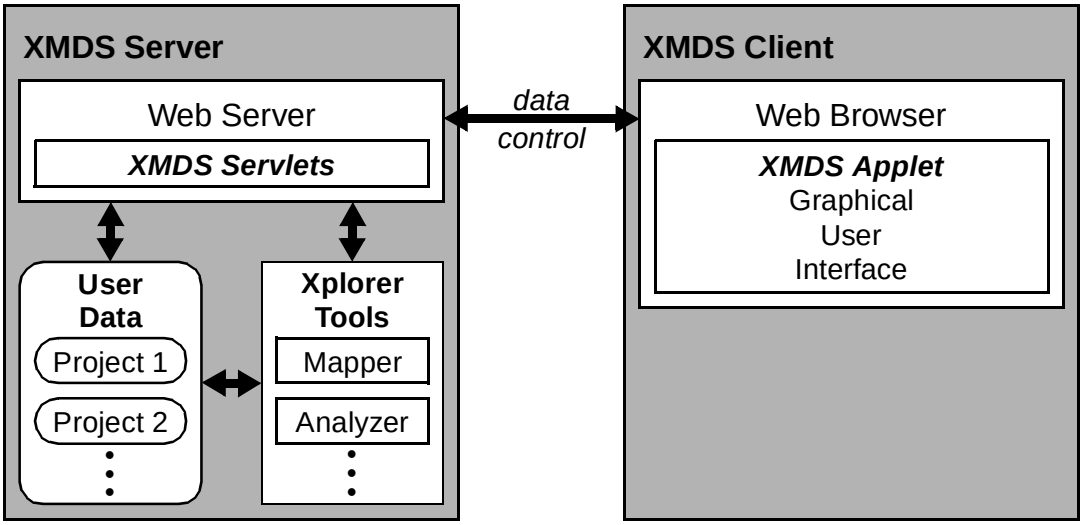


Figure 8-13: Overview on the XMDS client-server system adapted for the Xplorer framework

8.5.2 Project Management and Exploration Control

The user interface provides a convenient means to organize an exploration project. The user can establish an arbitrary number of projects, each resembling a folder on the user account. This project folder contains the applications in the domain being subject to the exploration. For each single application, a number of views is kept, with each view resembling a stage of the design process. Thus, there is a view for the ALE-X code, one for the α-, β-, and γ-intermediate formats each, and one for the hardware files determining the operator set.

Each view may have several instances representing different versions. Thus, the different architectures generated in the exploration process resemble versions of β- and γ-intermediate file views. For the setup of the project and retrieval of design data, the user can upload or download data for all views at any point.

An additional folder automatically generated by the system contains the rule sets available for the design suggestion generation. In the same way as different applications can be managed, different rule sets can be kept for different design objectives. A screenshot showing a typical Xplorer user interface scenario is shown in figure 8-14.

The figure shows an example project named "Example" with two applications "Datapath 1" and "Datapath 2" in the application domain. For "Datapath 1", all views as well as all versions for the γ-format view are expanded, showing five versions for this view. As can be seen in the figure, the exploration process itself is supported by according popup menus for each view, which allow to perform actions appropriate for this kind of data. Some actions apply also for whole

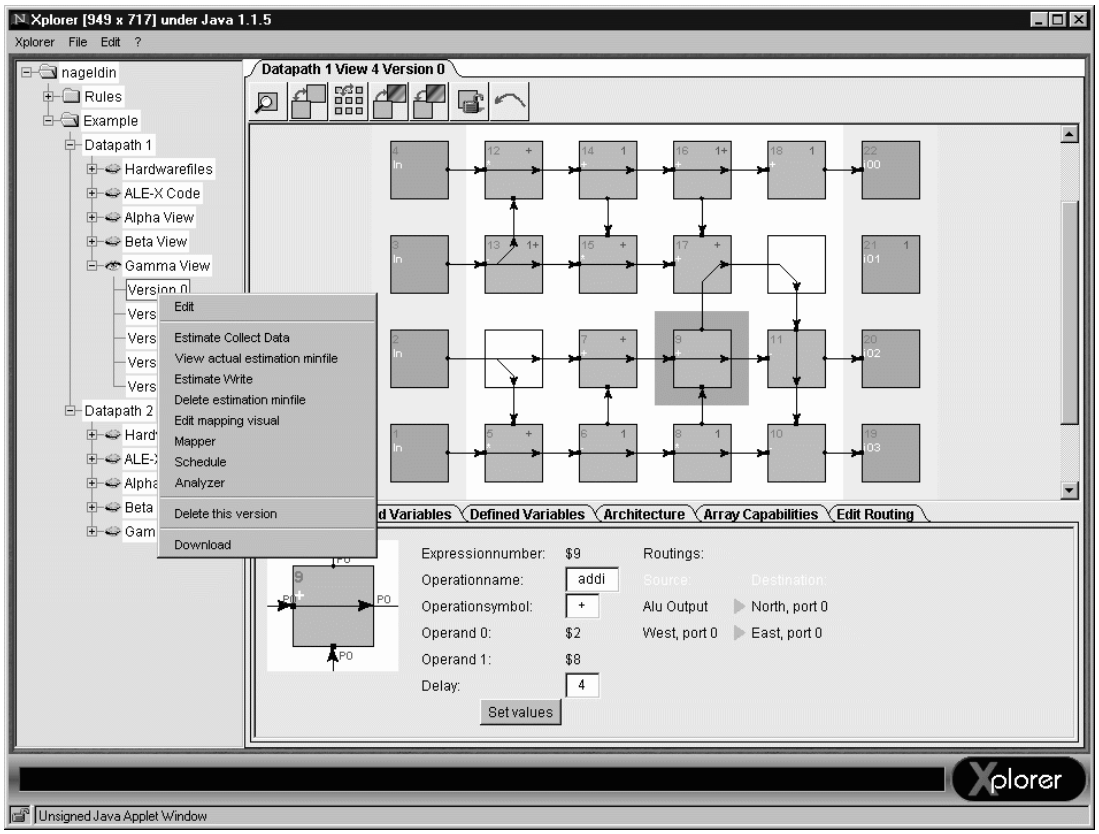


Figure 8-14: Typical XMDS Xplorer user interface scenario

project directories, e.g. the architecture estimation in the preparation phase. In the example scenario in figure 8-14, the menu applies to version 0 of the γ-format view, including actions like invocation of the analyzer or remapping. On the right window pane, the visual mapping editor is shown, along with the info window presenting detailed information about single rDPUs.

When a tool is invoked, output is presented on the browser window or on a dedicated window in the right pane of the user interface. An example application showing its results in the browser window is the analyzer, while e.g. the logs and the output of the ALE-X compiler and the mapper are presented in a window of the user interface. The output into the browser allows the use of HTML code for formatting of information. To illustrate the this feature, an example for the visualized results of the analyzer design suggestions is given in figure 8-15.

As can be seen, the ranks for the design actions (in the figure, a sample range of six actions is used) are presented in numeric format as well as graphically. Being sorted with the best rank being on top, the user can easily identify the action for enhancing the architecture, which the system considers the best one.

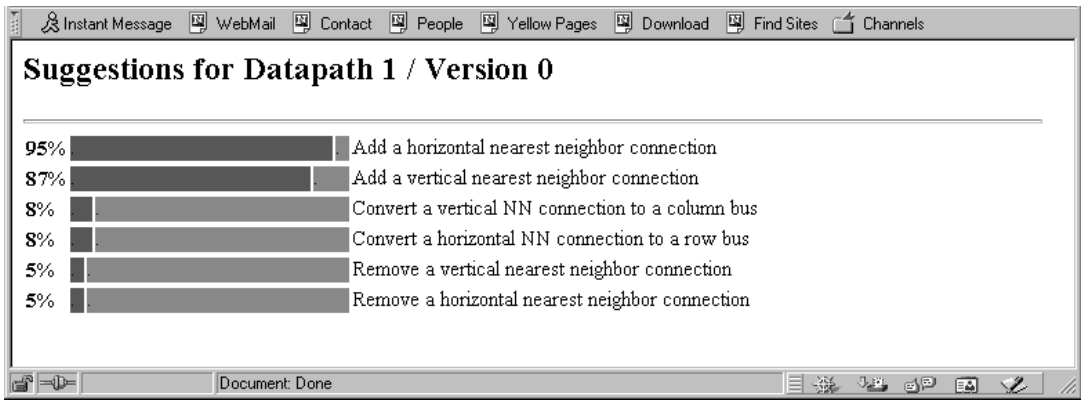


Figure 8-15: Example design suggestions shown in a browser window

8.5.3 Architecture Editor

The architecture editor incorporated in the user interface plays a central role in the exploration process, as it allows to change the current architecture. Possible actions include both editing of the number and properties of the communication resources as well as the editing of annealing parameters for the mapping process. For illustration, the architecture editor panel allowing the manipulation of communication resources is shown in figur e8-16.

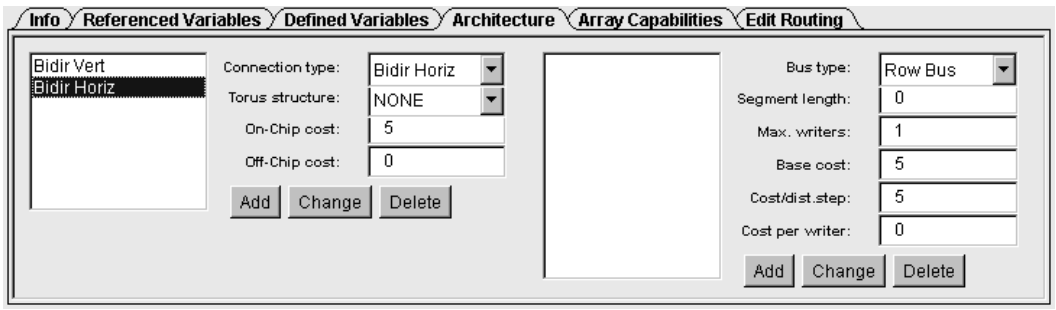


Figure 8-16: Communication resource editor panel of the Xplorer user interface

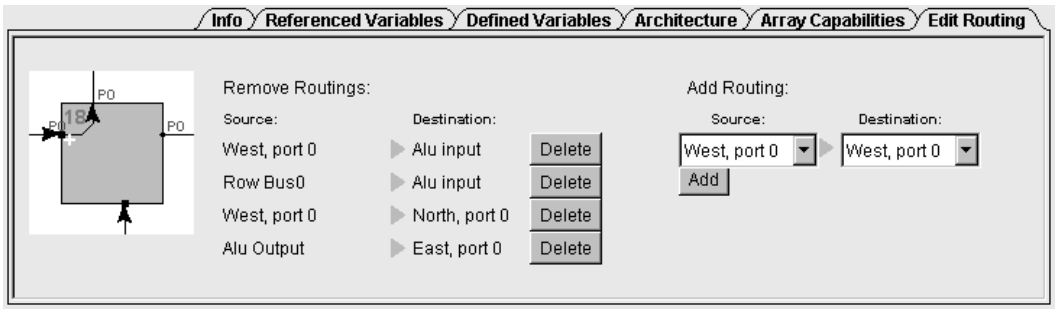
The figure shows the panel for an example architecture featuring just one horizontal and vertical bidirectional nearest neighbor link per rDPU. The horizontal connection is currently selected, allowing to change the type, torus parameters and cost factors for this link. The connection could also be deleted, or a new one could be added to the architecture. The same principle applies to the editing of the row and column backbuses in the right side of the panel.

8.5.4 Mapping Editor

The mapping editor allows the manipulation of the placement and routing information of γ-intermediate format files. The features of this editor include the direct editing of routing connections, swapping of single operator rDPUs or rDPU blocks, changing of I/O port properties, and setting of rDPU fixings.

The swapping of operators and the editing of the routing are mostly used for fine-tuning the mapping result. The port-setting feature is important at the beginning of the exploration process. The editing of rDPU fixings is also a means for improving and fine-tuning of the mapping. Two selected features of the mapping editor, the routing editor panel and the input port editor panel are illustrated in figure 8-17.

a)



b)

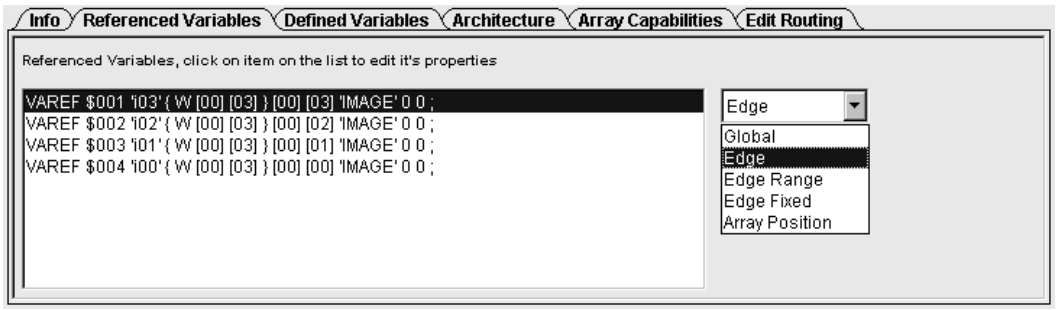


Figure 8-17: Elements of the Xplorer mapping editor:
a) Routing editor panel
b) Input port editor panel

The routing editor panel allows a quite low-level manipulation of the mapping result. As can be seen, single routing connections can be broken up or new connections can be added. This feature is meant to be used by the experienced designer, as it has the potential to alter the datapath and thus the semantics of the mapping. However, the user interface remembers connections which were broken up, and notifies the user about dangling links.

The input port editor panel is one of the two port manipulation features, the pendant being the output port editor panel, which looks quite similar. This editor plays in important role in the exploration process, as the ALE-X compiler generates by default global-bus ports. Thus, the port editor can be used to set the type of the I/O ports according to the design space described in section 6.4.3. Depending on the selected type, additional parameters can also be set, like e.g. the range and the side of the array for edge ports.