Journal of Universal Computer Science, vol. 6, no. 3 (2000), 324-344
submitted: 15/9/99, accepted: 25/11/99, appeared: 28/3/00 © Springer Pub. Co.

Coffein: Construction and Presentation of Design Knowledge

Stefanie Thies
(University of Paderborn, Germany,
thies@hni.uni-paderborn.de)

Abstract: Design is a hard problem: ill defined and open-ended. Schön [Schön (83)] characterized the process of designing an artifact as a successive refinement of reflection and redesign. Critiquing - the communication of reasoned opinion about an artifact - plays a central role in the design process. A computational critiquing mechanism provides an effective form of human-computer interaction to support these important aspects of design [Fischer 91]. Systems which realize such a computational critiquing mechanism are called Critiquing Systems. These systems provide context sensitive advice and rationale for an artifact designed by a user. This is realized by delivering so-called critiques, which contain relevant information for the user to the task at hand and are some kind of rule of thumb. But design experts are not programmers and programmers are not designers. So we need a module which supports design experts in stating their knowledge in form of critiques. The basis for this module is a a visual critiquing language (here called visual CiLa), completed by a knowledge construction supporting component. Furthermore a single design expert normally does not have all existing design knowledge. So the necessary information for building a complete design system is distributed among different stakeholders. Therefore we additionally need concepts and algorithms to combine and structure the critiquing knowledge of different design experts to construct a trustful, consistent and wise codesigner. This aspect is done by a module constructing the knowledge base and a module for constructing the virtual codesigner.

These two aspects - design knowledge construction and presentation - are realized in a tool called Coffein. This article deals with the way Coffein works and how it influences the design process.

Key Words: visual language, knowledge construction, multi-expert system, scientific visualization, knowledge based sysem, life-long-learning, infotmation systems.

Category: Multimedia Information Systems (H.5.1)

1 The Concept of Coffein

Design is a hard problem: ill defined and open-ended. Schön [Schön (83)] characterized the process of designing an artifact as a successive refinement of reflection and redesign. Critiquing Systems are successfully applied in design tasks as shown with the systems HERMES [Stahl 93], JANUS [Fischer 90], VDDE [Harstad 93] and IDIAS [Gutkauf 97]. Computational critiquing is the communication of a reasoned artifact by so-called critiquing systems. So these systems support the necessary underlying action-reflection-cycle of design: a user designs an artifact, the critiquing system analyses the artifact and delivers so-called critiques (this cycle is presented in [Fig. 1]).

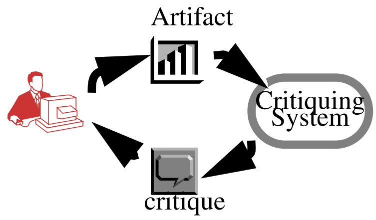


Figure 1: Structure of a Critiquing System

A critique contains three parts of information [see also Tab. 1]:

1. For which kind of artifact is the critique valid; e.g. how to analyse the artifact. This information is called *metric* and correlates to the if-condition of a rule.
2. The second information is what a user can do to solve the problem. This is called *advice* and correlates to the then-part of the rule.
3. Third a critique contains information why there is a problem with the artifact. This is called *reason*.

Structure of Critique	**If** ***(metric)***	**Then** ***(advice)***	**Because** ***(reason)***
Leading Question	In which situation is the critique valid?	What can you do to solve the problem?	Why should you thing about another solution, why is there a problem?
Example	**If** two colors of pie slices in a pie chart are not distinguishable	**Then** change one of the colors, to make them distinguishable	**Because** the correlation between legend and pie slices is harder if two colors are very similar.
Formalism	first order logic	text, script-language	text

Table 1: Specification of the Three Parts of a Critique

Critiques are knowledge carrying pieces and build the knowledge base of critiquing systems. These systems are easy to extend and to update by adding and modifying critiques step by step. But designers - who are able to formulate design critiques - are not programmers‘. Therefore we need a module for knowledge acquisition and construction [see right part of Fig. 2]. Another problem is that collecting domain knowledge from different domain experts can lead to inconsistencies resulting from the different point of views as well as to an information overload through information redundancy and overlaps. So we have to structure knowledge from different point of views; this can be done in the knowledge base. Last but not least, these structure (as well as the design knowledge) must be presented to the user by providing the metaphor of a virtual codesigner. This means a virtual person who wants to adapt to our perspective and who wants to give us consistent information and advice about a given designed artifact.

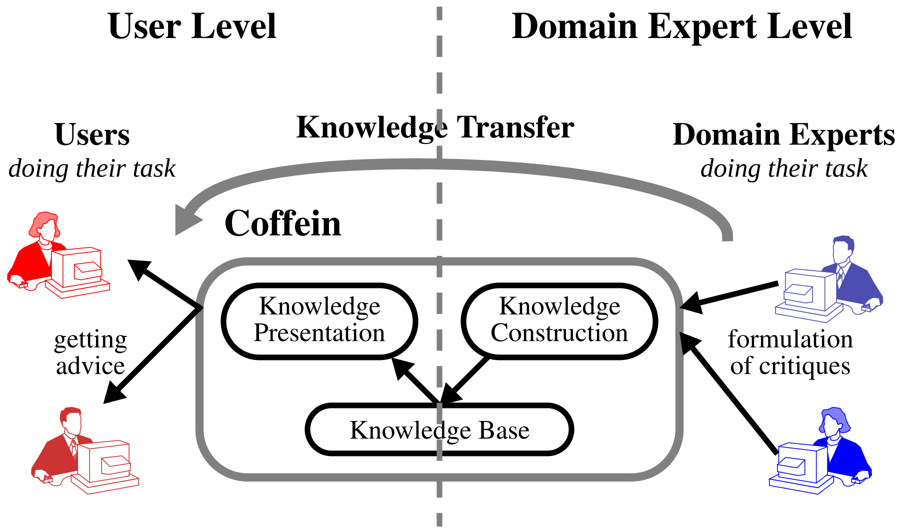


Figure 2: Concept of Coffein as Knowledge Transfer Medium

The following information is structured as follows. First (in Section 2) we explain the knowledge construction module, focusing on the basis, the visual critiquing language. Then (in Section 3) we show what inconsistencies can occur if different experts modify the knowledge base and how we can handle this situation. In Section 4 we show the method to construct a virtual codesigner. Specially we want to show how Coffein influences the design process on both sides: (a) influence for a user designing an artifact and (b) for domain experts formulating their knowledge. This article ends with a summary and an outlook.

2 The Knowledge Construction Module

Supporting domain experts in stating their knowledge is a difficult task, especially, if we want to have more than text (which is hard to analyse by a computer). This problem is well known in the context of knowledge based systems and this also applies to our problem of programming a metric. Different techniques of knowledge acquisition and construction have been explored in the last years. In principle three kinds of knowledge acquisition and construction by learning are distinguished [Knopik 91]: (1) rote learning, (2) learning by being told, and (3) machine learning. These classes are ordered by the load to the user: rote learning puts all the load on the user, but the user is aware of what s/he is doing and this technique works quickly. Prolog, Lisp, Agentsheets and CLIPS are examples of this technique. Learning by being told is similar to the technique of programming by demonstration [Cypher (93)]. The problem in the context of design is that changing the chart to a good chart is cannot be learned by simply looking at some examples.

The third technique of machine learning has no impact on the user. But a lot of examples are needed. If we assume that a designer works 2 hours on a design and that there are at least 129 different kinds of design, then we need at least 43 positive and negative examples (we assume, that we need at least 33% examples to learn the complete tree with ID3 as mentioned in [Görz (95)]); e.g. 86 h of work for every designer.

So we selected rote learning as the basic technique for constructing the metric. This technique is provided by visual CiLa, a visual critiquing language. The other two parts of the critique – the advice and the reason – are primarily text, which is quite easy to write. So we only have some simple supporting components for this. There is also a supporting component for investigating given abstracta and advice statements. We also need the specification of the connection between the design tool and the critiquing system, which is specified by the artifacts‘ hierarchy.

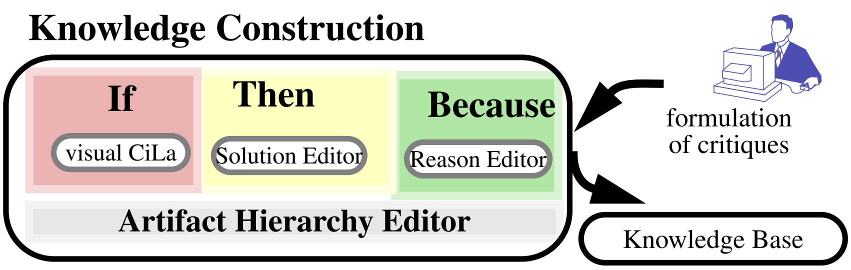


Figure 3: Structure of the Knowledge Construction Module

So we get the structure of the knowledge construction module as displayed in [see Fig. 3]. The included supporting components - the artifact hierarchy editor, visual CiLa, the solution editor, the reason editor and the inspection support - are explained in the following sections. Please notice that after describing the artifact hierarchy as the basis for all submodules, we put emphasize on the critiquing language for specifying the metric, because this is the hardest task for the user. We further assume that a designer creates a artifact, sees a problem with the artifact and starts to formulate the referring critique. This initiating artifact will be seen as the first negative example for the critique. Negative means that the critique fires; e.g. the system detects that the artifact has a shortcoming.

2.1 The Hierarchy of the Artifact

Basically the artifact hierarchy editor specifies the connection between the critiquing system and the design tool. But further on it was shown [Thies 95] that defining a hierarchy of the artifact eases the process of formulating a critique. Furthermore it lowers the burden of finding names of objects, describing parts of the artifact. Liebermann [Liebermann 89] points out that free defined labelling allows users to construct their own vocabulary and to model their concept of the artifact. As mentioned in the context of JANUS [Girgensohn (92)] it is hard for non-programmers to identify the right object by name. By allowing users to define their own hierarchy and naming, we overcome this problem. Because of the similarity to the needs of software specification languages, we use a subset of UML for describing the artifacts‘ hierarchy. Brown [Brown (89)] also points out that design in general contains a hierarchical structure. So an adapted UML-supporting editor allows to define the artifacts hierarchy and to specify the connection between the design tool and the critiquing system (by Apple Script™ or Live Connect™) and is used as the artifact hierarchy editor.

2.2 Visual CiLa

The metric of a critique has to specify when a given artifact has a shortcoming. In the context of design such decisions depend on how different objects influence each other and on the relation between them. For example a yellow square does not attract the eye because it is just coloured yellow. But it attracts the eye because all other surrounding squares have less saturated colours [see Fig. 4].

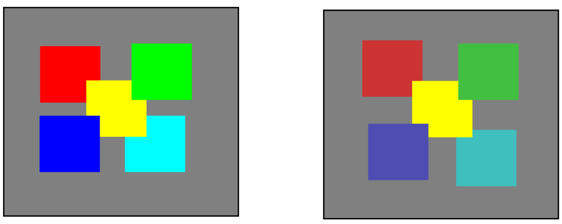


Figure 4: Visual Attributes of several objects influence each other. This in an example that good design often depends on several influencing factors and relations.

The challenge for the metric is to express the relations of the involved objects and to present this interactions easily. In JANUS [Girgensohn (92)] it becomes obvious that debugging and testing facilities for the inference process are necessary to support non-programmers. A visual language for the metric - which can visualize the inference process and the relations between the involved objects - can serve this purpose. Experts [Schiffer (98)] in the area of visual languages argue that visual programming languages can be easier to understand than linear text-based programming languages, because the visual expression of a programming task can more closely mirror the programmers way of thinking than the textual expression does. They are able to lower the load of identifying objects by allowing users to select an object from a given palette. We want to take these advantages and to support the visual recognition process, especially in the context of formulating critique metrics.

But first we want to examine the needed expressiveness and structure of the critiquing language: For example as mentioned above critiques can be seen as rules. This simple interpretation of critiques as a special kind of production rule also relies on the results of testing non-programmers‘ problem solving skills (see [Pane 98]). Tests showed that 54 percent of the non-programmers use some kind of production rule as programming style; 95 percent of the test subjects used set and subset specifications for performing operations on multiple objects. This leads to the conclusions that some kind of rule-like programming is an appropriate programming method for non-programmers. So the basic concept of rules - or more general first order logic - can be used for formulating metrics of critiques. This concept was applied in JANUS [Girgensohn (92)], IDIAS [Gutkauf 97] and VDDE [Harstad 93]. Also the concept of abstracta, introduced in Hermes [Stahl 93], can be seen as a predicate concept of first order logic. Referring to the known results [Wason 59], that expressing negative concepts is more difficult than expressing affirmative ones, no explicit negation is provided. So an easy to program *critique* has a DNF clause as condition. The predicates are called *abstracta*.

In order to provide a visual programming method for this language, we first look at existing innovative approaches of visual logic programming languages in order to construct an appropriate visual language for the metric of the critique. Agentsheets [Repenning 95] introduced the successful concept of spreadsheets and palettes to select predicates for a rule. Pictorial Janus [Saraswat 90] allows the display the inference mechanism by transporting objects between rules and VLPL [Puigsegur 96] has a completely visual defined inference process. TPM [Eisenstadt 88] shows the structure and inference process of Prolog programs. However, the challenge for the metric is that the relations of the involved objects are important. So the approaches used in Agentsheets, Pictorial Janus and VLPL fall short. The approach used in TPM is based on And/Or -Trees and describes relations between objects, but there is no intuitive understanding of the meaning of the tree itself. We decided to realize a spreadsheet-based visual language, which visualizes the inference process via floating objects and represents a real inference process. In order to allow an intuitive understanding of the structure of the critique, we transferred the *And/Or-Tree* to a pipe-, collector-, and filter-metaphor. These are explained in the following.

2.2.1 The Metaphors

To plug a critique metric together, one can select tiles from a palette and place them on a spreadsheet. We distinguish three kinds of tiles corresponding to the underlying metaphors: pipe tiles, filter tiles and collector tiles. These allow an intuitive understanding of the metric (i.e. the DNF-clause).

The Pipe Metaphor. Pipe tiles can have at most two inputs and two outputs. The basic concept is that the data flows from left to right and top down. Pipes can simply connect components of the puzzle as well as perform a kind of OR-connection: [Fig. 5] shows an example pipe tile with one data input and two data outputs. This means the datum *a* comes out at the right side of the tile and at the bottom side of the tile and the data *a* can fulfil the right or the lower part.

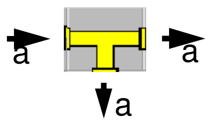


Figure 5: Pipe Tile

The Filter Metaphor. A filter represents a predicate and is build out of a head-filter-tile, an arbitrary number of input-tiles and an end-filter-tile. This allows us to formulate each n-inputs predicate. The principle is that the filter waits until all inputs received data. Then the predicate - associated with the filter - is checked for the given data. If it becomes true the data will be sent through the filter from left to right, as in a pipe. If not, the data will get stuck in the filter.

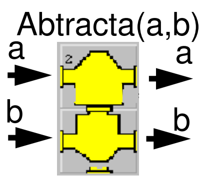


Figure 6: Fiter Tiles

The Collector Metaphor. A collector represents a connection of different pipes. It is based on the idea that a collector waits until the *data pressure* is high enough and then allows the data to flow.

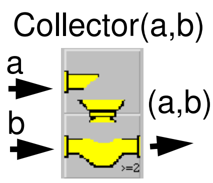


Figure 7: Collector Tiles

At the data inputs is a set of involved objects. [Fig. 7] shows the variables *a* and *b* at the inputs. At the data output - right side - is the set of *a* and *b*. The logic interpretation of this collector is that it presents the set of variables which are bounded by the interpretation I to fulfil it.

Other tiles in visual CiLa are the object tank tile and the critique tile. An object tank represents the need of a logical variable. Critique tiles present the advice as tile. The interaction between these tiles is shown in the following example.

2.2.2 An Example

It is clear that visual CiLa is able to present each kind of DNF clause. Now let‘s make clear how we present such an expression visually. Below [see Fig. 8] you see the expression in the form of a DNF-clause. The next diagram [see Fig. 9] shows the same expression represented by visual CiLa. In general you can distinguish two parts of the visual critique: the upper part presenting the first AND-clause and the lower part presenting the second AND-clause.

Metric in first order logic

$$\exists(x,y);(((Pred_1^1(x,y) \wedge Pred_1^2(x,y)) \vee Pred_2^1(y)) \wedge Bar(x) \wedge Bar(y)))$$

1th And-Clause Object-Constraints
2 th And-Clause

Figure 8: A typical DNF-Clause used for a critique-metric.

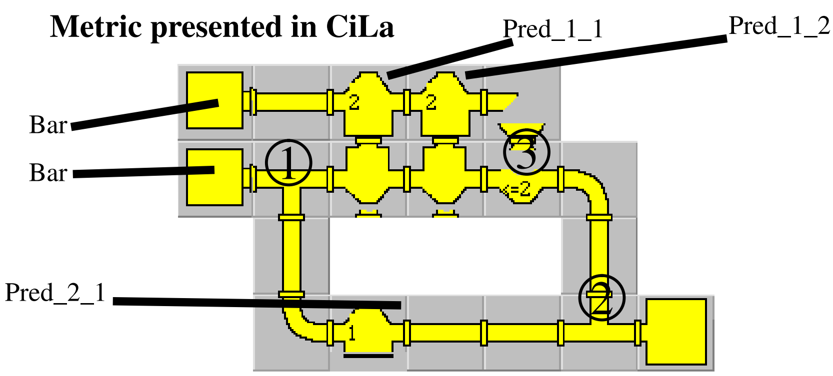


Figure 9: Example for an first order logic expression in visual CiLa. This example shows what the given logic expressions look like in visual CiLa. The upper part presents the first AND-Clause the lower part present the second AND-Clause. The numbers indicate special points in the graphic: (1) start of the or-connection (2) end of the or-connection (3) collector indicating that both objects are involved in this clause.

The frame of the critique is build by two object tanks at the left side and the critique tile at the right side. The object tank represents the connection to the given artifact and the critique tile represent the connection to the user; i.e. the critique tile contains the textual advice and textual reason for the user. The numbers in the diagram indicate interesting aspects of the visualization: (1) start of the or-connection (2) end of the or-connection (3) collector indicating that both objects are involved in this clause

2.2.3 Debugging Facilities

In general two faulty situations can occur: (a) a user gets a critique and does not know why, and (b) a user expects a specific critiques to appear but it doesn't. To overcome these problems, we must give the user the possibility to understand why s/he gets this critique and what the delivered texts mean. So s/he has to take into consideration which objects are involved in the critical situation and what relations they fulfil. Therefore the visual language has to represent the structure of the critique and must be able to simulate the process of inference. Visual CiLa shows the connection between the inferred objects and allows the visualization of the inference process by sending data through the pipes. So both requirements are fulfilled. [Fig. 10] demonstrates the concept of such a simulation.

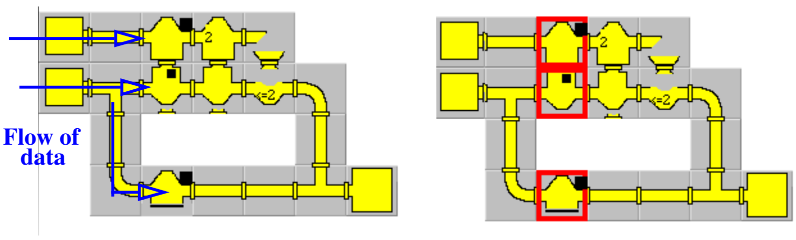


Figure 10: Simulation example of a critiquing metric in visual CiLa. After connecting two concrete objects of the artifact to the object tanks, the simulation starts. Pipes transport the data following their own form. Predicates check if the received data fulfils the predicate and then send it further to the right, or the data get stuck. Above both predicates are false - see marked as red - and the datum get stuck.

Such a simulation builds the starting point for debugging. On a more abstract level, debugging a critique means to compare the systems‘s actual behaviour against the user’s expectations and intentions. With poor debugging technology, this comparison must take place largely within the user’s head, and often overwhelms the ability of the users to deal with complexity [Liebermann 97]. [Liebermann 95] shows, that people first want to locate the source where the mismatch between their expectation and the system‘s behaviour takes place and then go into detail to correct the critique

So one needs a visualization of the process of checking a critique for a given artifact and the inference mechanism has to be presented in a linear, animated way. The integrated debugger of visual CiLa provides such a simulation. To allow an easy location of the mismatch, the control structure of the debugger is reversible; i.e. the user has bidirectional control over the debugging process. So a user can run forward until an error occurs and can then step backwards until the precise source of the error is located. The state or internal operation of the critiquing system is visualized by showing the data flow and the actual checked sub-condition. This demonstrates another advantage of abstracta: the structure to simulate remains simple and the textual description of the underlying predicate allows an easy understanding on a more detailed level.

2.2.4 Concept Summary of Visual CiLa

The language Visual CiLa allows the programming of critique metrics directly and is based on a special subset of first order logic. Visual CiLa is build on top of a pipe-, filter and collector-metaphor and allows users to plug critiques together by placing tiles on a spreadsheet. Furthermore Visual CiLa supports users in simulating the inference mechanism, which is represented by the flow of objects through the pipes. Filters and collectors visualize the functionality of inference and support the debugging process of such critiques.

2.3 The Supporting Modules

Besides the main component of the knowledge construction module, using visual CiLa, we realized so-called supporting modules which are mentioned only to get a complete view of the tool Coffein.

2.3.1 The Solution and the Reason Editor

In order to give a reason for a critique, one has to specify a reasoned opinion about an artifact; in the context of design this is called the design rationale. A design rationale gives the reason for designing an artifact the way it is or should be designed. To give a hint, i.e. to argue for a design rationale, one tries to make clear why a general rule can be applied to a given context. A critique has to allow these things: by defining a metric, one defines the condition of the referring rule. Like mentioned in Section 2, we assume that a user provides a negative example at the beginning of the process of constructing a critique. So the metric is valid for the given situation. The concrete objects - fulfilling the metric - are provided as a starting point to formulate the reason for the rule textually. In IDIAS it becomes obvious that people formulate the critiques referring to the concrete critical situation.

For example one user specified, that it is better to write long labels belonging to a bar top-down and not left-right. He constructed such a critical situation with a long label belonging to a blue bar, to make clear the spacing problem. Writing the text he referred to the blue bar, which was the critical bar with the long label in the given example; it was hard for him to translate from the given example back to the general rule.

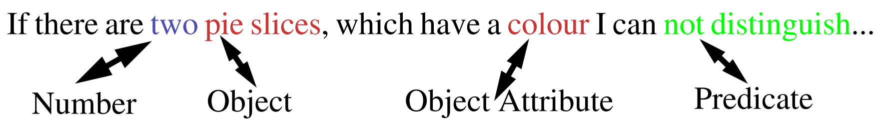
Visual CiLa supports the user in formulating the advice and reason in the context of the general rule: the concrete objects and cases of the metric can be referenced by clicking on them, and so errors like the one mentioned above are prohibited. The validation of the applied general rule can be provided by examples, pro and cons of the rule, links to related background knowledge and so on. Also suggestions on how to solve this conflict can help to get an understanding of the general rule.

If the user is also aware of a concrete solution, we ask him/her to identify the critical objects. Now we ask the user to solve the problem and trigger these action and generate a list of self-disclosure examples, as provided by Chart‘n Art [DiGiano 95]. These actions are textually described by the user and build the basis for providing a solution.

2.3.2 The Exploring Module

The exploration module allows the user to explore the already implemented functionality of the critiquing system. This means that users are able to find existing abstracta and to find already typed in reason- and advice-texts. So we implemented an abstraction parser that helps people to find the right abstracta for their metric and a text comparer, by finding similar texts to a given list of keywords.

The Abstraction Parser. Sometime users have problems in formulating their knowledge on a more formal level, such as first order logic. So we provide the possibility of describing the critical textual and interactively construct the first order logic expression. Let‘s assume we have the following textual description of the metric:



Referring to the artifact hierarchy, we parse the sentence looking for numbers, objects, objects attributes and colour names. The rest of the sentence - without expletives or fillers - are assumed as predicates (where the logical context is considered). So we look for similar predicates implemented in the database of abstracta. The determined abstracta are displayed.

The Text Comparer. We simply use the technique of LSA [Deerwester 90] for finding the relation between the query given by the user and the texts stored in the database as reason-texts and advice-texts. We provide the matching documents as a list similar to those used in traditional search engines like AltaVista™, Lycos™, metacrawler™ and so on.

3 The Knowledge Base Module

When a critique is formulated it is stored in the knowledge base. But not every update or extension of a knowledge base will lead to a better knowledge base, even if every part if the knowledge base is correct. This fact is called Knowledge Anomaly [Dücker 99]. For example a user adds a critique stating that printing text in blue is bad based on the fact that blue does not attract the eye. Another user suggests using a blue colour, which reminds us of the ink used for writing in school. A user who gets the first critique, changes the colour and then gets the second critique will be very confused, especially if this situation not only occurs with one pair of critiques but several pairs. To overcome this problem, we have to detect such inconsistencies and we must make them transparent to the user. So the knowledge base module checks each critique being placed in the knowledge base. It looks for inconsistencies between new critiques and the critiques already checked in. To do so, we first have to define what inconsistencies are (Section 3.1) and then we show how to store and use this information effectively (Section 3.2).

3.1 Definition of Inconsistency

For production-rules Ojelanki et. al. [Ojelanki 92] identified four types of rule inconsistencies: (1) *logical equivalence*, (2) *logical inclusion*, (3) *condition inconsistency*, and (4) *action inconsistency*, like shown in the following table [see Table 2]:

Inconsistency in the field of AI	**conditions are equivalent**	**conditions are not equivalent**	**conditions are inclusions**
actions are equivalent	logical equivalence	condition inconsistency	logical inclusion
actions are not equivalent	action inconsistency		

Table 2: Inconsistencies of Production Rules

Another type of inconsistency [Ojelanki 92] - a kind of rulebase inconsistency - is *logical incompleteness*; this means that no computation can be done for at least one situation. Our problem is that we have no secure information about the actions the user is told, especially considering that the advice in a critique can contain different suggestions for solutions (e.g. different actions are recommended). This *action inconsistency* is an important part of the design process and traditional definitions of inconsistency fall short. So we must take a closer look at the conditions, assuming that the information we have about the action - the advice and the reason - of the rules is very fuzzy. For example the fact that two actions are not textually identical does not ensure that the actions - i.e. the resulting actions of the user - are different.

Fist we want to check the consistency of the conditions of the two critiques. Let‘s assume the following formalism: $critique_i: (cond_i \rightarrow advice_i, reason_i)$. Now let‘s define a condition inconsistency for two critiques $critique_1, critique_2$. Now we can formulate consistency constraints valid for all designs. All possible relations between the two conditions can be formulated as a boolean function f_2 in this form:

$$\forall designs \qquad f_2(cond_1(design), cond_2(design))$$

Additionally we can assume that each of both conditions (at least for one design) becomes true and that this condition is false for at least one design. So f_2 has to fulfil the following:

$$\exists((x, x', y, y') \in \{0, 1\}) \qquad f_2(0,x) \wedge f_2(1,x') \wedge f_2(y, 0) \wedge f_2(y', 1)$$

Now let‘s look at all possible boolean functions f_2 [see Tab. 3]:

con1	con2	$\forall designs \quad f_2(cond_1(design), cond_2(design))$															
0	0	0	0	0	0	1	1	1	0	0	1	0	1	1	1	0	1
0	1	0	0	0	1	0	1	0	1	0	0	1	1	1	0	1	1
1	0	0	0	1	0	0	0	1	0	1	0	1	1	0	1	1	1
1	1	0	1	0	0	0	0	0	1	1	1	0	0	1	1	1	1
											↑ identity	↑ completeness	↑ alternatives	↑ refinement	↑ generalisation	↑ completeness	↑ separation

Table 3: Possible Relations between two critiques

The rows with a white background contain boolean functions which hurt the condition given above. So the red rows present possible boolean functions for the relation between two critiques. As seen, we identified six situations [see Tab. 4]:

Name	**Description**	**Problem**
identity	both conditions are either true or both are false.	the critiques criticise the same design, but have different lines of argument.
generalisation	whenever $cond_2$ is true also $cond_1$ is true.	$cond_1$ is a more general case of $cond_2$.
refinement	whenever $cond_1$ is true also $cond_2$ is true.	$cond_1$ is a special case of $cond_2$.
completeness	every time critique with $cond_1$ or $cond_2$ occurs; e.g. no design without a critique.	It is not possible to create a critique without a critique.
alternatives	It is not possible that both critiques occur at the same time; e.g. their compete.	These are potential conflicting point of views.
separation	Everything can happen.	no problem

Table 4: Condition Inconsistencies for Critiques

Condition identity, *condition refinement, condition generalisation* and *completeness* are seen as inconsistencies caused by overload and redundancy of information. *Condition alternatives* and *condition separation* are relations with cause no problems. The same situations can be defined between *n* conditions of critiques [Thies 99a]

3.2 Usage of (In)Consistency Information

After determining which inconsistencies exist, we build up a graph which presents these relations. The algorithm for construction is as follows:

- Each node represents a class of equivalent conditions, and has a link to all line of arguments. Formal: if $critique_i$ and $critique_j$ are condition identical we have a node $node_t$ which has a link to $cond_i$, $advice_i$, $reason_i$ and a second link to $cond_j$, $advice_j$, $reason_j$.
- Each black directed edge in the graph represents a condition refinement of two critiques. Formal: if $cond_i$ is a refinement of $cond_j$ and $node_s$ has a link to $cond_i$ and $node_t$ has a link to $cond_j$ then there is a black edge from $node_t$ to $node_s$.
- Each red undirected edge in the graph represents a ***condition completeness*** between two critiques. Formal: if $cond_i$ is condition complete to $cond_j$ and $node_s$ has a link to $cond_i$ and $node_t$ has a link to $cond_j$ then there is a red edge from $node_t$ to $node_s$.
- Each blue undirected edge in the graph represents an alternative between the two critiques. Formal: if $cond_i$ is an condition alternative to $cond_j$ and $node_s$ has a link to $cond_i$ and $node_t$ has a link to $cond_j$ then there is a blue edge from $node_t$ to $node_s$.

Please notice that all critiques linked to a node have the same relations to other critiques; that is to other nodes. Let's assume the graph looks like presented in [Fig. 11].

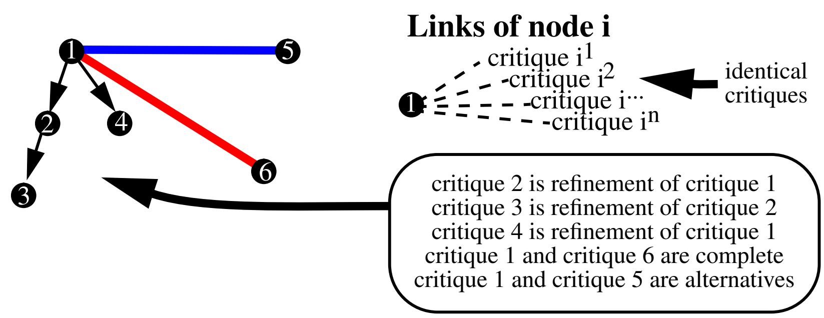


Figure 11: Critique Inconsistencies displayed as Graph (Example)

We can now use this graph for efficient evaluation of the critiques. First we check all nodes with no incoming edges and label them with the result; i.e. true or false. Then we follow the outgoing edges: nodes connected by a red edge are labelled with the opposite of the own label. If the node is labelled true, then the black connected nodes are evaluated. If the node is labelled false, then the blue connected nodes are evaluated and all black connected nodes are labelled with false. This loop continues until no further nodes can be labelled. All nodes labelled with true present critiques which are valid for the given context.

4 The Module for Knowledge Presentation

Designing an artifact is an open-ended problem (see [Rittel 73]) which allows a diversity of solutions. The situationally best solution depends on the user‘s goals, as well as on his/her specific background and abilities; i.e. on the users perspective. To obtain this solution, we often want to have a codesigner: (1) Who tries to adapt to our perspective; for example to our explicit and tacit goals. (2) Who has a consistent point of view but (3) also mentions other people‘s opinions and point of views. To construct such a codesigner, we use the knowledge base of critiques from different designers as a basis.

But simply putting together these knowledge bases is no solution, as shown in [Schekelmann (98)] and [Harstad 93]. We identified four types of inconsistencies which can occur by simply merging the knowledge bases: alternatives, identity, refinement and completeness.

For each inconsistency, an algorithm and a concept is developed to construct a consistent point of view for the virtual codesigner. The basis for all algorithm is the graph introduced in Section 3.2 and a user model which allows the comparison of the recipient of a critique with the author of a critique. This model is constructed by looking at the user‘s interactions with the critiquing system. This model will be explained in the following section. Afterwards we show how we can implement the user model and the graph to select a point of view for the virtual codesigner.

4.1 The User Model

Visual CiLa allows users to formulate and modify critiques of the critiquing systems. So users are not only recipients of delivered critiques, but they are also able to adapt them to their special needs, vocabulary, background and line of argument. So they become the authors of the adapted critique; thus, recipients of critiques are normally also authors of critiques. So a user has two roles: (a) recipient of critiques and (b) author of critiques. Therefore we have two kinds of information about the user: (a) The critiquing statistic; i.e. which critiques occur for the user; (b) the versions, lines of argument of critiques which are adapted by the user. The purpose of the user model is to get indicators of which texts are more comprehensible and appropriate to a specific user than others. These two kinds of information are used as follow. The statistic allows us to construct dependent sets of critiques by graph partitioning; this means that critiques of one set normally occur together and so present a specific lack of knowledge on the user‘s part. The user-adapted or formulated critiques are used to extract information about the user‘s specific wording. Latent Semantic Indexing [Deerwester 90] and also graph partitioning allows us to group critiques which contain the same wording. So the user model contains information which kind of wording group s/he belongs to and which sets of critiques present the user‘s lack of knowledge. We can compare users on the basis of lack of knowledge and wording. Of course the user is also able to inspect his/her user model. For more details see [Thies 99c].

Here we see that end-user modifiable critiquing systems - used as a basis for a group memory - allows special input for user modelling. Standard techniques of graph partitioning are used to establish a user model which allows a measurement of comprehensibility of other people‘s arguments for the user - depending on the user‘s specific wording and background knowledge. This helps to overcome the problem of information overload and incomprehensible information. A small test case with 5 persons produced promising results.

4.2 Selecting a Point of View

Let G be the graph of critiques as discussed in Section 3.2. Each directed path (i.e. each black path - $path_i$ - paths of length one are allowed) corresponds to one hint $hint_i$ of the virtual codesigner. Say $Set_{hint} = \bigcup_{path_i} hint_i$ is the set of all possible hints for the actual critique base. Define $authors_i$ is the list of authors connected to $hint_i$; e.g. $authors_i$ is the set of all authors of all critiques linked to a node of $path_i$. Formal: $authors_i = \bigcup_{node \in path_i} \{author | author \rightarrow node\}$.

4.2.1 Step 1: Complete Critiques:

First we decide the point of view based on complete critiques: If there is $path_i$, $path_j$ and there is a node $node_i$ of $path_i$ which is connected by a red edge with $node_j$ which is a node of $path_j$, then calculate the minimal distance of the recipient user and all member of $authors_i$ and all users of $authors_j$. Select the author $author_t$ of $authors_i \cup authors_j$ so, that the distance between that actual user and the selected author $author_t$ is minimal based on the calculated user model. The referring $path_t$ $t \in \{i, j\}$ - specially $hint_t$ - is marked as „point of view“. Please notice that we cannot guarantee that the selected author only occurs in $authors_i$ OR $authors_j$. But tests showed that this situation does not occur in real life. If this situation occurs, we select the $path_t$ $t \in \{i, j\}$ - especially $hint_t$ - randomly and mark the other hint as „contrary point of view“.

4.2.2 Step 2: Alternatives of Critiques

If there is $path_i$, $path_j$ and there is a node $node_i$ of $path_i$ which is connected by a blue edge with $node_j$ which is a node of $path_j$, and $path_j$ e.g. $hint_i$ is labelled with „point of view“ then mark $hint_i$ with „out of point of view“.

4.2.3 Step 3: Refinement of Critiques

For the remaining unlabelled hints in Set_{hints} we select the point of view based on refinement critiques. If a path $path_i$ has no label and contains a subpath $path_i'$ which is labelled, then label $path_i$ the same way. Label each remaining unlabelled hint $hint_t$ with „point of view“.

After applying these three steps we get the Set_{hints} where each element is labelled with: „point of view“, „out of point of view“, „contrary point of view“. Now we use this information to construct the behaviour of the virtual codesigner.

4.3 Construction and Behaviour of the Virtual Co-Designer

Let‘s assume the user provides a designer artifact and checks for critiques. Say G_{lab} is the resulting labelled graph of the critiques, as defined in Section 3.2. Define G'_{lab} as the subgraph of G_{lab} only containing the nodes which are labelled true and only those edges which connect true labelled nodes. For each **maximal** path $path_i$ of G'_{lab} look up label $label_i$ of the corresponding hint $hint_i$. So $label_i$ is „point of view“, „out of point of view“ or „contrary point of view“.

Case 1: Point of View.

Select the most specific node $node_t$ of the related path $path_i$ - i.e. at the ending of the path. Select the critique of this node $node_t$, whose author has a minimal distance to the actual user. The advice and reason of the selected critique are displayed directly.

All other critiques corresponding to this path are collected in a list, then they are ranked depending on the user model. This list is also shown as a selection list to the user. So the system provides information first and directly which is assumed to be more comprehensibility for the user. The selection list can be changed by the user and should be seen as support and not as a dictate. The user is also able to inspect his/her user model getting wording examples and examples of typical received critiques. The texts of the complete critique (if they exist) are mentioned as a opinion from another designer. So the display is structured as seen in Fig. 12.

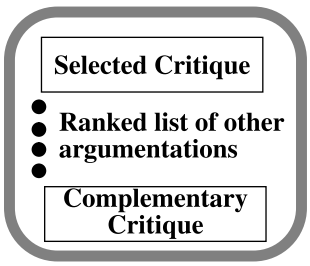


Figure 12: Structure of Critique Display: (a) Point of view critique, (b) other argumentations as list and (c) complementary/alternative critiques

Case 2: Out of Point of view. Select the critique $critique_{out}$, like the selected and directly displayed critique in Case 1. In the same way select $critique_{in}$, for the alternative connected path. Display as selected critique:

„Other people say the following:" +
text of $critique_{out}$ +
„But I thing: „ +
text of $critique_{in}$.

Figure 13: Example, how to display a selected critique

The rest is calculated like Case 1.

Case 2: Contrary Point of View. Show nothing to the user.

We decide which critique based on the author‘s knowledge is the best fitting for the user, also considering the existing inconsistencies. Those critiques become the „point of view“ of the constructed codesigner. The other critiques are considered as different points of view the codesigner knows and is able to mention or allude to. So the codesigner is also able to prod the user to think about different aspects or point of view.

5 A Usage Scenario

The project Intersim [Arias 97] tries to get citizens involved in the planning of bus routes and city design. They built a simulation of a city including traffic and bus routes with Agentsheets [Repenning 95] in order to allow citizens to explore the possibilities and disadvantages of new bus routes. One goal is to make this simulation available in different public libraries. So that many citizen at different places have the opportunity to get involved in the design process. We want to set up our usage scenario in this context:

A citizen called John looks at the simulation at the public library in Boulder. He looks at the bus route and realizes that a residential area has no bus stop. So he formulates a critique that a living area should have a “reachable” bus stop. The abstraction parser delivers the abstracta „near“, „very near“ and „not far away“ as possible abstracta. John looks up, if „near“ is the appropriate abstracta, he formulates his critique with this abstracta. Then he tests the critique in different situations and gets not satisfactory results. He decides that near is a too rigorous restriction and he tries „not far away“ which works fine for his requirements. He places a bus stop in the area, demonstrating a solution as well as for solving the problem itself. After playing around a while he stops this session.

The next day a citizen group at the university of Boulder meets and starts the same simulation as John did the day before. Sarah sees the new bus stop and removes it because she lives near this place and hates traffic. Immediately John‘s critique pops up and informs Sarah about his motivation for placing a bus stop there. Now Sarah has the possibility to make her own critique (referring her problem with traffic) public for discussion. After pointing out her objection the group argues that common issues are more important than personal issues. So Sarah provides a compromise by moving the bus stop a few meters and putting a critique into the system that she don't want to have a bus stop very near of her house. If John looks at the simulation again he will see the change and get the supporting information provided by Sarah‘s critique.

So the critiquing system is used as a group memory, even over distance, and is context sensitive. People can be sure that their constraints are considered, if they formulate the referring critiques. The argumentation for personal constraints are placed on a logical level and are based on hard facts, which can be proved and criticised. So compromises and solutions are easier to find.

6 Conclusion

We introduced the tool Coffein, a development environment for critiquing systems. The three main modules - knowledge construction module, knowledge base module and knowledge presentation module - are described. Visual CiLa (the basis of the knowledge construction module) is explained in detailed. We believe that this language is a framework which allows design experts to formulate, to test, and to debug their expertise as critiquing knowledge. A test with 30 children showed that even 9th graders with no programming skills were able to formulate critiques in the context of the game GO, to test them and to debug them. It also became obvious in the test how the visualization of the process supports the understanding of the mechanism itself. We identified four types of inconsistencies and showed how they can be stored and used as an effective inference mechanism in the knowledge base module. This information about inconsistencies is used in the knowledge presentation module: it selects a point of view depending on the user model and represents the information user specific and consistent; it also alludes to other point of views.

By providing support for expressing design knowledge in the form of critiques and by delivering these constructed knowledge in a context sensitive, consistent points of view, we provide a special kind of group memory. This memory can be filled even by non-programmers by using the visual critiquing language for formulating critiques as a piece of the designer‘s knowledge. The structuring and combining of these pieces of advice is done automatically and presented in a human manner, by a virtual codesigner. We therefore lower two basic burdens of group memories: to populate a group memory and to access information stored in a group memory.

Acknowledgements

I would like to thank Prof. Dr. Ernesto G. Arias, Prof. Dr. Gitta Domik, Prof. Dr. Gerhard Fischer, Dipl. Inform. Michael Thies, as well as the group Intersim.

References

[Arias 97] Ernesto, G. Arias: „The Bridging of the Virtual and Physical: The InterSim as a Collaboration Support Interface"; Proceedings AI-ED 97, 8th World Conference On Artificial Intelligence In Education (1997)

[Brown (89)] Browns, D.C., Chandrasekaran, B.:„Design Problem solving (Knowledge Structures and Design Strategies)"; Morgan Kaufmann Publischer (1989)

[Deerwester 90] Deerwester, S., Dumais, S. T., Furnas, G.W., Landauer, T. K., Harshman, R.: „Indexing by latent semantic analysis"; Journal of the American Society for Information Science, 41, 6 (1990), 391-407.

[DiGiano 95] DiGiano, C., Eisenberg, M.: "Self disclosing design tools: a gentle introduction to end-user programming", Proc. DIS'95 Symposium on Action Systems, G.Olson and S. Schon, ACM Press, Michigan (1995).

[Domik 94] Domik,G., Gutkauf, B.: "User Modeling for Adaptive Visualization Systems"; IEEE 1994, (1994).

[Dücker 99] Dücker, M., Gutkauf, B., Thies, S.: „A Collaborative Development Environment for Design-Oriented Critiquing Systems"; Proc. of the 32th Hawaii Int. Conference on System Sciences (HICCS 99), (1999).

[Eisenstadt 88] Eisenstadt, M., Brayshaw, M.: „The Transparent Prolog Machine (TMP): An Execution Model and Graphical Debugger for Logic Programming"; Proceedings of the Conference on Logic Programming, Austin, Texas, (1988).

[Fischer 91] Fischer, G., Mastaglio, T.: „A conceptual framework for knowledge-based critic systems"; Proc. Decision Support Systems 7, (1991).

[Fischer 90] Fischer, G., Girgensohn, A.: „End-User Modifiability in Design Environments"; Proc. Human Factors in Computer Systems, CHI '90, ACM, Seattle, WA, (1990).

[Girgensohn (92)] Girgensohn, A.: „End-User Modifiability in Knowledge-Based Design Environments", Ph.D. Dissertation, Department of Computer Science, University of Colorado (1992).

[Gutkauf 97] Gutkauf, B., Thies, S., Domik, G.:„A User-Adaptive Chart Editing System Based on User Modeling and Critiquing"; Proceedings of the Sixth International Conference UM97, Sardegna, Italy, Springer Verlag (1997).

[Görz (95)] Görz, G.:"Einführung in die künstliche Intelligenz", Addison-Wesley, Berlin (1995).

[Harstad 93] Harstad, B.: „New Approaches for Critiquing Systems: Pluralistic Critiquing, Consistency Critiquing, and Multiple Intervention Strategies", PHD-Thesis at: Department of Computer Science, University of Colorado at Boulder, December, (1993).

[Knopik 91] Knopik, T.: „Methoden des maschinellen Lernens für den interaktiven Wissenserwerb", Stuttgart Univ., Diss., (1991)

[Liebermann 97] Liebermann, H.: "Introduction and Guest Editor, Special Issue on Debugging and Software Visualization", Communications of the ACM, April (1997).

[Liebermann 95] Liebermann, H., Fry, C.: „Bridging the Gap Between Code and Behavior in Programming"; Proc. ACM Conference on Computers and Human Interface [CHI-95], Denver, April (1995).

[Liebermann 89] Liebermann, H.: „Programming Descriptive Analogies by Example"; Proc. Workshop on Inheritance Hierarchies in Knowledge Representation, Viareggio, Italy (1989).

[Ojelanki 92] Ojelanki, K., Bryson, N.: „A Formal Method for Analyzing and Integrating the Rule-Sets of Multiple Experts“; Proc. Information Systems; Vol. 17, ISBN: 0306-4379 925500-000, (1992).

[Pane 98] Pane, J. F., Ratanamahatana, C. A., Meyers, B. A.:„Analysis of the Language and Structure in Non-Programmers‘ Solutions to Programming Problems“, Report at: Carnegie Mellon University, School of Computer Science (1998).

[Puigsegur 96] Puigsegur, J. , Augusti, A., Robertson, D.: „A Visual Logic Programming Language“; Proceedings of the Conference on Visual Languages, Boulder, Colorado (1996).

[Repenning 95] Repenning, A., Summer, T.: „Agentsheets: A Medium for Creating Domain-Oriented Visual Languages“, Proc.„IEEE Computer“, Number 3, Volume 28, (1995)

[Rittel 73] Rittel, H. W. J., Webber, M.: „Dilemmas in a general theory of planning“; Proc. Policy Sciences, 4:155-169, (1973).

[Saraswat 90] Saraswat, V., Kahn, K. M., Levy, J.: “Janus: A Step forward towards Distributed Constraint Programming“; Proceedings of the North American Conference on Logic Programming, Austin, Texas, MIT Press, October (1990).

[Schiffer (98)] Schiffer, S.: “Visuelle Programmierung“, Addison Weslay Longmann Verlag, Bonn, ISBN 3-8273-1271-x, (1998).

[Schekelmann (98)] Schekelmann, A.: „Materialflußsteuerung auf der Basis des Wissens mehrere Experten“; HNI-Verlagschriftenreihe, ISBN 3-931466-46-9 (1998).

[Schön (83)] Schön, D.A.: “The Reflective Practitioner - How professionals think in action“; Basic Books, Inc., Publishers, New York, ISBN 0-465-06874-X, (1983).

[Cypher (93)] Cypher, A.: „Watch What I Do, Programming by Demonstration“, MIT Press, ISBN 0-262-03213-9 (1993).

[Stahl 93] Stahl, G.: „Interpretation in Design: The Problem of Tacit and Explicit Understanding in Computer Support of Cooperative Design“; Ph.D. dissertation. Department of Computer Science. University of Colorado at Boulder (1993).

[Thies 99a] Thies, S.: „Konstruktion von pluralistischem Visualisierungswissen“; GI-Graphiktag 1999, Rostock (1999).

[Thies 99b] S. Thies; „User Modeling for Critiquing“; Proc. User Modelling UM 99, Canada, Banff (1999).

[Thies 99c] Thies, S.: „Benutzeradaptivität für Critiquing Systeme“, Abis 99, Magdeburg (1999).

[Thies 95] Thies, S.: „Ein benutzeradaptives Kritiksystem zur Erstellung von Business-Grafiken, modelliert durch eine Regelsprache“; Diplomarbeit, Universität Paderborn, FB17, vorgelegt bei Prof. Dr. Gitta Domik (1995).

[Wason 59] Wason, P. C.: „The Processing of Positive and Negative Information“; Quarterly Journal of Experimental Psychology, Number 11, Volume 92, (1959).