

A Structured Interactive Workspace for a Visual Configuration Language

J.-Y. Vion-Dury* F. Pacull

Rank Xerox Research Centre
6 Chemin de Maupertuis 38240 Meylan - France
{Jean-Yves.Vion-Dury,Francois.Pacull}@grenoble.rxrc.xerox.com

Abstract

This paper shows how language technologies such as the automatic generation of parsers for analyzing user actions and visual parsing can be applied to build a flexible tool specialized in complex specification tasks, namely the configuration of distributed applications. The central issue is to propose to structure the workspace through a syntax of user actions on one hand, and a syntax of visual representations on the other hand. From the tool designer side, this approach makes the core of the tool explicit through the grammar rules, and eases the generation of final code by simplifying verifications. From the user side, properties of the workspace generated from high level specifications increase the usability and improve the perception of the underlying semantics.

1 Introduction

Significant efforts have been made in the VL community to establish the theoretical foundations of visual syntaxes. The challenge here is to reflect the expressivity of visual representations, which are intrinsically multi-dimensional, as opposed to textual representations considered as uni-dimensional. The important point is to be able to formalize such syntaxes into grammars suited to generating or parsing structures at a reasonable computing cost. Important advances have been made, concerning relational grammar[17, 16, 7], graph grammar[11] and positional grammar[6] formalisms. These latter seem to be the most tractable, by offering efficient bottom-up parsing of unordered visual items while preserving a satisfactory expressive power and simplicity. Fewer works (such as [12, 10]) consider the temporal structure of interactive systems, which may be in our opinion, the most innovative issue brought by computer technologies to information processing systems. This addresses the temporal structure of user action as well as the structure of visual animation. Ultimately, the relationship between the temporal syntax of user actions, visual animation and visual syntax

should be unified in a common formal framework which could be the foundation of a human-computer communication theory. This paper proposes a humble step in this direction by presenting a theoretical approach and its partial application to structure the user interaction (through the LR-grammar/parser approach), the visual representation (through positional-grammar/parser approach) and the visual operations and animations (by using graphical instructions, based on a formal visual type system).

Section 2 outlines the basic concepts proposed. Section 3 puts emphasis on the configuration of distributed application problems, the target of the tool presented in section 4. Section 5 concludes and sketches future work.

2 An Interactive Structured Workspace

This section presents the main features of the workspace model proposed. A visual type system allows us to structure the basic entities of the language and to formally define operations permitted on various types. This approach aims to bring simplicity via type polymorphism and to foster the scalability of visual representation. Polymorphism simplifies the handling of the visual entities, as well as the graphical instruction set, which will be presented at the end of the section. An example of polymorphism is given by the possibility of changing the size of an object of any visual type simply by changing the value of its size attribute. Scalability is a central problem when dealing with high quantitative complexity. Authors of [5, 15] addressed this important point in 2D and 3D contexts. The visual type system outlined here then puts emphasis on object resizing facilities in order to simplify the space management, and to provide visual language designers with integrated operations. Section 4 illustrates how this capability can be applied to potentially large workspaces.

2.1 A Visual Type system

It is commonly accepted that the basic language structures are composed of signs, organized in a system from which higher level structures are based (lexical units and syntactic units). In our framework, the sign system is an alphabet of visual items we call *glyphs*.

* in partnership with INRIA Rhones-Alpes (Projet SIRAC), 655 Av de l'Europe, 38330 Monbonnot Saint-Martin France

Basic types are:

$$\begin{aligned}\tau_{point} &= (a_{pos}, a_{color}) \\ \tau_{line} &= (a_{pos}, a_{color}, a_{size}, a_{orient}) \\ \tau_{poly} &= (a_{pos}, a_{color}, a_{size}, a_{orient}, a_{shape}) \\ \tau_{text} &= (a_{pos}, a_{color}, a_{size}, a_{orient}, a_{string})\end{aligned}$$

where attributes $a_{pos}, a_{color}, a_{size}, a_{shape}, a_{orient}$ represent perceptual values, i.e. physical parameters that a human observer perceives directly. For simplification, attribute a_x of a type t will be written t_x . Note that these parameters are independent (you can change one without affecting the other), which brings more expressivity to map information into visual representations. Fig 1 shows how attributes are structured and encoded.

Attribute	structure	encoding
a_{pos}	(x, y)	$x, y \in \mathbb{R}$
a_{color}	(h, s, v)	$h, s, v \in [0 \dots 1]$
a_{size}	s	$s \in \mathbb{R}^{+*}$
a_{orient}	α	$\alpha \in \mathbb{R}$
a_{shape}	$(\{v_1, \dots, v_n\}, \gamma)$	$v_i \in \mathbb{R}, \gamma \in \mathbb{R}$

Figure 1: basic perceptual attributes

Glyphs and composite types. From these basic types, glyph types are defined by: $\tau = \tau_{point} | \tau_{line} | \tau_{text} | \tau_{poly}$ and composite glyphs can be built up from either basic or composite glyphs by using the following composition:

$$\begin{aligned}\tau &= (a_{pos}, a_{size}, a_{orient}, \Phi) \\ \Phi &= \{\tau_1, \dots, \tau_n\}\end{aligned}$$

which assumes a set of constraints so that, $\forall \sigma \in \Phi$,

$$\begin{aligned}a_{pos} \vec{\sigma}_{pos} &= cste \\ a_{size} - \sum \sigma_{size} &= cste \\ a_{orient} - \sigma_{orient} &= cste\end{aligned}$$

The graphical engine that handles visual representations of composite glyphs has to maintain the consistency of position, size and orientation through constraint solving according to the previous definitions. Thus, the whole tree of sub-objects can be manipulated as if it were an unique atomic object.

Visual representation. Although attributes *per se* cannot be directly represented, they are combined together to build the actual visual items perceived by the user. The most specific attributes are:

Orientation, i.e. the angle of the glyph with the vertical \vec{V} . As an example, a line l is visually built by computing the segment $[p_0, p_1]$ such that $angle(\vec{V}, \vec{p_1 p_0}) = l_{orient}$ and $\vec{p_0} l_{pos} = \vec{p_1} l_{pos}$ (i.e. l_{pos} is the middle of the segment).

Shapes, specified through a pair $(\{v_1, \dots, v_n\}, \gamma)$, where v_i are the length of vectors from which the coordinates of the shape contour can be computed, relative to pos . The number of values n thus determines the number of edges and each value v_i can be associated with a vector $\vec{v}_i = v_i \times rotate(\vec{V}, \frac{2\pi}{n} + \alpha + \gamma)$. Note that the absolute angle of the first vector of the shape with \vec{V} is given by α (angle of the glyph) and γ (angle of the shape). The actual polygon $\{p_i\}$ representing the shape of a type $t = \tau_{poly}$

is computed by $p_i = t_{pos} + (\vec{v}_i \times (t_{size} / \max(\|\vec{v}_i\|)))^1$. Fig 2 shows examples of shape definitions. This format defines a shape algebra which allows morphing operations. It permits the generalization of animation commands to all kinds of attributes by accepting the basic algebraic operators $+$, $-$, \times , $/$ (see 2.3).

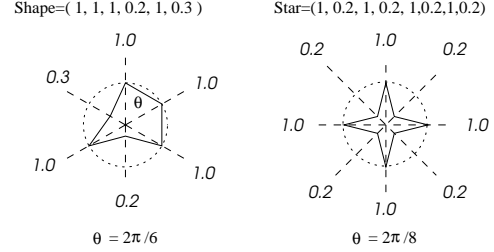


Figure 2: representation of shape attributes

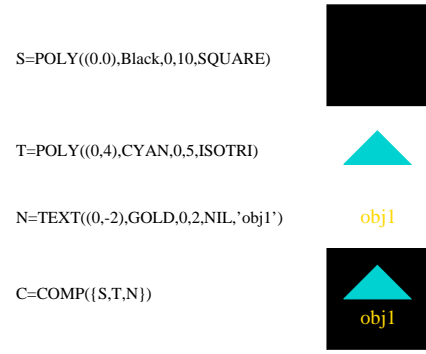


Figure 3: example of glyph composition

Properties of the type system. The reader may note that these definitions bring polymorphism since for all visual types, the $a_{pos}, a_{size}, a_{orient}$ attributes carry the same semantics and can be used and transformed through the same set of operators. For the designer of visual languages, as well as for the underlying architecture required, this particularity allows many simplifications. Üsküdarh's Visual Object Description Language [14] uses a layout grammar in a similar way (structural definitions and composition), excepted that layout relationships are of a higher-level and do not specify such a polymorphism. Our type system infers basic attributes of composite objects from lower level objects, in order to offer more simple definitions, but also generic operations and behaviors. Fig 3 shows an example of a composite glyph specification, and the visual item obtained. Although this system appears to be simple, its expressive power is high since basic items can be combined without qualitative or quantitative limits. Moreover, if 2D workspaces are addressed in this paper, this type system

¹ + and \times are here vectorial sum and product

is originally designed for 3D workspaces in order to take advantage of its potential richness² [15, 12].

2.2 Syntax of Actions

User actions can be captured through mouse motion, buttons and key press/release events. All these events are sequential in the temporal dimension. It can thus be envisaged to use known grammatical formalisms and associated parsers to specify legal user actions, provided that the scanning is done over a continuous input stream rather than a finite sequence. Note that here we mainly address the *operational* aspect of the problem, rather than the *qualitative* aspect, as done in [9] by using (high level) temporal extensions of a UAN (User Action Notation). The main issues of the action parsing we propose are:

Designation of objects and actions in the workspace. The alphabet of glyphs is defined by using the type system previously described. This means that, as for lexical definitions of textual languages, the concept of the grammar designates terminal symbols with visual type names. To symbolize the actions performed by the user on visual objects, the graphical engine which handles glyphs generates a flow of attributed events described in Fig 4. Note that the semantics of the actions ensures that leave/release events are always preceded by corresponding enter/press events.

user action	lexical unit	attribute
enters an object τ	' $+\tau$ '	object Id
leaves an object τ	' $-\tau$ '	object Id
presses a button n	'press n '	n
releases a button n	'releasen'	n
presses a key k	'keypress'	value
releases a key k	'keyrelease'	value

Figure 4: events generated by the graphical engine

Expressivity of grammar formalism. It is difficult to make a statement that a grammar is sufficiently powerful. Therefore, we choose the largest class of unambiguous context-free grammars: LR(1) grammars (see [1]), which produce bottom-up (and also input driven) parsers, well adapted to flow analysis.

Error handling without rigidity. This point is central to user-friendly interactive systems. We propose to first use the particularity that LR(1) parsers never reduce a rule if the input is wrong (it is not true for the LALR(1) parser [1]), and second the structural symmetry of lexical units. The first point ensures that, since the error is detected at the shifting stage, no wrong actions can be initiated (and thus, no corrective action has to be performed afterwards). For the second point, we noticed that all events are symmetric: if an unexpected *enter* event triggers an error, then

²in this case, a third basic type $\tau_{polyhedron}$ is added, and shape vectors have richer structures

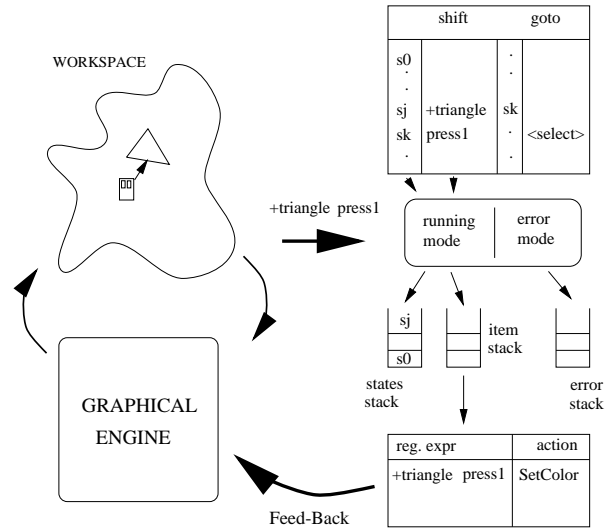


Figure 5: user action parsing

the parser stops the analysis, switches into an error mode and tries to resynchronize on the corresponding *release* event. A stack handles cumulated error sequences: new *enter/press* events are pushed and *leave/release* events allow popping. The parsing thus restarts when the error stack is empty. User awareness is ensured by displaying an error specific mouse form, and corrective actions are proposed to the user (after analyzing the stack). This generic approach, although simple, leads to a surprisingly flexible error management.

Visual feed back. The difficulty here is to deal with the dynamics of interactive systems: how to relate the parsing of user actions to the graphical instructions needed for the visual feedback. In the *batch processing* model, actions can be associated with rule reductions. For instance, the recognition of: $Select \rightarrow +triangle\ press1\ release1^3$ can produce the action $SetColor(triangle.objId, RED)$. The problem is that, due to the look-ahead behavior of LR(1) parsers, the rule is not reduced before the input of the following event, and thus, the visual feed back cannot be synchronized to the user actions. In our framework, actions can be attached either to the shift or to the reduce phases. The parsing algorithm works with two shifting stacks (state stack) $\langle s_0, \dots, s_j, s_k \rangle$ and (items stack) $\langle \dots, +triangle, press1 \rangle$ instead of the standard interleaved stack $\langle \dots, s_j, +triangle, s_k, press1 \rangle$. Thus, the visual language designer can specify feed-back actions by using special rules $expr \Rightarrow actions$ where $expr$ is a regular expression applied to the item stack that must match to trigger the action processing.

Infinite input/output streams handling. The input stream is

³The user enters visual item *triangle* and click on mouse button 1

seen as a FIFO from which items are expunged after recognition. For the output parse tree, it is up to the designer to handle deletion through reduction actions, depending on the context and functionalities attached to the syntax.

As a conclusion, this approach allows us to specify the interaction through grammatical rules, to recognize legal action sequences and to forbid wrong actions. Invariant properties of the workspace can therefore be ensured (e.g. a triangle can never be dragged onto another triangle). Dynamicity is addressed through the graphical instructions attached either to rule reduction actions or to shift actions (see 2.3). Fig 5 summarizes the architecture of the action parser.

2.3 Graphical Engine

The different functionalities of the graphical engine are: *Constraint solving*. The graphical engine is in charge of maintaining the consistency of visual objects, by applying simple constraints on *position*, *size* and *orientation*. This allows us to consider composite glyphs as first order objects.

Action lexeme synthesis. The goal is to build an event flow to feed the action parser. Users generate basic events through callbacks raised by the lower level graphical layer. The graphical engine relates these events to the internal structure of glyph types, to generate $+\tau$ and $-\tau$ events.

Visual operations. The most important functionality is to interpret instructions to handle the visual workspace. The graphical engine is an abstract stack machine that accepts a set of instructions such as CREATE-GLYPH, MOVE-TO, GET-MOUSE-POS, SET-CONSTRAINT. They homogeneously operate on typed glyphs to simplify the visual action specifications while preserving expressivity.

Animations. The graphical engine handles a list of animation scenarii. This list is periodically interpreted to animate glyphs. $(g, t_{start}, t_{end}, v : a_x)$ specifies animations, i.e. the glyph identifier, the start and end time and a vector which content depends on the attribute a_x .

To ensure the *perceptual continuity*⁴, the engine computes moderation factors for each modification step by using a function $f : [t_{start} \dots t_{end}] \rightarrow [0 \dots 1]$ such that $f(t) = (\frac{t-t_{start}}{t_{end}-t_{start}})^4$ (the exponent was tuned through experimentation). Thus the value added to g_{a_x} at each time step t is $v \times f(t)$. Fig 6 shows that the curve ensures smooth modifications. This approach is close to the *Path/Transition* paradigm [13] and the declarative approach proposed in [4] for specifying animations which puts emphasis on trajectories in the multidimensional representation space. Our approach is quite similar (although purely textual and simpler in this prototype), but introduces a notion of continuity and acceleration in animation. Of

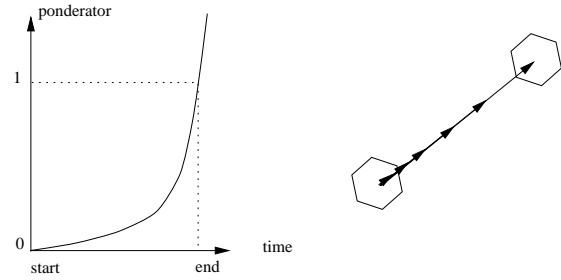


Figure 6: moderation curve for attribute modifications

course, this is not central to our framework which is not especially dedicated to algorithm animation.

3 Configuration of Distributed Application

Applications are distributed for multiple reasons:

Integration. An application integrates several services located on the provider sites (e.g. electronic commerce).

Isolation. Sensitive parts of the application should be located behind a firewall (e.g. authentication service).

Performance. Components of the application need intensive computation and are either hosted by a powerful machine or distributed (e.g. weather forecast).

Availability. Critical services are replicated and distributed on different machines (e.g. stock exchange).

However, configuring and monitoring such an application is not a simple matter. This implies (i) a logical description of the application in terms of components which interact together; (ii) a description of the underlying distributed system on which the application will run; (iii) the deployment of one to the other. Moreover, once the application is running, monitoring facilities are needed to control if the application works correctly and to act whenever it is required at the component level. (e.g. to restart a faulty service). The ideal, in term of distributed application administration is to combine both aspects (configuration and monitoring) to really offer flexibility to the application management. That is to say, to dynamically modify the application configuration while it is running and to adapt it according to information given by the monitoring.

Configuring and monitoring distributed applications is one area where we can and should make benefit of visual tools. Indeed, the complexity of the application combined with the complexity of the underlying distributed system makes it difficult to have a global view of the system. We therefore have to cope with qualitative and quantitative complexity at the same time. The former is due to the various events that may occur (e.g. failure, overloading) and the actions the manager would like to trigger (e.g. to start, or stop or migrate a component). The latter is due to the number of elements to take into consideration.

For the reasons of flexibility mentioned earlier, config-

⁴Complex modifications of the workspace may overload the user [12]

uration and monitoring should be tightly coupled through a common user interface. Thus, the quality of the visual representation of the system is essential. In particular, we must consider the scalability problem and highlight the component structure. Finally, the tool should be easy to customize to cope with different categories of users (i.e. different roles) that might be involved in the configuration and/or the monitoring of a distributed application.

4 Application to the CLF

The CLF (Coordination Language Facility) is a middleware environment for distributed coordination [2]. It provides a basic set of library tools for building *coordinators* and *resources managers*, namely software agents for, respectively, the coordination of complex tasks and the management of resources in distributed environments. It offers generic modules from which it is possible to construct dedicated services or to wrap legacy applications.

The CLF defines an extended object model which assumes that all resources managers accept two basic operations: removal or insertion of specified resources. Coordinators may realize transactions involving several resources managers. The behavior of the coordinator is specified through a rule based scripting language [2].

4.1 CLF application configuration language

In addition to this framework, we have developed a language that allows the description of a distributed application deployed on a distributed system. The idea is to associate a class to each kind of entity of the distributed system (*domain* and *host*) and the application (*application*, *proxy* and *script*). Instances of these classes define attributes describing the corresponding CLF entities and provide methods for acting on them. This low-level specification may be written by hand or automatically generated by higher level tools such as the visual configuration tool presented here.

The **distributed system** is described through *domains* containing *hosts*. A *domain* is associated with a local area network sharing CLF libraries through a file system (e.g. NFS). A *host* object corresponds roughly to a workstation on which a component of the application can run.

The **distributed application** is represented by an *application* object describing the data shared by the different CLF objects which compose it. For each CLF object there is a corresponding a *proxy* object. The attributes of the *proxy* describe on one hand the CLF object role in the application (e.g. its type: coordinator, name server, user defined component) and on the other hand its location in the distributed system (e.g. on which host it will run). A *proxy* also provides a set of methods that allows direct interaction with the CLF object (e.g. *stop*, *start*, *inspect*, used for monitoring purposes). A set of CLF rules is described through a *script* object whose attributes are the source code

and a reference to the coordinator *proxy*. A *script* also exports the method *run* whose effect is to make the code interpreted by the associated coordinator.

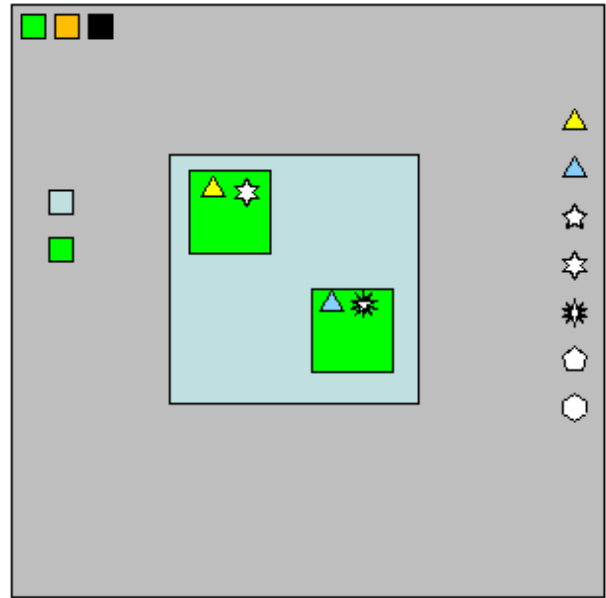


Figure 7: A hardware configuration. *The central square (domain) contains two smaller squares (hosts), decorated with operating system and processor types (small triangle and star shaped glyphs). The user can inquire for textual information by pointing the mouse on the glyphs*

The **deployment** consists in specifying the *host* attribute of each *proxy* and to invoke the *start* method. All these concepts and their underlying constraints are shown and manipulated through the tool. Its workspace (see Fig 8) reflects the three main tasks of configuration editing: specification of the logical structure of the application (upper left grey square), specification of the physical structure of the distributed platform (upper right grey square) and specification of the deployment (lower grey square). In each context, operations are performed by direct manipulation (with a mouse) on small glyphs (the three squares lined on the upper sides allow us to *load*, *save*, and *clean* the contexts by pointing/clicking actions). To specify the deployment, objects from upper contexts are dragged into the lower context, according to positional constraints that reflect the semantic constraints: the *proxy* of a CLF object must be put on a *host*, but neither on a *domain*, nor on another application object.

Before launching the interaction, part of the workspace is drawn by executing the following code:

```
CREATE application;PUSH;SET-POS(150,150);SET-SIZE200
CREATE platform;PUSH;SET-POS (250,150);SET-SIZE 200
CREATE deployment;PUSH;SET-POS (200,300);SET-SIZE 300
```

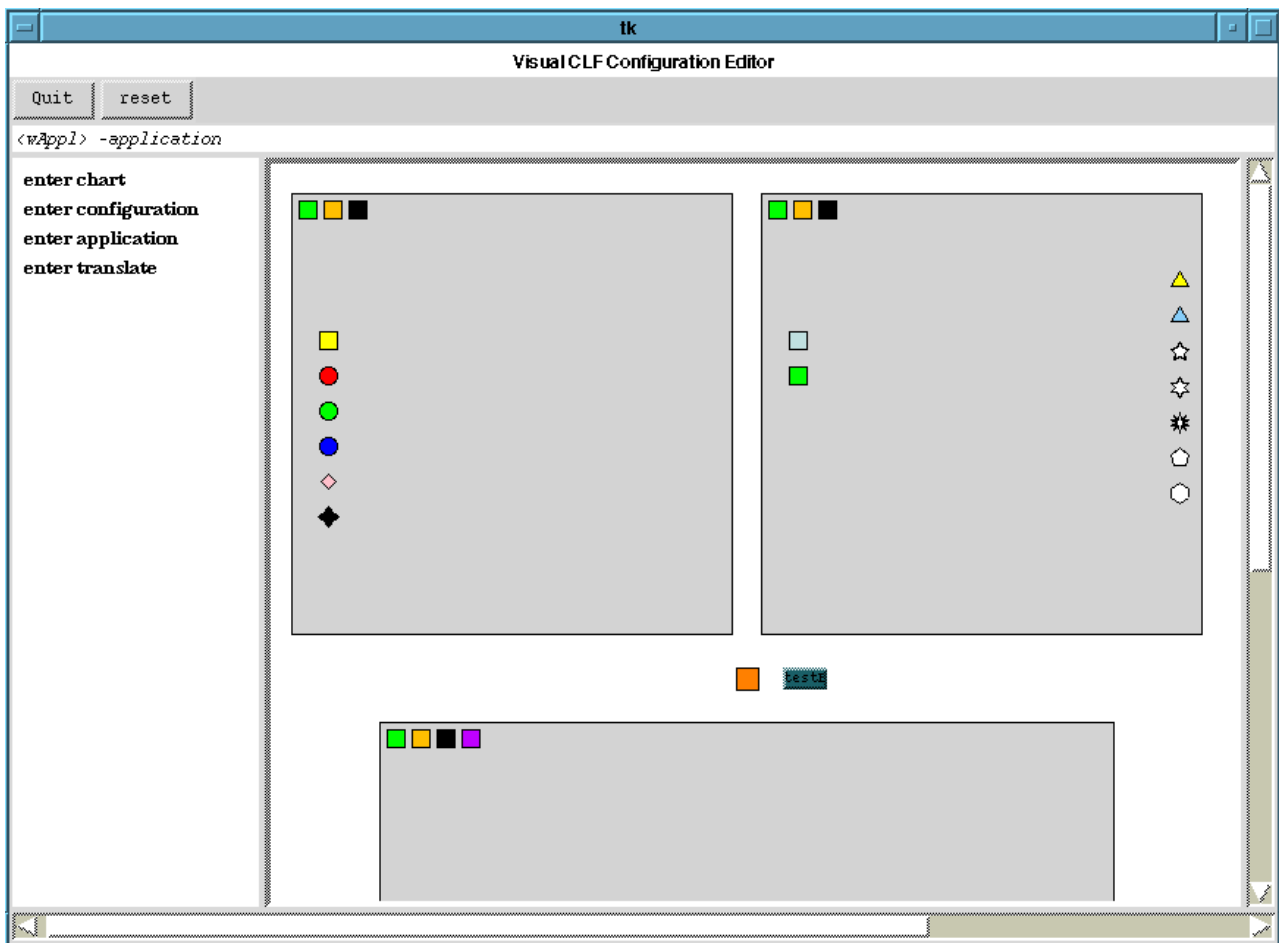


Figure 8: Overview of the workspace

The grammar rules expressing these three main tasks are⁵:

```
All      → Appli All
All      → Hard All
All      → Deployment All
Appli    → +application Doappl -application
Hard     → +hardware Dohard -hardware
Deployment → +deployment Domapp -deployment
```

The user specifies the hardware platform (see Fig 7) through the following actions: (i) *domain(s)* creation, (ii) *host(s)* creation on *domains*, (iii) *hosts* attribute definition. Visually, *domains* are created by pointing a blue square (of visual type *domMaker*), pressing the mouse button 1 (a new blue square pops up beside the pointer), pointing a target position in the context and finally by releasing the button (the square goes to the position, with a growing size).

```
Dohard   → CreateDomain
CreateDomain → +domMaker press1 -domMaker release1
```

The associated shift actions for visual feed back are:

```
*.+domMaker .press1 ⇒ CREATE domain/PUSH/GET-M-POS/
PUSH (20,20)/ADD /SET-POS/STICK-M
*.+domMaker .press1.* .release1 ⇒ UNSTICK-M/ PUSH/ GET-POS/
GET-M-POS/SUB /GET-TIME/PUSH/PUSH 0.2/ADD /PUSH 'pos'/
ANIMATE /...
```

⁵non terminals begin with an uppercase

The first rule creates a new glyph with type *domain*, and constraints its position to track the mouse motion. The second rule unsticks the object, and launches a double smooth animation: a motion to the target position pointed by the mouse at release time, and a growth of the shape. *Hosts* are created following the same operations, but must be placed over an existing *domain*. The corresponding rules are:

```
Dohard   → CreateSite
CreateSite → +siteMaker press1 -siteMaker+domain release1 -domain
and an additional action is specified:
*.+siteMaker .press1.+domain ⇒ GET-POINTED/PUSH /GET-SIZE/
PUSH 0.4/ ADD /SET-SIZE
```

which dilates the entered domain each time a new site is added. This kind of behavior is also a response to the scalability problem: the size of containers is allowed to grow incrementally, as far as they stay within a limit defined by the environment. Beyond this limit, all contained objects are incrementally reduced in order to keep constant the ratio free area/used area (this is a guarantee that new objects can be introduced). Special facilities such as fish-eye behavior

(pointed objects are temporarily dilated) allow the manipulation of small sized items (in 3D workspaces, this can be obtained by zooming). This is an example which shows how animations and polymorphism (all objects can be resized in the same way) can help to enlarge the workspace size. Host attributes (operating system and processor type) are specified by dragging colored small triangles and star shaped icons onto the hosts. Other attributes are specified textually through specialized editing forms.

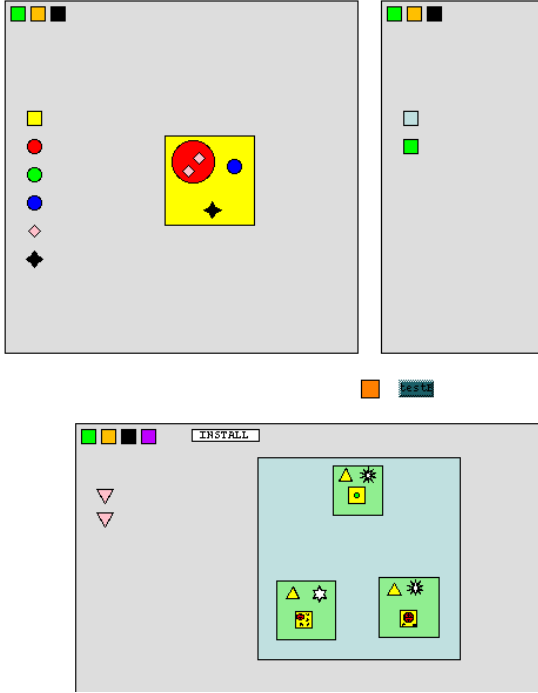


Figure 9: A compiled deployment (bottom context). The two triangles on the left side are warnings raised by the compiler. They may be clicked on to launch a animation showing the concerned glyphs

The logical application structure is specified in a similar way, with richer combination possibilities that reflect the richer semantics of the specification. Glyphs lined-up vertically on the left side of the application editing context (Fig 8) represent the following operators: object grouping; makers of basic object *proxies* (e.g. *coordinator*, *nameServer*), generic *proxy* maker and *script* maker. Fig 9 shows such a specification in the upper left context. Many interaction facilities are provided by the tool (such as object browsing and automatic popping names) which are not described here (the global action grammar contains about 100 rules). Moreover, some semantic verifications are performed online. For instance, a *script* cannot be dragged into a *proxy* different from a coordinator.

The context for defining the *deployment* allows the as-

sociation of the *proxy* of objects to *hosts*. This is also performed through dragging operations. Once the *deployment* has been completed it is possible to *compile* the specification, and then *install* the application. The compilation is done by a visual syntax analysis, followed by a verification and generation phase. If errors or warnings are detected, they are represented by small specific glyphs (lined up vertically on the left). The user can click on errors or warnings to display the associated text which explains the problem, and to highlight the concerned glyph. Fig 9 shows the successful compilation of a deployment specification (lower context) with two warnings (triangles on the left). Small glyphs included in *hosts* have been previously dragged by the user from the application context, and reduced when entered into *hosts*. Monitoring facilities will be offered in the next version of the prototype through an asynchronous interpretation of an external event flow. Indeed, the graphical engine is able to process several input channels at the same time, and the workspace can therefore be animated asynchronously to reflect external modifications. A representational function must also be defined to display a given set of information to the workspace, while maintaining the consistency of the representation (the user must still be able to act on visual objects). For instance, active *proxies* can be shown as rotating, as opposed to inactive, fixed proxies. Related commands such as launching or stopping a proxy should then be introduced according to the representation.

The visual parser is not yet automatically generated but can be specified by the following positional grammar (see [6] for the formalism):

A	\rightarrow	D besides A	$\Delta(A, \text{Domain})$
A	\rightarrow	D	$\Delta(A, D)$
D	\rightarrow	domain contains N	$\Delta(D, \text{domain})$
N	\rightarrow	Host besides N	$\Delta(N, \text{Host})$
N	\rightarrow	Host	$\Delta(N, \text{Host})$
$Host$	\rightarrow	host contains I	$\Delta(Host, host)$
I	\rightarrow	OStype besides CPUtype besides It	$\Delta(I, \text{OStype})$
It	\rightarrow	Item besides It	$\Delta(It, \text{Item})$
It	\rightarrow	Item	$\Delta(It, \text{Item})$
Item	\rightarrow	Coord	$\Delta(\text{Item}, \text{Coord})$
Item	\rightarrow	nameServer	$\Delta(\text{Item}, \text{nameServer})$
Item	\rightarrow	proxy	$\Delta(\text{Item}, \text{proxy})$
Coord	\rightarrow	Coord contains CLF	$\Delta(\text{Coord}, \text{CLF})$
CLF	\rightarrow	script besides CLF	$\Delta(\text{CLF}, \text{script})$
CLF	\rightarrow	script	$\Delta(\text{CLF}, \text{script})$

The Δ functions determine the transfer equations of attributes from the right side to the left side of the rules. *Contains* and *Besides* are spatial relationships such that:

$$X \text{ Contains } Y \Leftrightarrow \begin{cases} \|Y_{pos} - X_{pos}\| < BSR(X) \\ \|Y_{pos} - X_{pos}\| < BSR(Y) \end{cases}$$

where $BSR(X)$ gives the radius of the X bounding sphere

$$X \text{ Besides } Y \Leftrightarrow \begin{cases} \|Y_{pos} - X_{pos}\| > BSR(X) \\ \|Y_{pos} - X_{pos}\| > BSR(Y) \end{cases}$$

5 Conclusion and future work

We have described an approach based on a triple structuration of the workspace through a visual type system, user action grammars and visual parsing. The visual type system aims at providing expressivity and polymorphism,

in order to simplify the specification of visual operations. Other works propose type systems strongly inspired by functional languages, especially in data-flow visual languages, but poorly related to the visual representation. Systems such as Form/3 [3] propose higher level and visual type systems, but do not take full advantage of simplicity and geometric polymorphism. Some work should be done in order to design a higher level graphical language allowing formal verifications based on the properties of the type system. A promising solution may be related to spatial logic and its associated semantics ([8]), since spatial constraints between glyphs can be easily modeled through bounding circle abstractions. Action parsers using LR(1) grammar and reactive semantic actions linked to shifting phases have not to our knowledge been proposed. The latter combines the expressive power of LR(1) grammars, and their well known formalism to a simple basic event set. The error management appeared particularly generic, with a surprising flexibility, unexpected from a grammar based approach. Formal verification of semantic rules regarding the context-free rules will be investigated in order to detect inconsistencies (e.g. rules that could not be triggered given the associated grammar). The visual parsing is not innovative *per se*, but well integrated into an interactive system. An interesting point is the relationship between action parsing and visual parsing: a large part of visual syntax constraints is handled by the action parsing before the visual parsing. It thus behaves in a similar way to a semantic verification layer with respect to a syntax checking layer. We will investigate the possibilities of binding the visual parsing and the action parsing in a stronger way, and this through an incremental approach of positional parsing.

As a conclusion, this synthesis of several advanced technologies offers promising perspectives that have been partially experimented within a prototype: (i) the scalability problem, which has been addressed through object size management facilities, (ii) the explicit rendering of the interaction syntax through high level grammar rules (increases maintainability and provides low-cost *as soon as possible* error management), and (iii) simplification of the verification and generation of code. The scaling-up problem has been identified as a hot topic for VLs[3], and our solution is similar to the response of Citrin et al.[5], except that their approach relies on a stronger semantic hypothesis, and our sign system appear to be richer and more generic. Finally, from the user point of view, the perception of the underlying semantics seems easier, because contextual constraints are directly represented and manipulated.

References

- [1] A. Aho, R. Sethi, and J. Ullman. *Compilers*. Addison-Wesley, 1986.
- [2] J.-M. Andreoli, F. Pacull, D. Pagani, and R. Pareschi. Multiparty negotiation for dynamic distributed object services. *Journal of Science of Computer Programming*, to appear.
- [3] M. M. Burnett, M. J. Baker, C. Bohus, P. Carlson, S. Yang, and P. van Zee. Scaling up visual programming languages. *IEEE Computer*, 28(3):45–54, March 1995.
- [4] P. Carlson, M. Burnett, and J. Cadiz. A seamless integration of algorithm animation into a visual programming language. In *Proceedings of AVI'96: International Workshop on Advanced Visual Interfaces*. ACM, May 27-29 1996.
- [5] W. Citrin, R. Hall, and B. Zorn. Addressing the scalability problem in visual programming. Technical report, University of Colorado, Dpt of computer Science, April 1995.
- [6] G. Costagliola, G. Tortora, S. Orefice, and A. De Lucia. Automatic generation of visual programming environments. *IEEE Computer*, 28(3):56–66, March 1995.
- [7] F. Ferrucci, G. Tortora, M. Tucci, and G. Vitiello. Relation grammars: a grammatical model for a high-level specification of visual languages. In *Proceedings of AVI'96: International Workshop on Theory of Visual Languages*. ACM, May 27-29 1996.
- [8] V. Haarslev. Formal semantics of visual languages using spatial reasoning. In *Proceedings of VL'95*. IEEE symposium on Visual Languages, 1995.
- [9] H. Hartson and P. Gray. Temporal aspects of task in the user action notation. *Human-Computer Interaction*, 7:1–45, 1992.
- [10] M. Mezzanote and F. Paterno. Including time in the notion of interactor. *SIGCHI bulletin*, 28(2):57–61, April 1996.
- [11] J. Reckers and A. Shurr. A parsing algorithm for context-sensitive graph grammars. Technical report, Department of Computer Science, Leiden University, 1995.
- [12] G. Robertson, S. Card, and J. Mackinlay. Information visualization using 3d interactive animation. *Communication of the ACM*, 36(4):57–71, April 1993.
- [13] J. T. Stasko. The path transition paradigm: A practical methodology for adding animation to program interfaces. *Journal of Visual Languages and Computing*, 1(3):213–236, 1990.
- [14] S. Uskudarh. Generating visual editors for formally specified languages. In *Proceedings of the 10th International Symposium on Visual Languages*. IEEE, 1994.
- [15] J.Y. Vion-Dury and M. Santana. Virtual images: Interactive visualization of distributed object-oriented systems. In *Proceedings of OOPSLA'94*, volume 29, October 1994.
- [16] K. Wittenburg. *Recent Advances in Parsing Technologies*, chapter Predictive Parsing for Unordered Relational Languages. Kluwer, 1993.
- [17] K. Wittenburg and L. Weitzman. Relational grammars: Theory and practice in a visual language interface for process modeling. In *AVI'96 International Workshop on Theory of Visual Languages*, May 1996.