

# Toward a Language–Independent Legacy Software Visualizer: The Case of Procedural Languages<sup>1</sup>

Arturo J. Sánchez-Ruíz<sup>(\*)</sup>, Jesús Ruiz Coello<sup>(\*)</sup>, and Ephraim P. Glinert<sup>(†)</sup>

<sup>(\*)</sup>Laboratorio de Construcción de Herramientas Automáticas (AuTooLab)  
Centro de Ingeniería de Software y Sistemas (ISYS)  
Facultad de Ciencias, Universidad Central de Venezuela  
APARTADO 47642, CARACAS 1041–A  
VENEZUELA

E–mail: [arsanche@reacciun.ve](mailto:arsanche@reacciun.ve), {[asanchez](mailto:asanchez@orquidea.ciens.ucv.ve), [jruiz](mailto:jruiz@orquidea.ciens.ucv.ve)}@[orquidea.ciens.ucv.ve](http://anubis.ciens.ucv.ve/~asanchez/autoolab.html)  
<http://anubis.ciens.ucv.ve/~asanchez/autoolab.html>

<sup>(†)</sup> Computer Science Department  
Rensselaer Polytechnic Institute  
Troy, NY 12180  
USA

E–mail: [glinert@cs.rpi.edu](mailto:glinert@cs.rpi.edu)  
<http://www.cs.rpi.edu/~glinert>

## Abstract

This paper deals with our approach to visualize legacy software resources to be reused in a multilanguage programming environment. Under this class of environments, a solution can be thought of as a collage assembled from existing components, which are implemented in various imperative object–oriented, or procedure–oriented programming languages. Our approach uses reverse engineering techniques to extract information from the resource which is then expressed on a language–independent platform. Using this language–independent representation, the visualizer can show different views associated with the resource, such as (but not limited to) animated diagrams which depict the structure of objects, animated representation of member function and/or procedure functionality, animation of some member functions and/or procedures which are automatically generated for achieving multilanguage reusability. We present VROOM, a prototype of a legacy software visualizer for procedural languages.

**Keywords:** Software Visualization, Legacy Software, Multilanguage Reusability, Reverse Engineering, Reengineering for Reuse.

---

<sup>1</sup>This work was supported, in part, by *Consejo de Desarrollo Científico y Humanístico de la Universidad Central de Venezuela* under contract CDCH–03–13–348495.

# 1 Introduction

The term legacy software visualization refers to the process of representing properties associated with existing software resources by means of visual metaphors. Software visualizers are tools aimed at helping the user to elicit the resource's semantic behavior, by using visual renderings of its properties which are syntactically defined. As an example, consider a fragment of code written in a procedural programming language which specifies a binary tree of integers. This syntactic representation would probably use concepts such as nodes, pointers, and integers, and some might find it very cryptic even for such a well-known concept. Suppose we build an animation which shows a graphical representation of the node concept, such that it gets expanded when clicked upon showing that it internally contains three fields, an integer and two fields pointing to the same type of structure. We can continue clicking on this representation and at some point we will end up with a representation which is by far richer than the planar textual representation. Yet, this representation can be computed from the original code.

Our main goal is to build automatic tools which can be used to assist programmers in the process of building software systems by composing existing (software) resources. Since these resources are originally written in various programming languages (hence supporting various programming paradigms), it would be very useful if each of these parts were used *as is*, giving the so composed system the structure of an heterogeneous collage, an approach which we have called multilanguage reusability [4, 7].

Before using an existing resource, the programmer must have some level of understanding of the semantics associated with it. It is then when legacy software visualization tools come into play in order to bridge the gap between the syntactic representation of these resources and their semantic behavior. In this paper we present our approach toward the construction of language-independent legacy software visualizers. The independence with respect to the language refers to the ability these visualizers must possess to be directly used within the

framework of an ample family of programming languages. In our case, we are interested in imperative object-oriented and procedure-oriented languages.

To visualize a legacy software resource our approach first applies reverse engineering techniques in order to extract relevant structural information which is expressed in a language-independent formalism. We shall cover the details associated to this phase in Section 2. Once this information has been extracted, it can be used for either wrapping the resource so it can be reused from any of the programming languages supported by the underlying environment, or for visualizing it. The visualization issues are discussed in Section 3. In Section 4 we shall show how we applied this methodology in the construction of a prototype of a visualizer for procedural languages, which we call `VROOM`. Finally, in Sections 5 and 6, we mention some related contributions which have appeared in the literature (and in the cyberspace!), and comment on different avenues to be explored toward the construction of completely automated language-independent legacy software visualizers.

## 2 Reverse Engineering Legacy Software

The first step toward the visualization of legacy software resources is the extracting of information which can be used to build different views associated with them. In order to deal with software heterogeneity, we express this information on a language-independent platform with two components, namely data and control. The data component is expressed by using a formalism we call CTS, which stands for Common Type System. The control component is expressed by means of a language-neutral intermediate representation we call CIR, which stands for Common Intermediate Representation.

CTS provides concepts such as primitive types (e.g. integers, reals, characters, bool, enumerations), type constructors (e.g. arrays, records, unions, pointers) and classes with inheritance and genericity. Conceptually, we see CTS as a layered type system such that (see Fig. 1):

- Primitive types are atomic entities.

- Type constructors allow the definition of data structures.
- Classes allow the definition of higher-order objects, which have a representation implemented by means of data structures, and an interface containing the class member functions.
- Inheritance and genericity allow the construction of new classes from existing classes.

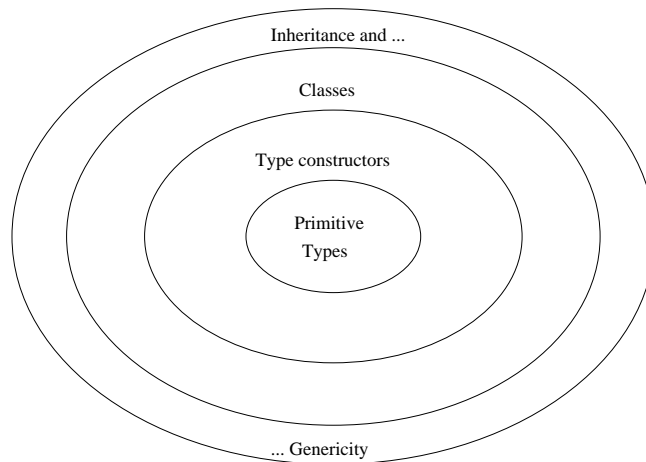


Figure 1: The layered structure of CTS.

In Fig. 2 we show an example of a CTS specification associated with a binary tree implemented in Pascal. The specification indicates the implementation language (`language: declaration`), the resource name (`where: declaration`), and the types associated with it. This resource has two types, namely `btree` which is a class<sup>2</sup>, represented by the concrete type (data structure) `node` and with the shown interface. For instance, member function `Left` takes a `btree` and returns a `btree`. Type `node`, on the other hand, is a record with three fields (named `info`, `left`, and `right`, respectively) such that the `left` field is a pointer to `node`. CTS is based on the concept of *type models*, which provides a formal framework on which seemingly different imperative object-oriented or procedure-oriented programming languages can coexist in harmony. The concept models the type construction process associated with this family

---

<sup>2</sup>In CTS classes are types which have, at least, an interface and a representation (they could also have parameters and/or ancestors). Therefore, CTS does not need to introduce a keyword such as `class`.

of programming languages in such a way that we can conceive of entities such as objects and classes in languages which lack this machinery, thus shortening the apparent distance between object-oriented and procedure-oriented imperative languages. For more details on CTS and type models, we refer the reader to [4, 6, 7].

```
language: Pascal
where: btree

type btree =
  representation:
    *node
  interface:
    Left (btree): btree;
    Right (btree): btree;
    Print (btree): void;
    IsEmpty (btree): int1;
    Root (btree): int1
eod;

type node = {
  info: int1;
  left: *node;
  right: *node
}
```

Figure 2: Example of a CTS specification of a Pascal binary tree.

Our Common Intermediate Representation (CIR) allows us to express several concepts such as syntax graphs, type dependency graphs and signatures. A syntax graph captures the structure of a CTS type, as can be seen in Fig. 3. A type dependency graph represents the relation “*is defined in terms of*” among CTS types in a software resource. An example of such a graph, involving types `btree` and `node`, is shown in Fig. 4 (a). Finally, a signature is a tuple, computed for each operation to be generated by the reengineering phase, which contains all information needed to produce the code of those operations in the resource’s native language. The main result of this reengineering phase is what we call a *wrapper class*,

which contains the original resource operations plus computed augmenting operations such as constructors, selectors, modifiers, equality test functions, copy procedures and operations for transparent pointer manipulation. In Fig. 4 (b) we show the signature associated with a generated constructor operation for `btree`.

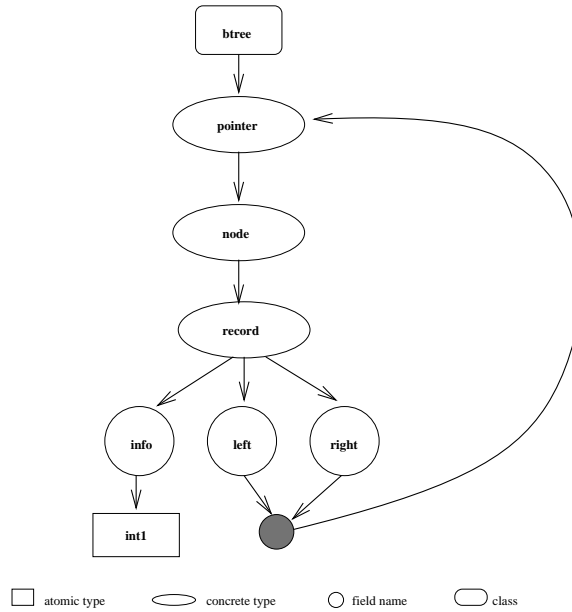


Figure 3: The syntax graph associated with CTS types `btree` and `node`.

Our reverse engineering approach takes a file containing the resource definition (in its native language) and first does some analysis to compute the CTS specification of involved types. This specification is the input to the following phase which generates the CIR specification, which in turn can be the input both to a phase for wrapping the resource (so it could be reused from any of the languages supported by the underlying programming environment), or to a phase for visualizing the resource. This reverse engineering process is depicted in Fig. 5. We refer the reader to [4, 5, 7] for details about the reengineering of legacy code for ulterior reuse under our ROOM<sup>3</sup> methodology. These references also cover the details associated with the computation of CTS and CIR specifications. Next section will deal with the use of CIR specifications to visualize its associated software resource.

<sup>3</sup>ROOM stands for Reverse Object-Oriented Multilanguage reusability.

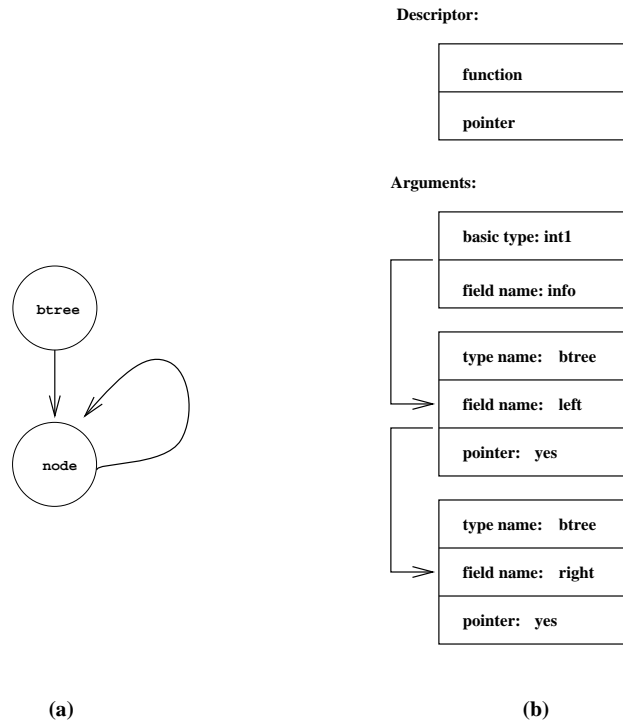


Figure 4: (a) Type dependency graph involving CTS types `btree` and `node`. (b) Signature associated with a constructor for `btree`.

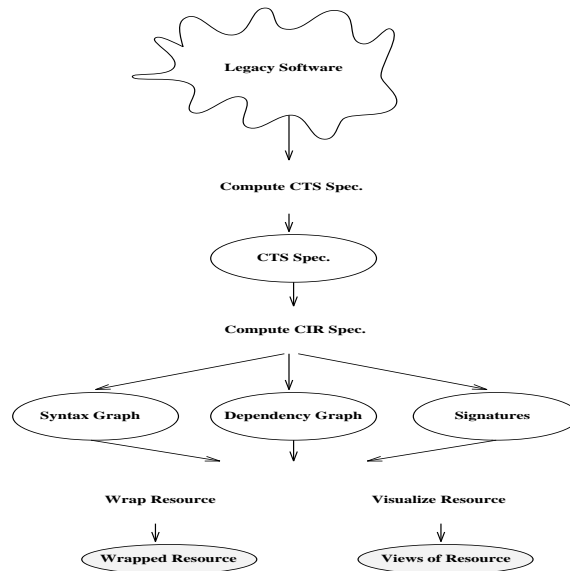


Figure 5: Our approach to reverse engineer legacy code.

### 3 A Language–Independent Visualizer

Our language-independent approach to legacy software visualization is based on the information provided by the specifications in CTS and CIR computed by the reverse engineering phase described in Section 2. Using this information it is possible to build a wide spectrum of views (which can in turn be rendered in many different ways), among which we can mention:

- Animated diagrams which depict the structure of data entities in the resource.
- Animated representation of member function/procedure functionality.
- Animated representation of those member functions/procedures computed by the reengineering phase.
- Hypertext entity–relationship diagrams, for entities such as procedures, member functions, data members, etc, and relationships such as “is used by”, “derives from”, “is called by”, “is a parameter of”, etc.

In this section we will briefly discuss issues associated with the construction of some of these views. Then, in Section 4, we will illustrate the way they are implemented in our prototype VROOM.

#### 3.1 Depicting the Structure of Data Entities

In order to depict the structure of data entities we need to define what we call a visual dictionary of CTS types, which associates a graphical representation with each type. For the sake of fixing some ideas at this point we give two examples, namely an arrow associated with CTS pointers, and a pentagon for CTS classes with clickable slots representing their interface, representation, ancestors, and parameters, respectively.

These visual representations should have attributes which help describe various concepts associated with data entities. So we can associate a range of colors with the depth of a data

structure, the depth of a geometric figure with the fact that it holds a structured object (3D-looking figure) or an atomic object (flat-looking figure). Other attributes we have used to build visualizations are “self-animated” and “interactively-animated”. A diagram is self-animated if it expands itself showing the internals associated with each component in the diagram. Since a diagram could be associated with a recursively defined CTS type, we could be dealing with a conceptually infinite process which is made finite by imposing geometric restrictions on the self-expanding diagram, such as window boundaries and non-interference with already drawn elements in the diagram. So, if an element cannot be expanded because it would interfere with an already drawn part of the representation, it simply turns black. These animations are combined with a dialog using the “tape deck metaphor” (which shows buttons for playback, stop and pause), and a slide button to control the animation speed. Interactively-animated diagrams, on the other hand, wait for the user to click on an expandable element in the diagram to show its internals. So these diagrams offer exactly the same functionality than their self-expandable counterparts only that, in this case, the user has control on the way the diagram gets expanded.

To build these diagrams, the visualizer uses the syntax graphs associated with involved CTS types. For self-animated diagrams, the graphs are traversed in level-order<sup>4</sup> (while there are elements which could be expanded) drawing, for each element, the corresponding visual representation. If the object is expandable then the first level of expansion is shown (unless there are geometric restrictions to do so, in which case the object turns black). Interactively-animated diagrams need a more elaborated algorithm because we need to keep information about generated elements, because it is not known in advance where the user will click. In Section 4 we will show still frames associated with the visual dictionary used by VROOM, and with an animation which shows the structure of a binary tree implemented in Pascal.

---

<sup>4</sup>That is to say, top-to-bottom and left-to-right.

## 3.2 Visualizing the Behavior of Procedures

To visualize the behavior of procedures which belong to a legacy software resource, we have used a high-level approach and a low-level approach. The high-level approach is based on the “black-box metaphor”, i.e. a procedure (or a function for that matter) is visualized as a box which receives entities and outputs entities. So, to visualize any procedure/function in the resource, either those which originally came with it or those computed by the reengineering phase, an animation is built which shows a dark box which opens its top to receive the graphical representation of its inputs, closes for a while, and then opens its top again from which its output emerges. This animation can be directly built from the signatures in the CIR resource representation.

Our low-level approach to visualize procedures/functions in the resource consists of showing the way they work to produce their outputs from their inputs. Currently we are focusing on generated procedures/functions only, for they are computed from their signatures, i.e. their code in the native language is generated from their signatures (which let us recall are language-independent representations). If we wanted to produce animations associated with native procedures/functions we would have to build interpreters for each language our environment deals with and the language parameterization would therefore be compromised. Low-level animation of procedures/functions are built by using the CIR representation associated with them. For a given procedure/function, the algorithm which produces the animation mimics the code generation algorithm used to build the augmenting functions (c.f. [4, 5] for details on the way the native code for augmenting functions is generated), producing views associated with involved operations (e.g. assignments, equality checks, component extraction) by using the representations from the visual dictionary. Next section will show an example of how VROOM visualizes the input-output behavior of procedures/functions.

## 4 VROOM: A Prototype

We built VROOM with a twofold goal in mind, namely to be able to visualize multilanguage software resources to be reused from any other language supported by our environment, and to have a visual user interface for our prototype environment called ROOM [4, 7]. Since VROOM makes heavy use of animation to interact with its user, it is very difficult to faithfully render all its functionality in a static medium such as this. So we invite the reader to try to get a glimpse of VROOM by downloading it from the web at:

*<http://anubis.ciens.ucv.ve/~asanchez/legvis.html>*

At that site we also have a version of this paper which shows the still frames that will be shown in this section in full color.

We start by showing VROOM's main window in Fig. 6. This window has the classical **File–Options–Help** buttons, such that **File** allows the user to load, visualize or reuse a resource, **Options** allows the user to change the look–and–feel of VROOM, and **Help** allows the user to get help with the visual dictionary (plus the classical “about” button). When the mouse cursor gets on top of any of the five icons, they react by textually showing what they can be used for. Their meaning, from left to right, is: exit, load a resource, reuse a resource, visualize a resource, and help. The visual dictionary used by VROOM is shown in Fig. 7. For lack of space, we are unable to show the way each CTS type is represented, so we invite the reader to use the web for this.

Let us suppose the user wants to visualize a software resource. She has to go through the following steps: (a) load the file containing the resource implementation in its native language, (b) possibly edit it, (c) ask VROOM to visualize it. If the resource's CIR representation was previously computed, the user can go ahead to directly visualize it. VROOM then offers the user the option of visualizing the resource's CTS classes and other types in it. At this point the user also has the opportunity of indicating VROOM whether she wants the diagrams to be self–expanding or interactively–expanded. If the resource in question is the one shown in



Figure 6: vROOM's main window.

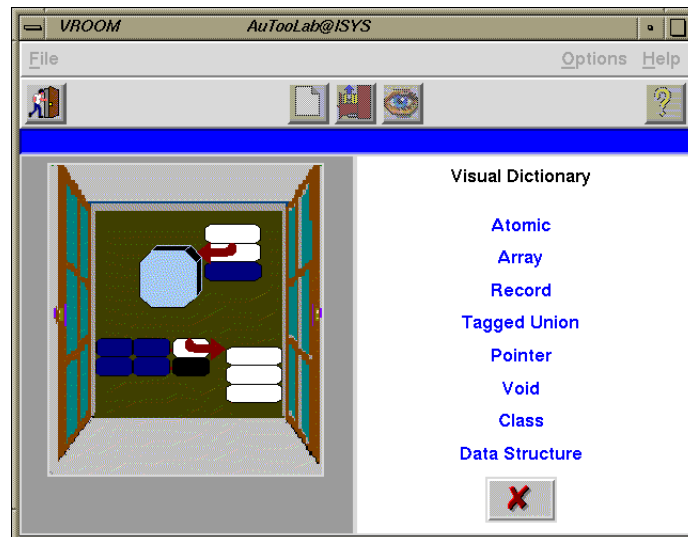


Figure 7: Visual dictionary used by vROOM.

Fig. 2, and the user asks to visualize CTS class `btree`, then VROOM will show its graphical representation, which consists on a blue pentagon with four clickable slots, namely the ones labeled BTREE, O, H, and R. Clicking on BTREE will show the user the CTS specification of this resource. Clicking on R will show the user the resource's representation. In Fig. 8 we show the final frame associated with the self-expanding visualization of `btree`'s representation. If

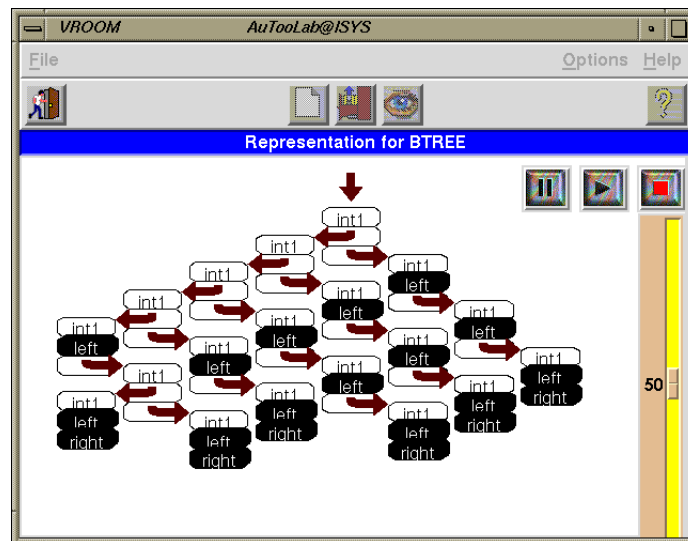


Figure 8: Final frame of a self-expanding diagram showing the structure of `btree`'s representation.

the user clicks on O she will get a list of all operations associated with `btree` (original and generated), clicking on any of the original operations will show the user its code. On the other hand, clicking on any of the generated operations will show the user a high-level animation of this operation. An intermediate frame of this animation is shown in Fig. 9.

VROOM also offers the possibility of reusing the resource by computing a wrapper class as explained above. In this case the user needs to build a client program, in the language of her choice, from which she can reuse all the resources wrapped by ROOM. This program, in the client language extended with references to the resources, is translated into a functionally equivalent program without extensions. Finally, the ROOM environment invokes the appropriate compilers so an executable can be generated.

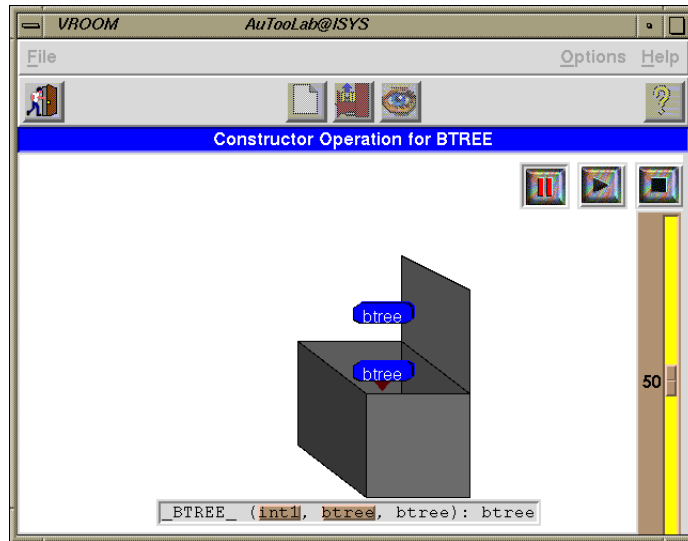


Figure 9: Intermediate frame of an animated visualization of a constructor operation for `btree`.

## 5 Related Work

In [3] the reader can find an impressive sample of reverse engineering tools that includes links to the involved web sites, some of which have downloadable demos and/or prototypes. In this section we discuss two important issues which we believe are either improvable or not covered at all by these systems. Since all of them are referenced in [3] we do not include their references here. Two additional sources with references to software visualization tools are [1, 2]. It is worth mentioning that a thorough discussion involving all systems and all relevant issues is well beyond the page limit of this paper.

*The use of animation can still be further exploited.* For example, if we consider the graphical representation of a hash table used by *Imagix 4D*<sup>5</sup>, the recursive nature of each collision list is not directly evoked from it. On the other hand, if a self-animated diagram like the ones implemented in VROOM would have been used, this property would have been made more evident, simply because the animation just follows the pattern given by the data structure. Another system, *Rigi*, which includes in its demo distribution an example of a C

<sup>5</sup>Demo available from <http://www.imagix.com>

code implementing a single-linked list, shows visualizations which don't suggest the intuitive idea associated with this kind of data structure.

*Coexistence of multiple languages on a common platform.* From what we have seen, it seems that none of the systems listed in [3], were originally aimed at visualizing multilanguage systems. This has, at least, two implications. The first one is that each language is treated separately, i.e. it is not possible to concurrently visualize several languages. The second one is that the language processors used by the visualizers (parsers, translators, etc) need to be generated for each language. We must mention, however, that *Refinery's* approach seems to go toward the direction of automatic generation of parsers, but this approach does not seem to mix object-oriented with procedure-oriented languages. Additionally, the *REDO Project* uses a common platform for data integration, similar in spirit to our CTS, but they do not mix object-oriented with procedure-oriented languages.

## 6 Summary and Future Work

We have presented our approach toward building language-independent legacy software visualizers that can be used in a multilanguage programming environment. This approach starts by applying reverse engineer techniques to extract all relevant information from the resource, which is expressed in our language-independent formalism CTS, CIR. This language-independent specifications can be used to either reengineer the legacy resource (for ulterior multilanguage reuse) or to visualize it. The visualization techniques use animation to show properties such as structure of data entities, member function/procedure functionality, member function/procedure dynamic behavior, etc. We presented VROOM a prototype of legacy software visualizer for procedural languages.

Currently we are working on building multilanguage language processor generators to completely automate the way a particular language is mapped into the language-independent formalism. The goal is to be able to automatically generate visualizations for an ample

family of programming languages so multilanguage resources can be visualized using a consistent cross-language visual metaphor. We want this generators to produce their parsers from examples of phrases.

Since there are many animation systems already built<sup>6</sup>, we would also like to parameterize the visualizers with respect to the animation system used. We are also working on adding features for the user to define the visual dictionary associated with data entities and the way the animated diagrams should be generated. Lastly, we would like produce *Java*<sup>7</sup> applets which can be used to build these class of visualizers, and to explore the impact of virtual reality in the visualization of legacy software.

## References

- [1] Arne Frick. Bibliography on Software Visualization. URL <http://i44s11.info.uni-karlsruhe.de/~frick/SoftVis/bibliography.html#VL:92:Proceedings>, 1997.
- [2] Luis Gómez. Referencias sobre Visualización de Programas. URL <http://avellano.datsi.fi.upm.es/~lgomez/seminarios/PV.html>, 1997.
- [3] Hausi A. Müller. Understanding Software Systems Using Reverse Engineering Technologies Research and Practice. URL <http://www.rigi.csc.uvic.ca/UVicRevTut/UVicRevTut.html>, 1997.
- [4] Arturo J. Sánchez Ruíz. *On Automatic Approaches to Multi-Language Programming via Code Reusability*. Phd dissertation, Computer Science Department, Rensselaer Polytechnic Institute, Troy, NY, USA, 1995. (<http://anubis.ciens.ucv.ve/~asanchez/recent.html>).
- [5] Arturo J. Sánchez Ruíz. Reengineering legacy code: A case study. In *Proceedings of ISAS '96: the International Conference on System Analysis and Synthesis*, pages

---

<sup>6</sup>A good starting point is <http://www.research.digital.com/SRC/zeus/home.html>

<sup>7</sup><http://sunsite.berkeley.edu/java-corner>

283–290, Orlando, USA, 1996. International Institute of Informatics and Systemics.  
(<http://anubis.ciens.ucv.ve/~asanchez/recent.html>).

[6] Arturo J. Sánchez Ruíz. Type models: Toward unifying concepts for language interoperability. In *Proceedings of PANEL'96: the XXII Latin American Conference on Informatics*, pages 541–552, Bogotá, Colombia, 1996. CLEI.  
(<http://anubis.ciens.ucv.ve/~asanchez/recent.html>).

[7] Arturo J. Sánchez Ruíz and Ephraim P. Glinert. ROOM: A reverse object-oriented approach to multi-language reusability. Technical Report 95–17, Computer Science Department, Rensselaer Polytechnic Institute, Troy, NY, USA, 1995.  
(<http://anubis.ciens.ucv.ve/~asanchez/recent.html>).