A Formal Specification of a Visual Language Editor

Jeannette M. Wing   Amy Moormann Zaremski

School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213

Abstract

This paper presents a non-trivial case study on the use of the Larch specification languages to describe the Miró visual languages and graphical editor. In addition to excerpts from the specification, we discuss properties of Miró provable from the specification, limitations of Larch, and general lessons learned from this exercise.

1 Introduction

The Miró visual languages let users specify formally through pictures the security configuration of file systems (i.e., which users have access to which files) and general security policy constraints (i.e., rules to which a configuration must conform). With the Miró editor, users draw both types of pictures and access other Miró tools.

This paper presents a non-trivial case study on the use of the Larch specification languages to describe the Miró languages and editor. We had two goals in mind while writing the specification: to end up with a formal *specification* that could serve as both documentation of and a basis for formal reasoning about the *specificand*, i.e., the Miró languages and editor; and to apply Larch to determine its strengths and weaknesses. Though there are a growing number of cases studies of formal specifications for formal methods like VDM and Z, very few examples of Larch specifications have been published. Moreover, whereas previous Larch papers have emphasized and presented Larch traits, ours presents a balance of both traits and interfaces.

We begin with brief descriptions of Miró and Larch in Section 2 and present excerpts from the specification in Section 3. In Section 4 we discuss properties of the specificand provable from the specification, limitations of Larch, and general lessons learned from this exercise. All of the traits have been checked for syntactic and static semantic correctness using the Larch Shared Language (LSL) Checker and the interface specification has been checked for syntactic and type correctness using the Generic Concurrent Interface Language (GCIL) Checker.[1] Including blank lines and comments in the code and specification, the code size of the editor is 8400 lines of Common Lisp and the Larch specification is 1035 lines of Larch traits and interfaces.

2 Miró and Larch

2.1 Specificand: Miró

Miró consists of two visual languages, the *instance* language and the *constraint* language [HMT+90]. An *instance (language) picture* graphically denotes an access matrix that defines which users have which accesses to which files. Instance pictures model the specific security configuration between a set of users and a set of files, e.g., Alice cannot read Bob's mail file. A *constraint (language) picture* denotes a set of instance pictures (or equivalently, the corresponding set of access matrices) that satisfies a particular security constraint, e.g., users with write access to a file must also have read access. When an instance picture, *IP*, is in the set denoted by a constraint picture, *CP*, we say *IP* "matches" *CP* or *IP* is "legal with respect to" *CP*.

[0]This research was sponsored by the Avionics Lab, Wright Research and Development Center, Aeronautical Systems Division (AFSC), U. S. Air Force, Wright-Patterson AFB, OH 45433-6543 under Contract F33615-90-C-1465, Arpa Order No. 7597. A. Moormann Zaremski is also supported by a fellowship from the Office of Naval Research.

[1]The full specification is available as a technical report upon request from the second author[Zar91]. Tools are available upon request from the first author.

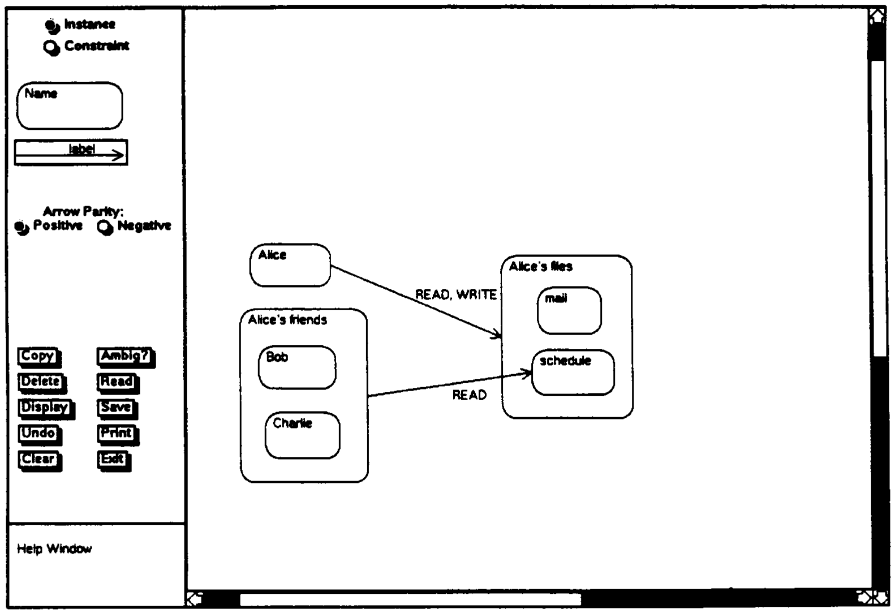

Figure 1: The Miró editor and a sample instance picture

The basic elements in the instance language are *boxes* and *arrows*. Boxes that contain no other boxes represent users and files. Boxes can contain other boxes to indicate groups of users and directories of files. User group boxes may overlap to indicate a user is in more than one group. Labeled arrows go from user (group) boxes to file (group) boxes; the label indicates the access mode, e.g., read or write. Access rights are inherited by corresponding pairs of boxes contained within boxes connected by arrows, thus minimizing the number of arrows necessary to draw. Arrows may be negated to indicate the denial of the labeled access.

Figure 1 shows an instance picture, as drawn in the Miró editor. The positive arrow from **`Alice`** to **`Alice's files`** indicates that Alice has read and write access to her files. The positive arrow from **`Alice's friends`** to Alice's **`schedule`** file indicates that both Bob and Charlie have read access to Alice's schedule. By default, since there are no arrows between **`Alice's friends`** and her other files, Bob and Charlie do not have read access to Alice's mail file. We could also show this property with an explicit negative arrow between **`Alice's friends`** and her **`mail`** file.

The *constraint* language also consists of boxes and arrows, but here the objects have different meanings. A box labeled with an expression defines a set of instance boxes. E.g., the left-hand box in Figure 2 denotes the set of instance boxes of type User. There are three types of arrows, allowing us to describe three different relations between boxes in an instance picture, *IP*: syntactic (solid horizontal) – whether an arrow explicitly appears in *IP*; semantic (dashed horizontal) – whether an access right exists in the matrix denoted by *IP*; and containment (solid vertical with head inside box) – whether a box is nested within another in *IP*. Additionally, the ***thickness*** attribute of each constraint object is key in defining a constraint picture's meaning: in general, for each set of instance objects that matches the thick part of the constraint, there must be another set of objects (disjoint from the set matching the thick part) that matches the thin part. Figure 2 shows a constraint picture that specifies that users who have write access to a file must have read access to it as well.

The Miró editor provides the facilities to create, view, and modify both instance and constraint pictures. Pictures can be saved in files and read back into the editor. The editor also serves as an interface to other Miró tools, e.g., one that generates the access matrix denoted by an instance picture, and one that translates a picture into PostScript form. The left-hand side of the window in Figure 1 displays a menu from which users can select the type of picture and object they wish to draw. Other menu buttons provide additional editing commands and interfaces to

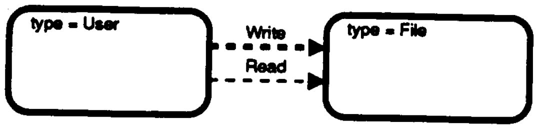

Figure 2: A sample constraint picture

the other Miró tools. The editor maintains language-specific constraints as users draw pictures. For example, all arrows in Miró pictures must be attached to boxes: if a user moves a box in the picture, all arrows attached to that box also move. Our formal specification captures this behavior precisely.

2.2 Specification Language: Larch

We wrote our specifications using Larch specification languages. We assume some rudimentary knowledge of Larch, present a refresher here, and give further details as we present the specification. See [GHW85, GHM90] for more details.

Larch provides a "two-tiered" approach to specification. In one tier, the specifier writes *traits* in the Larch Shared Language (LSL) to assert state-independent properties of a program. Each trait introduces *sorts* and *operators* and defines equality between terms composed of the operators (and variables of the appropriate sorts). E.g., the *Box* trait (Figure 4) introduces the sort B and the operator *copy_box*; four equations constrain the meaning of *copy_box*.

In the second tier, the specifier writes *module interfaces* in a Larch interface language, such as the Generic Concurrent Interface Language (GCIL)[Ler91], to describe state-dependent effects of a program. A **requires** clause states each operation's pre-condition; an **ensures** clause, its post-condition; a **modifies** clause lists those objects whose value may possibly change. The assertion language for the pre- and post-conditions is drawn from LSL traits. Through **based on** clauses, a Larch interface links to LSL traits by specifying a correspondence between (programming-language specific) types and LSL sorts. An object has a type and a value that ranges over terms of the corresponding sort.

Part of the interface specification for the editor below defines the type *Editor*, which is based on the *Ed* sort, introduced in the EdTrait trait. The *MoveBoxes* operation's pre-condition requires that some non-empty set of objects be selected. The post-condition says that the state of the picture in the editor is updated (as defined by the *move_boxes* operator whose meaning is obtained from EdTrait) and that all objects are unselected. In a post-condition an undecorated formal, *e*, stands for the initial value of the object; a subscripted one, *e!post*, stands for the final value. The **modifies** clause states that *MoveBoxes* may change only the editor and no other object.

```
object miro_editor
. . .
type Editor based on Ed from EdTrait
. . .
operation MoveBoxes (delta : Cp, e : Editor)
requires (¬ (isEmpty(e .selected_objs)))
modifies (e)
ensures
  ((e!post .picture) = move_boxes((e .picture),
     boxes(e .selected_objs), delta))
  ∧ ((e!post .selected_objs) = {})
```

3 The Specification

We divide the specification into two main pieces: one specifying properties of Miró pictures and one specifying the behavior of the editor. We use LSL to describe Miró pictures and then additionally use GCIL to specify the editor's operations that manipulate the pictures.

Figure 3 illustrates how the traits of the LSL part of the specification fit together. Each oval corresponds to a trait and an arrow indicates that one trait **includes** another. A picture in either the instance or constraint language is a collection of boxes and arrows (BasicPicture trait). Miró pictures are further constrained to satisfy *well-formedness* properties (Picture and WF-Picture), which include, for example, the condition that arrows be attached to boxes. Pictures drawn in the instance and constraint languages are structurally very similar, so our approach is to factor out the properties common to both languages (bold ovals in Figure 3), and then specialize for each language (dotted for instance and dashed for constraint). At the bottom we define the *EdTrait* trait, which includes all the others; it is the link between the LSL and GCIL tiers in the editor specification. In this paper we will walk through only the traits along the boldface spine in the figure. Also, to save space we will typically present only the signature and not the equations in each trait.

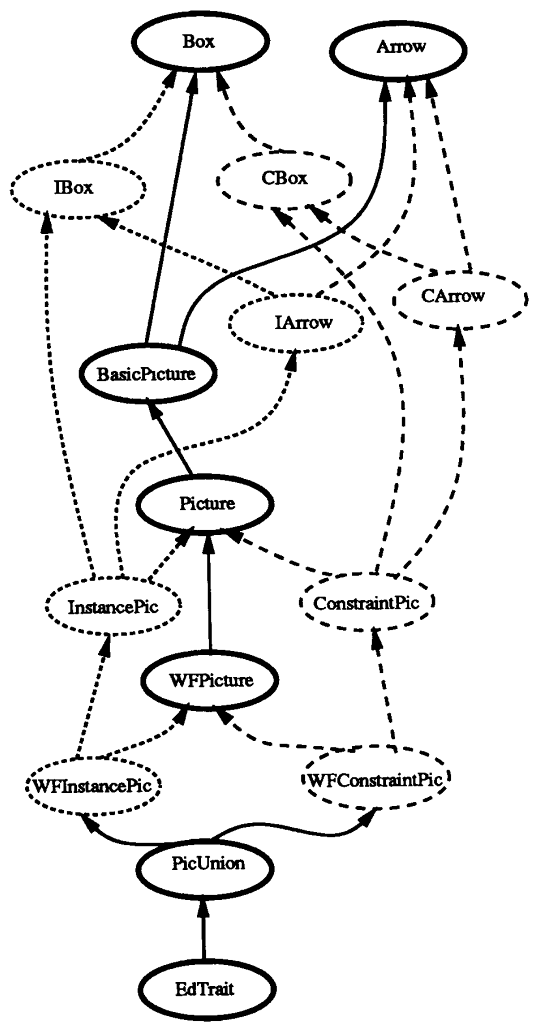

Figure 3: The dependencies of the Miró traits

3.1 The Miró Languages

3.1.1 Basic Objects

Boxes and arrows are the basic objects of any Miró picture. Instance and constraint pictures differ only in the attributes of their respective boxes and arrows and in the rules for combining them into pictures. Traits for boxes and arrows are later specialized to distinguish between instance and constraint pictures.

A box object has a value of sort B (see Box trait

```
Box(B) : trait
  includes CoordPair, Integer
  B tuple of pos : CP, size : CP, label : LabelSort,
    thick : Bool, starred : Bool, sysname : Int
  introduces
    copy_box : B → B
  asserts
    ∀ b : B
      (copy_box(b)).pos == b.pos,
      (copy_box(b)).size == b.size,
      (copy_box(b)).thick == b.thick,
      (copy_box(b)).starred == b.starred
```

Figure 4: The Box Trait

in Figure 4) and has *pos* and *size* attributes.[2] A box also has a *label* attribute, which will be customized for instance and constraint boxes, and the boolean attributes *thick* and *starred*, needed for constraint pictures. Finally, we use *sysname* to identify a box. Although we do not require it at this level, we assume that sysnames are unique. Sysnames serve two purposes: to distinguish between otherwise identical objects and to provide an easy way to identify objects in a picture for other operations (e.g., deleting a box).

The *Box* trait also introduces operators on boxes. The **tuple of** notation in Larch automatically produces the *generator* for the tuple sort: an operator *[...]* that takes as its arguments all of the attributes of the sort and produces something of the tuple sort. The tuple shorthand also generates operators of the form *b.field* and *set_field(b, field_val)* for each field *field* (where *b* is of sort *B* and *field_val* has the same sort as the field *field*).

The **introduces** clause declares additional operators and the equations in the **asserts** clause give them their meaning. The reason we even need a *copy_box* operator as opposed to relying on Larch's built-in equality operator for all sorts is that not all values of all fields are the same when one box is a copy of the other. Having an explicit *copy_box* operator also makes it convenient to specify default initialization values for certain fields. For example, one issue in the design of the editor was whether or not a copy of a box should have the same label, or should have a default label (the empty string). Thus, we choose instead to write equations only for fields we require to be the same, and allow values of other fields to be specified in another trait or at the interface level.

Boxes in the instance language differ from those in the constraint language in two ways. First, the sort

[2]The *CP* sort is defined in the *CoordPair* trait as a tuple of two integers.

of values for some attributes are different. Namely, an instance box's *label* is a string whereas a constraint box's label is "box descriptor" – a boolean expression that describes a set of boxes. We handle this difference by using the generic sort *LabelSort*; then in the *InstanceBox* trait, we rename *LabelSort* with the sort *Str* (for strings) and in the *ConstraintBox* trait, we rename it with the sort *BoxDesc* (for box descriptors).

Second, some attributes are meaningful for only constraint pictures and hence are unnecessary for instance pictures (e.g., *thickness* and *starred*). We could avoid this problem if either Larch provided a way to extend (subtype) tuples or we were willing to use nested tuples (see Section 4). However, since there are only a few of these attributes, we choose instead to include at the trait level all possible attributes for the two kinds of boxes, and then make assertions at the interface level that place constraints that will distinguish between instance and constraint boxes.

We specify properties on arrows similarly (using the **tuple** construct) and omit the details here. *To_box* and *from_box* are two of many fields in the tuple for arrows and are used to keep track of the boxes at the arrow's head and tail.

3.1.2 Pictures

A picture is a set of boxes and a set of arrows. To avoid a long and confusing trait, we divide the specification of pictures into three main traits: *BasicPicture*, *Picture*, and *WFPicture*. *BasicPicture* introduces the picture sort, *Pic*, and basic operators on a picture. *Picture* introduces the sort *Object*, operators that manipulate objects in the picture, and sorts and operators for the Change Attribute operation. Finally, *WFPicture* introduces the definition of well-formedness and operators that build on top of the basic operators to create and manipulate well-formed pictures.

The BasicPicture trait (Figure 5) introduces the sort *Pic* for pictures. The **includes** clause lets us use all sort and operator names from the included traits with appropriate renamings. E.g., the renaming of sort identifiers in the Set trait (from the Larch Handbook) gives us a sort BSet for sets of boxes and an operator *empty_BSet* to denote the empty set of boxes. The operators that generate a picture are *create_picture*, *insert_box*, and *insert_arrow*. This trait also introduces the operators *move_a_box*, *resize_a_box*, *delete_box*, *delete_arrow*, and *copy_picture*. We use *pic_union* later in traits to perform the interface level copy operation. The operators *boxes* and *arrows* are observer functions on pictures.

The axioms defining these operators are straight-

```
BasicPicture : trait
  includes Box, Set(B, BSet,empty_BSet for {}),
    Arrow, Set(A, ASet,empty_ASet for {})
  introduces
    create_picture : → Pic
    insert_box : Pic, B → Pic
    insert_arrow : Pic, A → Pic
    move_a_box : Pic, B, CP → Pic
    resize_a_box : Pic, B, CP → Pic
    delete_box : Pic, B → Pic
    delete_arrow : Pic, A → Pic
    copy_picture : Pic → Pic
    pic_union : Pic, Pic → Pic
    boxes : Pic → BSet
    arrows : Pic → ASet
    . . .
  asserts
    Pic generated by (create_picture, insert_box,
      insert_arrow)
    Pic partitioned by (boxes, arrows)
    . . .
```

Figure 5: Part of the BasicPicture Trait

forward and given in the standard style of "algebraic" specifications (define the meaning of each non-constructor operator in terms of each constructor operator). We give details of only one here. *Copy_picture* recurses through the objects in the picture and calls the appropriate copy operator on each object (box or arrow).

```
copy_picture(create_picture) == create_picture
copy_picture(insert_box(pic,b)) ==
    insert_box(copy_picture(pic), copy_box(b))
copy_picture(insert_arrow(pic,a)) ==
    insert_arrow(copy_picture(pic), copy_arrow(a))
```

The *Picture* trait (Figure 6) builds on the *BasicPicture* trait by introducing a new sort *Object*, which is a union of the box and arrow sorts. Most of the editor's procedures work regardless of whether its argument is a box or arrow; thus, for each operator common to the box and arrow sorts, we introduce an equivalent operator for the *Object* sort.

The operator *objects* returns the set of all objects in a picture; *boxes* and *arrows* extract the sets of boxes and arrows from a set of objects. The operator *toggle_in* adds the specified object to a set of objects if it is not already in it, otherwise it deletes the object. The editor trait uses *toggle_in* to maintain a set of selected objects in a picture. The operator *copy_object* is the object analog of the corresponding box and arrow operators. The operator *change_attr* lets us set the value of an object's attribute. Some of the complexity of this trait results from our decision to use the **union**

```
Picture (Pic) : trait
  includes BasicPicture,
    Set(Object, ObjSet, empty_OSet for {})
  Object union of (box : B, arrow : A)
  Label enumeration of (label, sysname, thick, starred,
    pos, size, parity, from_box, to_box, kind)
  Value union of (bool:Bool, cp:CP, str:Str, int:Int,
    b:B, arrow_kind:ArrowKind)
  introduces
    objects : Pic → ObjSet
    boxes : ObjSet → BSet
    arrows : ObjSet → ASet
    toggle_in : ObjSet, Object → ObjSet
    copy_object : Object → Object
    change_attr : Object, Label, Value → Object
    . . .
```

Figure 6: Part of the Picture Trait

construct for objects. The advantage of putting this complexity here is that we keep the interfaces clean and simple; we discuss some of the disadvantages in Section 4.

3.1.3 Well-formed Pictures

BasicPicture and *Picture* traits introduce the picture sort *Pic* and basic operators on pictures. However, Miró pictures drawn in the editor cannot be an arbitrary collection of boxes and arrows; we require that they be *well-formed*.

```
WFPicture (Pic) : trait
  includes Picture
  introduces
    well_formed : Pic → Bool
    all_arrows_attached: Pic → Bool
    extract_wf : ObjSet → Pic
    . . .
```

Figure 7: Part of the WellFormedPicture Trait

Among the many operators in the WFPicture trait (Figure 7), we introduce operators for defining well-formedness and extracting a maximal well-formed subset of a set of objects. We only introduce but give no equations defining *well_formed* operator in this trait because instance and constraint pictures have different notions of well-formedness. However, since one well-formedness condition common to both instance and constraint pictures is that all arrows must be attached to boxes, we introduce and define here the operator *all_arrows_attached*. There are other well-formedness conditions, e.g., arrows must go from user boxes to file boxes in an instance picture, but for this paper,

```
WFInstancePic : trait
  includes InstancePic, WFPicture(IPic,
    create_instance_pic for create_picture, IB for B, Str
    for LabelSort, IA for A , IO for Object, IOSet for
    ObjSet)
  introduces
    ambiguous : IPic → Bool
  asserts ∀ (ipic:IPic)
    well_formed(ipic) == all_arrows_attached(ipic)
```

Figure 8: The WellFormed Instance Picture Trait

we assume only the arrows-attached property.

We use *extract_wf* in the editor interface to describe the behavior of the command to copy a set of selected objects, which itself is a picture that may or may not be well-formed. The result of *extract_wf(os)* is a picture that contains all the objects of *os* except the "dangling" arrows (i.e., arrows that are not attached to boxes in *os*).

$$boxes(extract\_wf(pic, os)) == boxes(os)$$
$$a \in arrows(extract\_wf(pic, os)) == (a \in arrows(os)) \\ \wedge((a.to\_box) \in boxes(os)) \wedge ((a.from\_box) \in boxes(os))$$

To show how we define properties of a well-formed instance picture, let us look at the WFInstancePicture trait (Figure 8), which includes and specializes the WFPicture trait. The **asserts** equation states that an instance picture is *well_formed* if and only if all arrows in the picture are attached. In WFInstancePicture we also introduce the operator *ambiguous*. Because there are both positive and negative arrows in the instance language, it is possible to draw pictures that lead to the reasonable interpretation that a user has access to a file and to the reasonable interpretation that the user does not. The semantics of ambiguity is well-defined[MTW90], but is somewhat complex, so we do not reproduce it here. The Miró editor enforces well-formedness, but does not require that the pictures drawn are always unambiguous. For this reason, we cannot write the equation:

$$well\_formed(ipic) == all\_arrows\_attached(ipic) \wedge \\ \neg ambiguous(ipic)$$

Finally, most of the editor's procedures work on pictures regardless of whether they are drawn in the instance or constraint language. For example, moving a collection of boxes behaves the same regardless of whether they are instance or constraint boxes. To avoid duplicating the entire interface (e.g., have two separate *Move_Instance_Picture* and *Move_Constraint_Picture* operations) we introduce a union sort P to handle both instance and constraint pictures – just as we introduced an Object sort to han-

```
PicUnion: trait
  includes WFInstancePic, WFConstraintPic,
    Set(A,ASet,empty_ASet for {}),
    Set(B,BSet,empty_BSet for {}), Set(O,OS)
  B union of (ibox:IB, cbox:CB)
  A union of (iarrow:IA, carrow:CA)
  O union of (iobj:IO, cobj:CO)
  P union of (instance:IPic, constraint:CPic)
  BoxLabel union of (ilabel:Str, clabel:BoxDesc)
  PicType enumeration of (instance, constraint)
  introduces
    boxes : P → BSet
    create_picture : PicType → P
    insert_box : P, B → P
    insert_arrow : P, A → P
    copy_picture : P → P
    pic_union : P, P → P
    toggle_in : OS, O → OS
    change_attr : O, Label, Value → O
    move_boxes : P, BSet, CP → P
    resize_box : P, Int, CP, CP → P
    delete_objs : P, OS → P
    extract_wf : P, OS → P
    well_formed : P → Bool
    ambiguous : P → Bool
    . . .
```

Figure 9: Part of the PicUnion Trait

dle both boxes and arrows. Figure 9 shows part of the signature for the PicUnion trait. By providing a union sort P and the appropriate operators, the editor's operations can now work on either type of Miró picture. Most of the operators introduced in PicUnion trait deal with coercing to and from values of the picture sort P and values of instance and constraint picture sorts.

3.2 Editor

Now that we have a way of describing Miró pictures, we are ready to describe the editor's effects of manipulating them. We first describe the editor state at the trait level and then use interfaces to describe state changes. We ignore much of the lower-level detail (e.g., mapping to mouse and keyboard actions, how text interaction works); these are described informally in the Editor User's Guide.

3.2.1 LSL Level

The EdTrait trait (Figure 10) introduces the sort, *Ed*, which we use to model the editor state, again via the **tuple** construct. The *pos* and *size* fields indicate the location and size of the editor window on the screen.

```
EdTrait: trait
  includes PicUnion, PixelMap, AttributeSorts
  OT enumeration of (box, arrow)
  LineThickness enumeration of (thin, thick)
  Ed tuple of ( pos : CP, size : CP, picture : P,
    picture_type : PicType, object_type : OT,
    arrow_kind : ArrowKind, arrow_parity: Bool,
    thickness : LineThickness, starred : Bool,
    selected_objs : OS )
  introduces display_window : Ed → PixelMap
```

Figure 10: The Editor Trait

The *picture* field contains the current Miró picture, of sort P (introduced in the PicUnion trait). *Selected_objs* is the set of currently selected objects in the picture. The remainder of the tuple describes the current "mode" of the editor (as indicated in the menus): *picture_type* indicates whether the current picture is an instance or constraint picture, *object_type* is either box or arrow, *arrow_kind* is the kind of arrow - syntactic, semantic or containment (relevant for only constraint pictures), and the rest of the attributes are self-explanatory. EdTrait introduces one additional operator: *display_window* is left unspecified here, but is intended to be a mapping from the abstract editor value to actual screen pixels.

3.2.2 Interface Level

Now that we have build up a rich trait, we are ready to specify the editor's interface. First we name the object we are specifying (*miro_editor*), and then establish correspondences between types and sorts (only 4 of the 12 are shown below). The **invariant** specifies properties that must be true after every operation and before all operations except those named in the **initialized by** clause. Here, the invariant states that the picture maintained by the editor is always well-formed.

```
object miro_editor
using Editor
initialized by CreateEditor
private e : Editor
invariant (well_formed(e .picture))
type Cp based on CP from Editor
type Box based on B from Editor
type Editor based on Ed from Editor
type ObjSet based on OS from Editor
...
```

There are twelve operations specified in the full interface specification. They include creating an editor, drawing a box, drawing an arrow, selecting an object, selecting a group of objects, unselecting objects, moving boxes, resizing boxes, deleting objects, copying objects, changing an object's attribute, and clearing the editor. Below we walk through three to show what information is typically captured in an interface and how we rely on the traits to make this precise.

CreateEditor is the operation that gets things started. Its effect is to initialize a new editor object with the default initial modes:

```
operation CreateEditor (pos, size : Cp)
      returns (e : Editor)
   requires true
   ensures (newobj (e)) ∧
      (e = [pos, size, create_picture(instance), instance,
         box, syn, false, thin, false,{}])
```

Resizing in the editor is fairly restrictive: a user can resize only boxes (not arrows), and must resize exactly one at a time. The third conjunct of the *ResizeBox* **requires** clause captures these restrictions. The hard "work" in the **ensures** clause is done using the trait operator *resize_box* defined for pictures; the post-condition also states that once a resize operation is performed all objects are unselected.

```
operation ResizeBox (b:Box, pos : Cp, size : Cp,
      e : Editor)
requires (b ∈ boxes((e .picture))) ∧
   ((selected(b)) = true) ∧
   ((e .selected_objs) = {box_to_O(b)})
modifies (e, b)
ensures ((e!post .picture) = resize_box((e .picture),
         sysname(b), pos, size)) ∧
      ((e!post .selected_objs) = {})
```

The copy operation in the editor is somewhat complex because of the well-formedness constraint. Copy operates on a subset of the currently selected objects, namely the maximal well-formed subset (i.e., all objects except "dangling" arrows). The **ensures** clause of *CopyObjs* thus says the new picture object, *newpic*, is the result of copying the well-formed subset of the selected objects of *e.picture*. *e!post.picture* is then the result of combining the existing picture with *newpic*, which has been moved by *delta*. *CopyObjs* also unselects all objects.

```
operation CopyObjs (objs : ObjSet, e : Editor,
   delta : Cp) returns (newpic : Picture)
requires (objs = (e .selected_objs))
modifies (e)
ensures (newobj (newpic)) ∧(newpic =
   copy_picture(extract_wf((e .picture), objs)))
   ∧((e!post .picture) = pic_union((e .picture),
      move_boxes(newpic, boxes(newpic), delta)))
   ∧((e!post .selected_objs) = {})
```

4 Discussion

We first discuss some properties about Miró pictures and the editor that are provable consequences of our specification, then two limitations of Larch that arose out of this specification exercise, and finally some general lessons learned from having done the specification after the implementation.

4.1 Stating Consequences

Larch provides a way to state consequences of a trait's theory through an **implies** clause. This clause is a good place to document additional assertions about a specificand. As a simple example, and also one that shows the interplay between traits and interface, consider copying objects. At the interface level the *CopyObjs* operation copies only the subset of selected objects that form a well-formed picture. We could have defined the *copy_picture* operator in the BasicPicture trait to copy only the well-formed subset but decided it was more appropriate to specify this restriction at the interface level, leaving the trait level more general: the *copy_picture* operator copies all objects in a picture. We add the following **implies** clause to the BasicPicture trait in order to record this decision explicitly. We cannot make the stronger statement that *copy_picture(p) == p* because when objects are copied, not all of the fields, e.g., box labels, are copied.

```
implies ∀ (p:Pic)
   size(boxes(copy_picture(p))) == size(boxes(p))
   size( arrows(copy_picture(p))) == size( arrows(p))
```

We can also state the strong assertion that a well-formed picture is actually just a graph where boxes are nodes and arrows are edges. Thus, we add the following consequence to the WFPicture trait:

```
implies Graph(B, A, Pic, CreatePicture for empty,
   insert_box for addNode, insert_arrow for addEdge)
```

where we are reusing the Graph trait from the Larch Handbook.

Finally, we can take this one step further. If we add to the WFInstancePic trait the requirement that each arrow goes from a user box to a file box and no arrow goes from a box of one type to a box of the same type, we can state as a consequence that a well-formed instance picture is a bipartite graph.

4.2 Assessment of Larch

4.2.1 Subtyping Tuples

We made a critical design decision by representing each Miró graphical object as a tuple. Tuples conveniently let us associate attributes with each kind of object and give us operators that let us set and get values of each of those attributes. However, the main drawback to using tuples is that Larch does not permit tuple "subtyping". It would be more general to define a GraphicalObject trait that introduces a tuple sort GO with fields like *label* and then to define Box and Arrow traits, introducing B and A sorts, each as a "subtype" of GO and InstanceBox and ConstraintBox as subtypes of Box. B would "automatically" have the same fields as GO plus ones like *position* and *size*; A would add fields like *from_box* and *to_box*. Then we could write for *b* of sort B, *b.label* and *b.size*, and for *a* of sort A, *a.label* and *a.from_box*. However, in Larch, if we were to factor out attributes common to all Miró objects into a GraphicalObject trait, we would be forced to use nested tuples in the tuples in the Box and Arrow traits and to write *b.go.label* or *a.go.label* where *go* is the field name of sort GO.

Instead, we decided to avoid nesting tuples entirely since the resulting specifications would be less readable. However, this decision forced us to include attributes of tuples in some traits that make sense only for subsequent uses of that trait. For example, the Box trait's tuple has a *thick* attribute that makes sense only for constraint, not instance, pictures. This kind of problem and solution is well-known in the "object-oriented" community.

4.2.2 Union Sorts

Sort checking is invaluable in Larch, but one place where it gets in the way is in the use of unions. In the PicUnion trait, the *P* sort is introduced to be the union of instance and constraint pictures. In the editor, it does not matter whether we have an instance picture or a constraint picture, we just want to select an object or copy objects. So, if the operators *copy_picture*: P -> P, *copy_picture*: IPic -> IPic and *copy_picture*: CPic -> CPic are defined, then ideally we would like the constraints on *copy_picture* to hold regardless of whether the picture is an instance or constraint. Instead we are forced to first determine whether the picture is an instance (or a constraint) picture and then coerce the picture to be an instance (or constraint) picture so that we can apply the appropriate, more specific, copy operator, and finally, coerce back the result into the more general picture sort, *P*:

```
copy_picture(p) == if tag(p) = instance
  then instance(copy_picture(p.instance))
  else constraint(copy_picture(p.constraint))
```

Another variation of this problem arose from specifying the *change_attr* operator in the Picture trait. We would like to specify *change_attr* with the following simple equation:

```
change_attr(obj, field, value) == set_field(obj, value)
```

but we cannot for two reasons. First, an LSL operator name is an unstructured identifier – *set_field* really stands for a set of possible identifiers depending on what the actual name of *field* is in the left-hand side of the equation above. Instead, we are forced to use a large if-then-else statement to cover each possible field. Second, the *obj* and *value* parameters to *change_attr* are union sorts. As argued above, we need to do explicit coercion between the union sort and the specific sorts of the union (and vice versa) in order to achieve the intended effects of the above equation. Hence, *change_attr* becomes one big two-layered if-then-else clause, first on object sort (box or arrow), then on field name. For each valid object/field pair, there is a clause to do the appropriate coercions and assign the value to the appropriate field:

```
change_attr(obj, field, value) ==
  if tag ( obj ) = box then
    if (field=pos) then
      box(set_pos(obj.box, value.cp))
    else ...
```

4.3 Specifying After Implementing

We began writing formal specifications in parallel with designing the Miró languages and designing and implementing of the editor. We wrote three major versions where the last version (this one) was written after the implementation was running. The current version itself went through at least six minor iterations. Writing a formal specification after an implementation has two obvious implications. One is that the specification tends to be biased towards the implementation; the other is that places for improving the implementation become embarrassingly evident. We found both to be true in our case.

Having already implemented the editor before completing the specification, we had a concrete model of the languages and editor in mind. This model led to an implementation-biased initial specification. In each subsequent iteration we removed some of the "implementation details." We believe the final specification is relatively unbiased, but that we would have taken fewer steps to get where we are had we written more of the specification before the implementation.

One example of implementation detail we removed is in the definition of well-formedness. In the implementation of the editor, the condition that all arrows be attached to boxes is enforced automatically by checking and reestablishing certain constraints on arrow objects. So, in a previous version of the specification, the arrow sort A had fields for the coordinate positions of the arrow's head and tail. The *WFPicture* trait defined an operator called *adjust_arrows* that took a picture and a box, and based on the position of the boxes at the arrow's head and tail, algorithmically recomputed the new coordinates. But at a more abstract level, coordinate information for an arrow is not important; we only need to know which boxes are at its head and tail. As a result, we define an abstract well-formedness property (e.g., all arrows are attached) at the trait level and require through an interface invariant that all editor operations enforce this property. No where in the current specification do we ever give a precise algorithm for enforcing well-formedness. It would be up to the implementor to decide whether and how arrows need to be adjusted to maintain well-formedness.

The second observation from doing the specification after the implementation is that doing the specification earlier would have resulted in a better implementation. One example to support this argument is our experience in implementing multiple selection in the editor. The initial implementation of the editor allowed at most one object to be selected at any time. While working on the specification we decided to extend the editor to allow selecting multiple objects. So before coding multiple selection, we wrote a mostly formal specification defining precisely what the effects of each editor operation would be on the set of selected objects. The result was an implementation of multiple selection that was clean, consistent and relatively easy to add to the editor.

5 Related Work

The largest collection of Larch traits is the Larch Handbook[GHW85] (990 lines). The extended example in [GGH90] specifies some traits for a simple windowing system. Larch has also been used to specify properties of objects in a transaction-based distributed system[Win88]. To our knowledge, our specification is the largest Larch specification ever written and the only written for a "real" system.

In our specification, we assume that details about how Miró pictures are represented on the screen and what keyboard and mouse inputs activate the specified procedures are specified elsewhere. This in itself is a difficult problem, but has been addressed by others ([GGH90] and[Bow89]). Thus, we chose to focus at the next level of detail: properties of pictures and the editor. Formally specifying graphics packages and user interfaces is an active research area, as witnessed by the recent workshop, "Formal Methods in Computer Graphics" [DF91].

6 Acknowledgments

We thank Allan Heydon, Rick Lerner, Mark Maimone and Doug Tygar for their extremely helpful comments on earlier drafts. We are also indebted to Brad Myers and the rest of the Garnet group for help in implementing the Miró editor.

References

[Bow89] Jonathan Bowen. Formal specification of window systems. TR PRG-74, Oxford University Computing Laboratory, June 1989.

[DF91] D. A. Duce and G. P. Faconti, eds. *Proceedings of the Workshop on "Formal Methods in Computer Graphics"*. Eurographics Association, June 1991.

[GGH90] S.J. Garland, J.V. Guttag, and J.J. Horning. Debugging Larch Shared Language specifications. TR 60, DEC-SRC, 1990.

[GHM90] J.V. Guttag, J.J. Horning, and A. Modet. Report on the Larch Shared Language: Version 2.3. TR 58, DEC-SRC, 1990.

[GHW85] J.V. Guttag, J.J. Horning, and J.M. Wing. Larch in five easy pieces. TR 5, DEC-SRC, 1985.

[HMT+90] A. Heydon, M.W. Maimone, J.D. Tygar, J.M. Wing, and A.M. Zaremski. Miró: Visual specification of security. *IEEE TSE*, 16(10):1185–1197, Oct. 1990.

[Ler91] Richard Lerner. Specifying objects of concurrent systems. Ph.D. thesis,CMU-CS-91-131, 1991.

[MTW90] M.W. Maimone, J.D. Tygar, and J.M. Wing. Formal semantics for visual specification of security. In S.K. Chang, ed., *Visual Languages and Visual Programming*. Plenum, 1990.

[Win88] J.M. Wing. Specifying Avalon objects in Larch. TR CMU-CS-88-208, CMU, 1988.

[Zar91] Amy Moormann Zaremski. A Larch specification of the Miró editor. TR CS-91-111, CMU, 1991.