A Larch Specification of the Miró Editor

Amy Moormann Zaremski

February 25, 1991

CMU-CS-91-111$_3$

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

The Miró visual languages [HMT+90] allow a user to specify the security configuration of a file system and general security policy constraints. We describe our use of the Larch specification languages [GHW85, GHM90] to specify Miró pictures and the Miró graphical editor; we include the complete specification as an appendix.

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. The author is also supported by a fellowship from the Office of Naval Research.

Keywords: formal specification, visual language, Larch, Miro

1 Introduction

The Miró visual languages allow a user to specify the security configuration of a file system (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, a user can draw both types of pictures and invoke other Miró tools on them.

This paper describes our use of the Larch specification languages to specify the Miró languages and editor and discusses some of the issues that arose from this work. This formal specification has two main purposes: in the application domain, the specification serves as formal documentation of and a basis for reasoning about the Miró languages and editor. In the specification domain, it serves as a useful exercise to determine some of the strengths and weaknesses of Larch.

We begin with brief descriptions of Miró and Larch (Section 2). In Section 3, we present a sketch of the specification, followed by a description of some of the assertions we would like to make about the languages and editor in Section 4. Section 5 discusses some of the more general issues that arose in the work, and we close with areas for further exploration and conclusions (Section 6). The full specification is presented in the appendix. All of the traits have been checked for syntactic and static semantic correctness using the Larch Shared Language (LSL) Checker. The interface specification has been checked for syntactic and type correctness using the Generic Interface Language checker[Ler91].

2 Miró and Larch

2.1 Miró

Miró consists of two visual specification languages, the *instance* language and the *constraint* language [HMT+90]. The meaning of a picture in the instance language is an access matrix that defines which users have which accesses to which files. Instance pictures are used to model the specific security configuration of a particular set of users and files. The constraint language provides a means for defining more general security policies to which a file system configuration must conform. The meaning of a picture in the constraint language can be thought of as the set of instance pictures (or the corresponding set of access matrices) that satisfies a particular security constraint.

The basic elements in the instance language are *boxes* and *arrows*. Boxes that contain no other boxes represent users and files. Boxes can be contained in other boxes to indicate groups of users and directories of files (user group boxes may also overlap so that a user can be in more than one group). Labeled arrows go from one box to another; the label indicates the access mode. The relationship represented by an arrow between two boxes is also inherited by all pairs of boxes contained in those two boxes. Arrows may be negated, indicating the denial of the specified access.

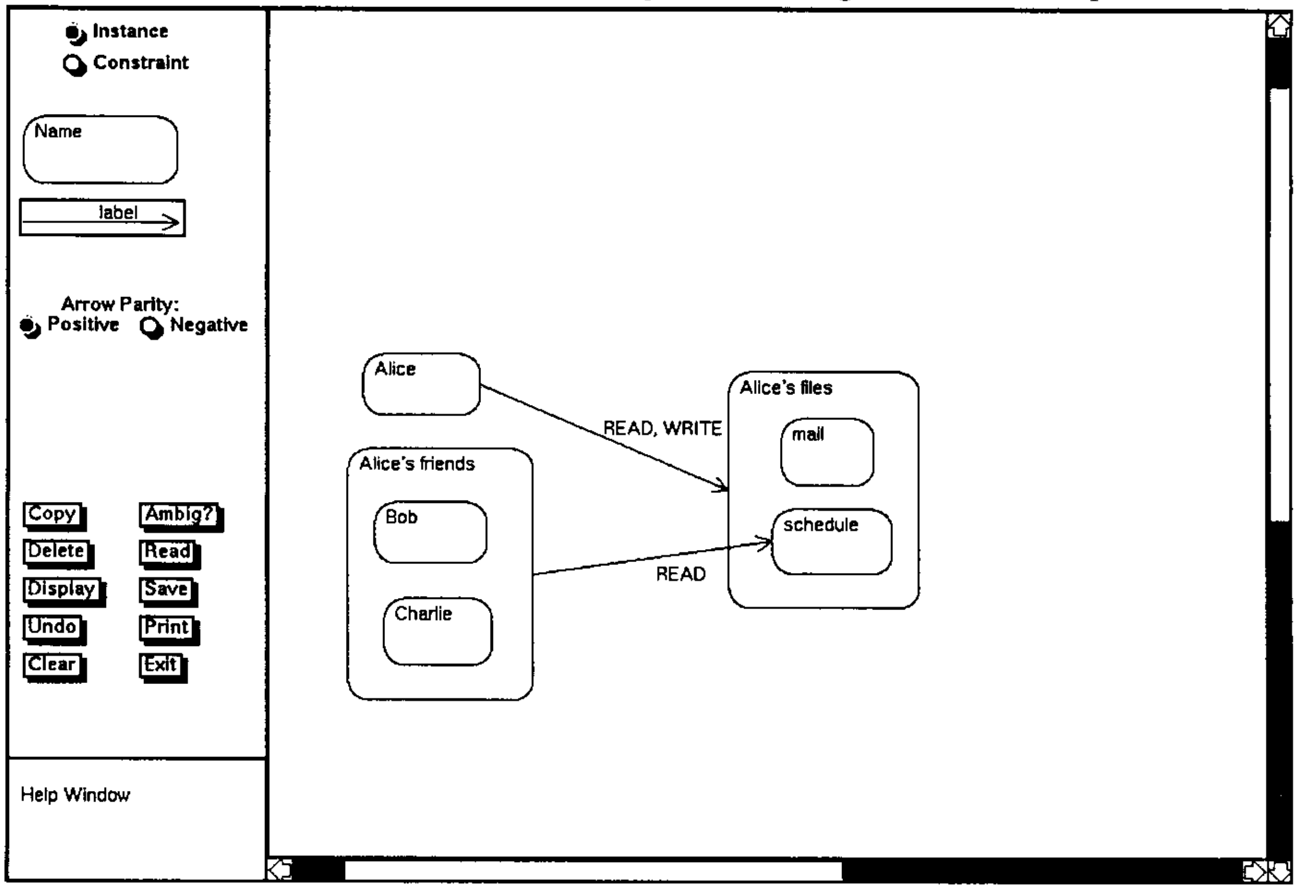

Figure 1: The Miró editor and a sample instance picture

Figure 1 shows a typical 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 is no arrow to indicate the relation between Alice's friends and her mail file, Bob and Charlie do not have read access to Alice's other files (e.g., mail).

The Miró *constraint* language also consists of boxes and arrows, but here the objects have different meanings: a constraint picture defines a set of instance pictures. If a given instance picture satisfies the restrictions in a constraint picture, we say it is *legal.* Different sets of constraints can be used to describe different security policies. In a constraint picture, a box is labeled with an expression that defines a set of instance boxes (for example, in Figure 2, the left-hand box refers to the set of instance boxes of type User). There are three types of arrows, which allow us to describe different relations between boxes in an instance picture: syntactic (solid horizontal), semantic (dashed horizontal), and containment (solid vertical with head inside box). Additionally, each constraint object is either *thick* or *thin* (we call the thick part of the constraint the *trigger*). The thick/thin attribute is key in defining the semantics of a constraint picture: 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 which specifies that a user who has write access to a file should have read access to it as well (i.e., the thick boxes and arrow must match a user u who has write access to a file f. For each such u and f, u must also have read access to f).

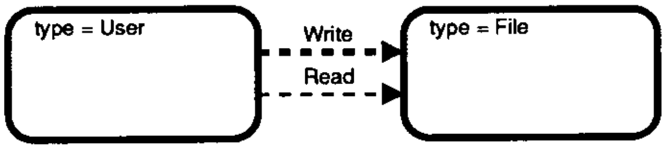

Figure 2: A sample constraint picture

A visual language is not very useful unless there is a way to create and manipulate pictures in the language. 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 that generate the access matrix corresponding to an instance picture, or translate an instance or constraint picture into PostScript form. Figure 1 shows the Miró editor window. A user selects the type of picture and object he or she wishes to draw from the menu along the left side of the window. Buttons in the menu provide additional editing commands and interfaces to other Miró tools. There are some assumptions about the languages built in to the editor. For example, all arrows in Miró pictures must be attached to boxes. The editor maintains this condition. So, for example, if a user moves a box in the picture, all of the arrows that are attached to that box also move.

2.2 Larch

We wrote our specifications using the Larch specification languages. We present a brief overview 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). For example, the *Box* trait (Figure 4) introduces the sort *Bx* and the operators *copy_box* and *is_on_box*; five equations constrain the meaning of *copy_box*.

In the second tier, the specifier writes *module interfaces* in a Larch interface language, such as GCIL [Ler91], an extended Generic Interface Language [Che89], to describe state-dependent effects of a program. A **requires** clause states each procedure'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 EditorState trait. The *ResizeBox* procedure's pre-condition requires that the set of selected objects is exactly the box that is to be resized (*box_to_O* is a coercion operator). The post-condition says that the value of the box is updated (as defined by the *set_size* and *set_pos* operators whose meanings are obtained from EditorState) and that all objects are unselected. In a post-condition an undecorated formal, *e*, stands for the initial value of the object; *e'* stands for the final value. The **modifies** clause states that *ResizeBox* may change only the editor and no other object.

object miro_editor

.

.

   type Editor **based on** Ed **from** EditorState

.

.

   operation ResizeBox (b : Box, pos : Cp, size : Cp)
      requires e.selected_objs = {box_to_O(b)}
      modifies (e_{obj})

```
ensures  b′ = set_size(set_pos(b,pos),size) ∧
         e′.selected_objs = {}:OS
```

3 The Specification

There are two main parts of the specification: specifying properties of Miró *pictures*, and specifying the behavior of the *editor*. We use LSL to describe the properties of Miró pictures and GCIL to define the editor operations that manipulate 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. The *Box* and *Arrow* traits define each kind of graphical object in a picture (boxes and arrows). The *BasicPicture* trait introduces the picture sort and basic picture operators. In order to capture more of the structure of Miró pictures, we define a more restricted kind of picture that includes the *well-formedness* property (*WFPicture*). This includes, for example, the condition that arrows must be attached to boxes. The shaded "helper" traits introduce auxiliary sorts or operators; the *BandASorts* trait defines the sorts for many of the box and arrow attributes.

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 (denoted by bold ovals in Figure 3), and then specialize for each language (denoted by dashed ovals in the figure). The *PicUnion* trait provides a union sort that allows us to talk about either instance or constraint pictures at the editor level. At the bottom we define the *EditorTrait*, which includes all the others; it is the link between the LSL and GCIL tiers in the editor specification. In this section, we describe the traits in bold ovals, the instance traits, and the interface specification. The appendix contains the specification in its entirety.

3.1 Miró Pictures

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.

The *Box* and *Arrow* traits introduce sorts for boxes and arrows respectively. The important

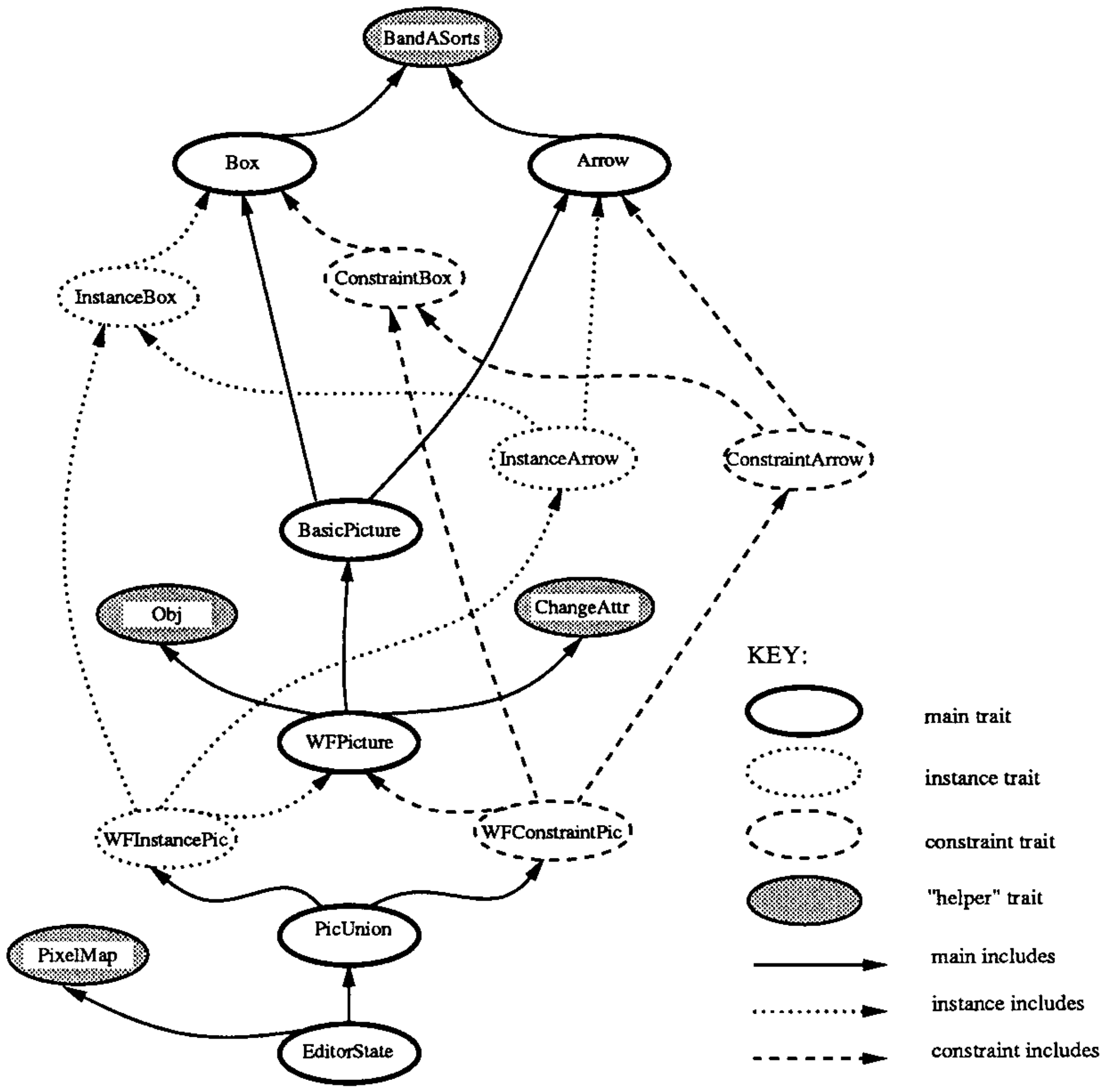

Figure 3: The dependencies of the Miró traits

properties of a box or arrow are its distinguishing graphical and semantic characteristics, such as size, type or name. Larch provides a convenient shorthand for defining a sort as a collection of attributes: the tuple.

3.1.1 Box

Figure 4 shows the *Box* trait, which introduces the tuple sort *Bx*. The sorts of each of the fields (e.g., *CoordPair* and *BoxType*) are defined in the *BandASorts* trait. *Bx* contains the attributes *pos*

$Box(Bx)$: **trait**
  includes $BandASorts$

  Bx **tuple of** $pos : CoordPair, size : CoordPair, b\_label : BoxLabel, thickness : LineThickness,$
    $starred : Bool, box\_type : BoxType$

  introduces
    $copy\_box : Bx \rightarrow Bx$
    $is\_on\_box : CoordPair, Bx \rightarrow Bool$

  asserts
    $\forall\ b : Bx$
      $copy\_box(b).pos == b.pos$
      $copy\_box(b).size == b.size$
      $copy\_box(b).thickness == b.thickness$
      $copy\_box(b).starred == b.starred$
      $copy\_box(b).box\_type == b.box\_type$

Figure 4: The Box trait

and *size*, which specify the position and size of the box in the picture[1]. We assume that *pos* specifies the coordinates of the bottom left corner of a box, and that *size* specifies the width and height of a box. The *b_label* attribute will be customized for instance and constraint boxes; *thickness*, *starred* and *box_type* further define the appearance and semantics of the box.

The *Box* trait also introduces operators on boxes. The **tuple** 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 operator is [...], which in this case has the following signature:

$$[\_\_, \_\_, \_\_, \_\_, \_\_, \_\_] : CoordPair, CoordPair, BoxLabel, LineThickness, Bool, BoxType \rightarrow Bx$$

The tuple shorthand also produces 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* is the same sort as the field *field*). *b.field* returns the value of the field *field*, and *set_field(b,field_val)* returns a tuple that is equal to *b* except for the field *field*, which has value *field_val.*

In the **introduces** clause of the trait, we declare operators on the box sort *Bx*. 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. One issue

[1]The *CoordPair* sort is defined in the *BandASorts* trait as a tuple (pair) of integers

in the design of the editor was whether a copy of a box should have the same label or a default label (the empty string). Thus for *copy_box* we write equations only for the fields we require to be the same, and allow the values of the other fields (in this case *b_label*) to be specified in another trait or at the interface level. *Is_on_box* is intended to return true if the coordinate is "on" (or within some small delta of) any point on the outline of the box. We leave this operator unspecified.

Boxes in the instance language differ from those in the constraint language in two ways. First, the sorts of some of the 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 specify this difference by using the generic sort *BoxLabel* in the *Box* trait; then in the *InstanceBox* trait, we rename *BoxLabel* with the sort *Str* (for strings) and in the *ConstraintBox* trait, we rename it with the sort *BoxDesc* (for box descriptors).

The second difference is that some attributes of the box tuple are meaningful for only constraint pictures and hence are unnecessary for instance pictures. For example, *thickness* is unnecessary for instance boxes. We could avoid unnecessary field names if either Larch provided a mechanism to extend (subtype) records or we were willing to use nested records (see Section 5). However, since there are only a few of these attributes, we instead specify all of the attributes in the box sort, and then define separate operators to create instance and constraint boxes, specifying default values where necessary.

3.1.2 Arrow

The *Arrow* trait is similar to the *Box* trait, but with some additional fields to reflect these additional parameters (Figure 5). The *parity* field indicates whether an arrow is positive or negative, and *kind* determines the type of an arrow.

The fields *from_box* and *to_box* are the boxes to which the tail and head of the arrow are connected. In the Miró languages, the head and tail of every arrow must be attached to a box; actual locations (i.e., coordinates) of arrows are not important. The sort of the boxes (*Bx*) is not defined in this trait, although the intent is that it is the same as the sort *Bx* in the *Box* trait (as indicated by the **assumes** clause). The *InstanceArrow* and *ConstraintArrow* traits include the *InstanceBox* and *ConstraintBox* traits, respectively, to satisfy this assumption.

```
Arrow(Ar) : trait
    includes BandASorts
    assumes Box(Bx)

   Ar  tuple  of kind : ArrowKind, a_label : Str, parity : Parity, thickness : LineThickness,
       starred : Bool, from_box : Bx, to_box : Bx

    introduces
       copy_arrow : Ar → Ar

    asserts
       ∀ a : Ar
          copy_arrow(a).kind == a.kind
          copy_arrow(a).parity == a.parity
          copy_arrow(a).thickness == a.thickness
          copy_arrow(a).starred == a.starred
          copy_arrow(a).from_box == copy_box(a.from_box)
          copy_arrow(a).to_box == copy_box(a.to_box)
```

Figure 5: The Arrow trait

```
InstanceBox(IB) : trait
    includes String(Str for C, null_string for new), Box(IB, Str for BoxLabel)

    asserts
       ∀ ib : IB
          copy_box(ib).b_label == null_string
```

Figure 6: The InstanceBox trait

3.1.3 InstanceBox and InstanceArrow

On top of these general traits we define the specific traits for the Instance and Constraint languages. Figures 6 and 7 show the *InstanceBox* and *InstanceArrow* traits, respectively. The *InstanceBox* trait includes the *Box* trait and the *String* trait from the Larch Handbook. 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 *Box* trait gives us the sorts *IB* for instance boxes and *Str* for box labels. Recall that we used the *BoxLabel* in the *Box* trait as a "place-holder" for the sort of labels on boxes, since instance boxes have a different sort of label than constraint boxes. The trait also contains an additional equation for *copy_box* that specifies *copy_box(ib)* should result in a box whose label is an empty string.

InstanceArrow : **trait**
  includes *InstanceBox*, *Arrow*(*IA*, *IB* **for** *Bx*)

Figure 7: The InstanceArrow trait

The *InstanceArrow* trait includes both *InstanceBox* and *Arrow*. By renaming *IB* for *Bx* in *Arrow*, the sort used for the from- and to-boxes of instance arrows is *IB*, the same sort as for the instance boxes.

3.1.4 BasicPicture

```
BasicPicture(Pic) : trait

  includes Box, Set(Bx, BxSet), Arrow, Set(Ar, ArSet)

  introduces
    create_picture :→ Pic
    insert_box : Pic, Bx → Pic
    insert_arrow : Pic, Ar → Pic
    move_all_boxes : Pic, CoordPair → Pic
    copy_picture : Pic → Pic
    pic_union : Pic, Pic → Pic
    delete_box : Pic, Bx → Pic
    delete_arrow : Pic, Ar → Pic

    boxes : Pic → BxSet                                    % observers
    arrows : Pic → ArSet
    arrows_attached_to_box : Pic, Bx → ArSet
    arrows_attached_to_boxes : Pic, BxSet → ArSet
    is_on_a_box : CoordPair, Pic → Bool
    box_at : CoordPair, Pic → Bx

  asserts
    Pic generated by create_picture, insert_box, insert_arrow
    Pic partitioned by boxes, arrows
```

Figure 8: Signatures from the BasicPicture trait

A Miró picture is essentially a collection of boxes and arrows. The *BasicPicture* trait introduces the picture sort *Pic* as well as basic operators on pictures. Figure 8 shows the signatures from the *BasicPicture* trait. The **includes** statement includes traits for boxes, sets of boxes, arrows, and sets of arrows. The *Set* trait is defined in the Larch Handbook; the renaming of sort identifiers in the first *Set* trait gives us the sort *BxSet* for sets of items of sort *Bx* and all other set operators. Sets of arrows (*ArSet*) are similar. The operators that generate a picture are *create_picture*, *insert_box*, and *insert_arrow*. We define each of the remaining operators in the trait with equations in the **asserts** clause (see appendix for the complete *BasicPicture* trait). We now discuss some of these equations.

Move_all_boxes(pic,delta) moves each box in the picture *pic* by *delta*. The picture is either empty, the result of inserting a box, or the result of inserting an arrow (since these are the three generating operators). We write equations for each of these cases, as shown in Figure 9. The second equation states that if the picture argument to *move_all_boxes* is the result of inserting a box *b* in a picture *pic* then the result of *move_all_boxes* is the picture formed by changing the position of *b* and inserting that new (moved) box value into the result of *move_all_boxes(pic, delta)*. Since we only wish to move boxes and not arrows, the third equation simply inserts the (unchanged) arrow into the result of moving the boxes in *pic*.

$$\begin{array}{l}
move\_all\_boxes(create\_picture, delta) == create\_picture \\
move\_all\_boxes(insert\_box(pic, b), delta) == \\
\quad insert\_box(move\_all\_boxes(pic, delta), set\_pos(b, b.pos + delta)) \\
move\_all\_boxes(insert\_arrow(pic, a), delta) == \\
\quad insert\_arrow(move\_all\_boxes(pic, delta), a)
\end{array}$$

Figure 9: Equations for *move_all_boxes*

Copy_picture is similar to *move_all_boxes*, although with *copy_picture(pic)*, we wish to create a new picture that is a copy of *pic*. Thus, for each box and arrow in *pic*, we must insert a *copy* of that object in the resulting picture (recall that a copy of a box or arrow has different values for some of its fields).

Later traits use the *pic_union* operator to perform the higher-level copy operation. The result of *pic_union(pic1,pic2)* is a picture that contains all the boxes and arrows of *pic1* and *pic2*. Figure 10 shows the equations for *pic_union*.

$$\begin{array}{l}
pic\_union(create\_picture, pic_2) == pic_2 \\
pic\_union(insert\_box(pic, b), pic_2) == pic\_union(pic_1, insert\_box(pic_2, b)) \\
pic\_union(insert\_arrow(pic, a), pic_2) == pic\_union(pic_1, insert\_arrow(pic_2, a))
\end{array}$$

Figure 10: Equations for *pic_union* and *create_picture_sets*

With *delete_box* (and *delete_arrow*), the result is a picture with the appropriate box (or arrow) deleted. Thus, our equations check to see whether the current object is the one we wish to delete. We also leave the result of *delete_box(create_picture, b)* and *delete_arrow(create_picture, a)* unspecified, since terms that would produce error values are typically left unspecified and handled appropriately at the interface level. The equations for *delete_box* are shown in Figure 11.

```
$delete\_box(insert\_box(pic, b), b_1) ==$
    if $b = b_1$ then $pic$
    else $insert\_box(delete\_box(pic, b_1), b)$
$delete\_box(insert\_arrow(pic, a), b) ==$
    $insert\_arrow(delete\_box(pic, b), a)$
```

Figure 11: Equations for *delete_box*

Boxes, *arrows*, *arrows_attached_to_box*, *is_on_a_box*, and *box_at* are observer operators; they return information about a picture. *Boxes* and *arrows* return the set of boxes and arrows in a picture respectively; their equations are very straightforward. *Arrows_attached_to_box(pic,b)* returns the set of arrows in *pic* that are attached to *b*. *Arrows_attached_to_boxes(pic,bs)* returns the set of arrows that are attached to any box in the set *bs*; i.e., the union of the arrows attached to each box in the set. Figure 12 shows the equations for these last two operators.

```
$arrows\_attached\_to\_box(create\_picture, b_1) == \{\}$
$arrows\_attached\_to\_box(insert\_box(pic, b), b_1) ==$
    $arrows\_attached\_to\_box(pic, b_1)$
$arrows\_attached\_to\_box(insert\_arrow(pic, a), b_1) ==$
    if $(((a.from\_box) = b_1) \vee$
    $((a.to\_box) = b_1))$ then
    $insert(arrows\_attached\_to\_box(pic, b_1), a)$
    else $arrows\_attached\_to\_box(pic, b_1)$

$arrows\_attached\_to\_boxes(pic, \{\}) == \{\}$
$arrows\_attached\_to\_boxes(pic, insert(bs, b)) ==$
    $arrows\_attached\_to\_boxes(pic, bs) \cup$
    $arrows\_attached\_to\_box(pic, b)$
```

Figure 12: Equations for *arrows_attached_to_box* and *arrows_attached_to_boxes*

Is_on_a_box(cp, pic) is simlar to *arrows_attached_to_box*; it checks each box in *pic* to see whether *cp* is on that box and returns true if *cp* is on some box in *pic*. The final operator in *BasicPicture* is *box_at* (Figure 13), which returns the box at the coordinate *cp* if such a box exists.

$box\_at(cp, insert\_box(pic, b)) ==$
  if $is\_on\_box(cp, b)$ **then** b
  else $box\_at(cp, pic)$
$box\_at(cp, insert\_arrow(pic, a)) == box\_at(cp, pic)$

Figure 13: Equations for *box_at*

3.1.5 Obj and ChangeAttr

There are two characteristics of pictures that we chose to separate into individual traits to avoid an even longer picture trait. The *Obj* trait defines two new sorts that are useful in manipulating objects in a trait regardless of whether they are boxes or arrows; the *ChangeAttr* trait introduces sorts and operators to change an arbitrary attribute in a box or arrow tuple.

Since many of the picture operators are essentially the same for both boxes and arrows, we would like to operate on *objects* or sets of objects rather than having separate operators for boxes and arrows in a picture. For example, selecting an object does not depend on whether that object is a box or an arrow, so we would like a single operator to select either a box or an arrow. The *Obj* trait (Figure 14) introduces the new sorts *Ob*, a union of the box and arrow sorts, and *ObjSet*, a set of objects.

The **union of** shorthand provides coercion operators between the union sort and its component sorts. So the union declaration:

Ob **union of** $box : Bx, arrow : Ar$

produces operators with the following signatures:

$box : Bx \rightarrow Ob$
$arrow : Ar \rightarrow Ob$
$\_\_.box : Ob \rightarrow Bx$
$\_\_.arrow : Ob \rightarrow Ar$
$tag : Ob \rightarrow Ob\_tag$ % where Ob_tag **enumeration of** *box, arrow*

The operators *box* (and *arrow*) coerce a box (or arrow) to an object, *.box* (and *.arrow*) coerce an object back to a box (or arrow), and *tag* is used to determine whether an object is a box or an arrow.

The *Obj* trait also introduces operators to manipulate sets of objects. 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.

```
Obj : trait

    includes Set(Ob, ObjSet)
    assumes BasicPicture(Pic)

    Ob union of box : Bx, arrow : Ar

    introduces
        objects : Pic → ObjSet
        boxes : ObjSet → BxSet
        arrows : ObjSet → ArSet
        toggle_in : ObjSet, Ob → ObjSet

    asserts
        ∀ b : Bx, bs : BxSet, a : Ar, as : ArSet, obj : Ob, os : ObjSet, pic : Pic
            objects(create_picture) == {}
            objects(insert_box(pic, b)) == insert(objects(pic), box(b))
            objects(insert_arrow(pic, a)) == insert(objects(pic), arrow(a))

            boxes({}) == {}
            boxes(insert(os, obj)) ==
                if tag(obj) = box  then insert(boxes(os), obj.box)
                else boxes(os)

            arrows({}) == {}
            arrows(insert(os, obj)) ==
                if tag(obj) = arrow  then insert(arrows(os), obj.arrow)
                else arrows(os)

            toggle_in(os, obj) ==
                if obj ∈ os  then os − {obj}
                else os ∪ {obj}

    implies
        converts objects, boxes : ObjSet → BxSet, arrows : ObjSet → ArSet, toggle_in
```

Figure 14: The Obj trait

3.1.6 ChangeAttr

$ChangeAttr$: **trait**

assumes $Box, Arrow, Obj$

$Label$ **enumeration of** $b\_label, a\_label, thickness, starred, pos, size,$
$parity, from\_box, to\_box, kind, box\_type$

$Value$ **union of** $bool : Bool, cp : CP, box\_label : BoxLabel, str : Str,$
$b : Bx, arrow\_kind : ArrowKind, line\_thickness : LineThickness,$
$parity : Parity, bt : BoxType$

introduces
$valid\_attr : Label, Ob \rightarrow Bool$
$valid\_value : Value, Label \rightarrow Bool$
$change\_attr : Ob, Label, Value \rightarrow Ob$

Figure 15: Signatures from the ChangeAttr trait

The *ChangeAttr* trait (Figure 15) contains the specification for the *change_attr* operator, which takes an object (*obj*), field name , and value, and returns a new object that is the same as *obj* except that it has the new value for the field *fieldname.* We would like to specify *change_attr* with the following simple equation:

change_attr(obj, fieldname, value) == set_fieldname(obj, value)

But there are two problems with this equation. First, Larch does not permit "structured names." That is, we cannot put a field name (*fieldname*) in our operator names. Instead, we must use a large if-then-else statement to cover each possible field. Second, the object and value parameters to *change_attr* are union sorts (*obj* is a union of the box and arrow sorts and *value* is a union of all possible field sorts), so we must check whether the object is a box or an arrow and wrap coercion operators around *obj* and *value.*

Change_attr thus becomes one big two-layer if-then-else clause, first on object sort (box or arrow), then on label name. For each valid object/label pair, there is a clause to do the appropriate coercions and assign the value to the appropriate label. For example, if *obj* is a box and *fieldname* is *pos*, then the first case of *change_attr* is matched, and the object returned is the result (coerced to object) of changing the *pos* field of the obj (coerced to *box*) to the value *value* (coerced to *cp*). Figure 16 shows the first case of the *change_attr* operator.

```
$change\_attr(obj, fieldname, value)$ ==
    % boxes
    **if** $(tag(obj) = box)$ **then**
        **if** $(fieldname = pos)$ **then**
            $box(set\_pos(obj.box, value.cp))$
        **else** ...
```

Figure 16: Part of the equation for *change_attr*

We also define the operators *valid_attr* and *valid_value*; these define which attributes are valid for each object and which labels are of which sorts, respectively. These are used to ensure that all the values are valid before calling *change_attr* in the interface specification.

3.1.7 WFPicture

With the *BasicPicture* trait, we have introduced the picture sort, *Pic*, and the basic operators on pictures. However, in the Miró languages and editor we add an additional constraint that pictures be *well-formed.* One well-formedness condition is that the ends of each arrow be connected to boxes. There are some additional semantic well-formedness conditions (e.g., that arrows must go from user boxes to file boxes), but for this specification, we assume only the arrows-attached constraint.

```
WFPicture(Pic) : trait
    includes BasicPicture(Pic), Obj, ChangeAttr

    introduces
        extract_wf : ObjSet → Pic
        delete_objs : Pic, ObjSet → Pic
        delete_wf_box : Pic, Bx → Pic
        delete_wf_arrow : Pic, Ar → Pic
        delete_arrows : Pic, ArSet → Pic

        arrows_attached : Pic, ArSet → Bool
        arrow_attached : Pic, Ar → Bool
        well_formed : Pic → Bool
```

Figure 17: Signatures from the WellFormedPicture trait

The Well-Formed Picture trait (Figure 17) introduces operators that define well-formedness properties and new "well-formed" versions of the operators that create and modify a picture. In many cases, the result of a well-formed operator differs from the result of its non-well-formed counterpart. For example, deleting just a box may violate well-formedness, since it could result in "dangling" arrows. Hence, *delete_wf_box* must delete all attached arrows before deleting the box. Thus, we introduce an additional operator, *delete_arrows*, to delete a set of arrows from a picture, and define *delete_wf_box* to be the result of deleting all the arrows attached to the box as well as the box. Both *delete_wf_box* and *delete_wf_arrow* delete the box or arrow only if it is in the picture. The operator *delete_objs* uses *delete_wf_box* and *delete_wf_arrow* to return a picture that is the result of deleting a set of objects in a well-formed manner. For each object in the set of objects to be deleted, it checks to see whether the object is a box or an arrow and then uses the appropriate operator. The equations for *delete_objs* and *delete_wf_box* are shown in Figure 18.

Well-formedness also explains why we define the operator *move_boxes* as opposed to *move_objects* in the *BasicPicture* trait. Just moving an arrow could result in the head or tail of that arrow not touching a box, so this is not allowed; similarly, if a box to which an arrow is attached moves, the

```
$delete\_objs(pic, \{\}) == pic$
$delete\_objs(pic, insert(os, obj)) ==$
    $\textbf{if } tag(obj) = box \textbf{ then } delete\_objs(delete\_wf\_box(pic, obj.box), os)$
    $\textbf{else } delete\_objs(delete\_wf\_arrow(pic, obj.arrow), os)$

$delete\_wf\_box(pic, b) ==$
    $delete\_box(delete\_arrows(pic, arrows\_attached\_to\_box(pic, b)), b)$
```

Figure 18: Equations for *delete_objs* and *delete_wf_box*

end of the arrow must also move to remain attached, regardless of whether the arrow was selected.

The operator *extract_wf* returns a picture that is the maximal well-formed subset of a set of objects. The editor interface specification uses *extract_wf* to define the behavior of the CopyObjs operation. The set of objects to be copied describes a sub-picture, which 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*). Since *extract_wf* depends on information about other objects in *os*, we cannot define it in terms of the set constructors. Instead, we define which boxes and arrows are in the picture returned by *extract_wf* (Figure 19).

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

Figure 19: Equations for *extract_wf*

Note that although the *well_formed* operator is introduced in this trait, it is not defined. This is because instance and constraint pictures may have different notions of well-formedness. A well-formedness condition that is common to both instance and constraint pictures is that all arrows must be attached to boxes. This condition is defined in the *arrows_attached* operator. *Arrows_attached* checks a set of arrows in a picture to see that all of them are attached. It uses *arrow_attached* to check each arrow in the set. Figure 20 shows the equations for *arrows_attached* and *arrow_attached.*

3.1.8 WFInstancePic

The *WFInstancePic* trait (Figure 21) includes the *InstanceBox* and *InstanceArrow* traits, and the *WFPicture* trait with appropriate substitutions and introduces the *create_ibox* and *create_iarrow* operators. These operators are defined in the *WFInstancePic* and *WFConstraintPic* traits rather

$$
\begin{array}{l}
\mathit{arrows\_attached}(\mathit{pic}, \{\}) == \mathit{true} \\
\mathit{arrows\_attached}(\mathit{pic}, \mathit{insert}(\mathit{as}, a)) == \\
\quad \mathit{arrow\_attached}(\mathit{pic}, a) \ \wedge \mathit{arrows\_attached}(\mathit{pic}, \mathit{as}) \\
\\
\mathit{arrow\_attached}(\mathit{pic}, a) == \\
\quad (((a.\mathit{to\_box}) \in \mathit{boxes}(\mathit{pic})) \ \wedge ((a.\mathit{from\_box}) \in \mathit{boxes}(\mathit{pic})))
\end{array}
$$

Figure 20: Equations for *arrows_attached* and *arrow_attached*

$$
\begin{array}{l}
\mathit{WFInstancePic}: \ \textbf{trait} \\
\quad \textbf{includes} \ \mathit{InstanceBox}, \mathit{InstanceArrow}, \\
\quad\quad \mathit{WFPicture}(\mathit{IPic}, \mathit{create\_instance\_pic} \ \textbf{for} \ \mathit{create\_picture}, \mathit{IB} \ \textbf{for} \ \mathit{Bx}, \\
\quad\quad \mathit{IBSet} \ \textbf{for} \ \mathit{BxSet}, \mathit{Str} \ \textbf{for} \ \mathit{LabelSort}, \mathit{IA} \ \textbf{for} \ \mathit{Ar}, \mathit{IASet} \ \textbf{for} \ \mathit{ArSet}, \\
\quad\quad \mathit{IO} \ \textbf{for} \ \mathit{Ob}, \mathit{IOSet} \ \textbf{for} \ \mathit{ObjSet}) \\
\\
\quad \textbf{introduces} \\
\quad\quad \mathit{create\_ibox} : \mathit{CP}, \mathit{CP}, \mathit{Str}, \mathit{BoxType} \rightarrow \mathit{IB} \\
\quad\quad \mathit{create\_iarrow} : \mathit{IB}, \mathit{IB}, \mathit{Parity}, \mathit{Str} \rightarrow \mathit{IA} \\
\quad\quad \mathit{ambiguous} : \mathit{IPic} \rightarrow \mathit{Bool} \\
\\
\quad \textbf{asserts} \\
\quad\quad \forall \ \mathit{ipic} : \mathit{IPic}, \mathit{cp}_1, \mathit{cp}_2 : \mathit{CP}, \mathit{parity} : \mathit{Parity}, \mathit{label} : \mathit{Str}, b, b_1 : \mathit{IB}, \mathit{bt} : \mathit{BoxType} \\
\\
\quad\quad\quad \mathit{create\_ibox}(\mathit{cp}_1, \mathit{cp}_2, \mathit{label}, \mathit{bt}) == \\
\quad\quad\quad\quad [\mathit{cp}_1, \mathit{cp}_2, \mathit{label}, \mathit{thin}, \mathit{false}, \mathit{bt}] \\
\\
\quad\quad\quad \mathit{create\_iarrow}(b, b_1, \mathit{parity}, \mathit{label}) == \\
\quad\quad\quad\quad [\mathit{syn}, \mathit{label}, \mathit{parity}, \mathit{thin}, \mathit{false}, b, b_1] \\
\\
\quad\quad\quad \mathit{well\_formed}(\mathit{ipic}) == \mathit{arrows\_attached}(\mathit{ipic}, \mathit{arrows}(\mathit{ipic})) \\
\\
\quad \textbf{implies} \\
\quad\quad \textbf{converts} \ \mathit{create\_ibox}, \mathit{create\_iarrow}, \mathit{well\_formed}
\end{array}
$$

Figure 21: The WellFormedInstancePicture trait

than the *BasicPicture* trait because they require different arguments for the different languages. Namely, in the constraint language, there are more parameters for the constraint-specific fields (thickness and starred for boxes and arrows, plus kind for arrows). We specify default values for these fields in the *WFInstancePic* trait. The equation for *well_formed* specifies that a picture is well-formed if all arrows in the picture are attached. If we were to add additional well-formedness conditions, we could define an additional operator for each condition; *well_formed(pic)* would then be a conjunction of these conditions.

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 we call ***ambiguous***. Essentially a picture is ambiguous if there are both positive and negative arrows concerning a user box and a file box such that one arrow does not clearly override the others. The semantics of ambiguity are well-defined[MTW90]. Rather than reproduce them in the specification, we simply declare the operator. The Miró editor enforces well-formedness, but does not require that the pictures drawn are always unambiguous. For this reason, we cannot write *well_formed* as *all_arrows_attached(ipic)* $\wedge$ $\neg$*ambiguous(ipic)*.

3.1.9 PicUnion

Most of the editor operations will be performed on both instance and constraint pictures. For example, moving a collection of boxes behaves the same regardless of whether the picture is an instance picture or a constraint picture. Rather than duplicate the entire interface, we use the same technique as we did in the *Obj* trait for handling operators on both boxes and arrows: we introduce a union type, P, of the instance and constraint picture sorts, and operators on P that simply check the type and coerce and call the appropriate instance/constraint operators. We also introduce union sorts for each other sort that is a parameter of or is returned by these operators and is not the same for instance and constraint pictures (i.e., B, A, O, and BL). Figure 22 shows the signature for the *PicUnion* trait. By providing a union sort P and the appropriate operators, our editor can now work on either type of Miró picture.

$PicUnion$: **trait**

includes $WFInstancePic, WFConstraintPic, Set(A, ASet),$
$Set(B, BSet), Set(O, OS)$

$PicType$ **enumeration of** $inst\_pic, const\_pic$ % picture type
B **union of** $ibox : IB, cbox : CB$ % box union
A **union of** $iarrow : IA, carrow : CA$ % arrow union
O **union of** $iobj : IO, cobj : CO$ % object union
P **union of** $instance : IPic, constraint : CPic$ % picture union
BL **union of** $ilabel : Str, clabel : BoxDesc$ % box label

introduces
$\_\_.iobjset : OS \rightarrow IOSet$
$\_\_.cobjset : OS \rightarrow COSet$
$iobjset : IOSet \rightarrow OS$
$cobjset : COSet \rightarrow OS$
$box\_to\_O : B \rightarrow O$
$set\_pos : B, CoordPair \rightarrow B$
$\_\_.pos : B \rightarrow CoordPair$
$set\_size : B, CoordPair \rightarrow B$
$\_\_.size : B \rightarrow CoordPair$
$create\_picture : PicType \rightarrow P$
$insert\_box : P, B \rightarrow P$
$insert\_arrow : P, A \rightarrow P$
$copy\_picture : P \rightarrow P$
$pic\_union : P, P \rightarrow P$
$is\_on\_a\_box : CoordPair, P \rightarrow Bool$
$box\_at : CoordPair, P \rightarrow B$
$objects : P \rightarrow OS$
$boxes : OS \rightarrow BSet$
$toggle\_in : OS, O \rightarrow OS$
$valid\_attr : Label, O \rightarrow Bool$
$change\_attr : O, Label, Value \rightarrow O$
$create\_box : PicType, CoordPair, CoordPair, BL, LineThickness, Bool, BoxType \rightarrow B$
$create\_arrow :$
$PicType, B, B, Parity, Str, ArrowKind, LineThickness, Bool \rightarrow A$
$move\_all\_boxes : P, CoordPair \rightarrow P$
$delete\_objs : P, OS \rightarrow P$
$extract\_wf : PicType, OS \rightarrow P$
$well\_formed : P \rightarrow Bool$

Figure 22: Signatures from the PicUnion trait

Recall that the **union of** shorthand introduces coercion operators, as well as a *tag* operator (Section 3.1.5). As an example of how the coercion operators are used in *PicUnion*, consider the *insert_box* operator. The equations for *insert_box* (Figure 23) consist of checking to see whether the picture is an instance or constraint, performing the corresponding instance/constraint operator on the (coerced) picture, and then coercing the result back to a picture. Note that these equations assume that if p is instance then b is ibox (and similarly for constraint).

```
insert_box(p, b) ==
    if tag(p) = instance  then instance(insert_box(p.instance, b.ibox))
    else constraint(insert_box(p.constraint, b.cbox))
```

Figure 23: Equation for *insert_box*

In addition to the explicitly declared union types for boxes, arrows, objects, pictures, and box labels, we also wish to use sets of boxes, arrows and objects as unions. For example, *OS* is the sort for a set of *O* values (object unions). We need to be able to "coerce" terms of sort *OS* to terms of sort *IOSet* or *COSet*, since many of the operators take object sets as arguments. Thus, we define our own "coercion" operators for object sets and box sets, assuming that objects in a set will be either all instance objects or all constraint objects. Figure 3.1.9 shows the equations for *__.iobjset* and *iobjset*.

```
({}).iobjset == {}
(insert(os, obj)).iobjset == insert(os.iobjset, obj.iobj)
iobjset({}) == {}
iobjset(insert(ios, io)) == insert(iobjset(ios), iobj(io))
```

Figure 24: Equations for *__.iobjset* and *iobjset*

Objects and *delete_objs* are two examples of operators that use these set coercion operators; their equations are shown in Figure 25.

```
objects(p) ==
    if tag(p) = instance  then iobjset(objects(p.instance))
    else cobjset(objects(p.constraint))

delete_objs(p, os) ==
    if tag(p) = instance  then instance(delete_objs(p.instance, os.iobjset))
    else constraint(delete_objs(p.constraint, os.cobjset))
```

Figure 25: Equations for *objects* and *delete_objs*

3.2 Miró Editor

Given this model of the Miró languages, we now build a description of the Miró editor. We begin by establishing a model of the editor state at the trait level. The interface level specification then introduces the editor operations defined in terms of changes to that state. Much of the lower-level detail (e.g., mapping to mouse and keyboard actions and how text interaction works) is assumed. Many of these details are described in the informal specification of the editor found in the Editor User's Guide[Zar90].

The basic Miró editor interface is straightforward. Along the left side of the editor window, there are several sets of objects and buttons that allow the user to specify the kind of picture he or she wishes to draw (instance or constraint), the kind of object (box or arrow), and attributes of the object (e.g., arrow parity). Other buttons in the menu provide editing commands and interfaces to other tools (I/O, ambiguity checker). The main part of the editor is the *drawing region*, where the user actually draws the picture. Various mouse buttons provide additional editing commands here (e.g., right button to draw an object, left button to select an object).

3.2.1 LSL Level

```
EditorState : trait
    includes PicUnion(ObjType_Obj for O, Box_Obj for B, Arrow_Obj for A), PixelMap

  OT enumeration of box, arrow                                     % object type

  Ed tuple of
      pos : CoordPair, size : CoordPair,                           % position info
      picture : P,                                                 % graphical objects
      picture_type : PicType,                                      % picture type
      object_type : OT,                                            % object mode info
      arrow_kind : ArrowKind,
      arrow_parity : Parity,
      thickness : LineThickness,
      starred : Bool,
      selected_objs : OUS                                          % selection

  introduces
      display_window : Ed → PixelMap                               % not defined here
```

Figure 26: The EditorState trait

The *EditorState* trait (Figure 26) introduces the sort *Ed*, a tuple that we use to model the editor state. The *pos* and *size* fields indicate the location and size of the editor window on the screen. The *picture* field contains the current Miró picture, of sort P^2 (introduced in the *PicUnion* trait), and *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 (this is only interesting for constraint pictures); the rest of the attributes are self-explanatory. The only operator introduced in the *EditorState* trait is *display_window*. *Display_window* is not specified here, but is intended to be a mapping from the abstract *Ed* sort to an actual mapping of the pixels on a screen. This defines the actual appearance of the editor as a function of the editor state. The *FiniteMapping* trait is from the Larch Handbook.

3.2.2 Interface Level

The next step of the specification is to use the properties defined in the traits to specify the interfaces for the Miró editor operations. The semantics of the Generic Interface Language (GCIL) are defined as predicates on state pairs; state is a mapping of *objects* to *values*. In GCIL, we specify each operation in terms of its pre- and post-conditions (**requires** and **ensures** clauses). The **modifies** clause specifies which objects may be changed in the operation. GCIL uses call-by-reference for parameter passing.

The first part of the interface specification (Figure 27) names the object module we are specifying (*miro_editor*), and establishes correspondences between the types of objects manipulated in the interface and sorts in the trait-level specifications. For example, "**type** Box **based on** B **from** PicUnion" introduces the type Box, which has the object sort *Box_Obj* and the value sort *B*. This distinction between object sorts and value sorts allows us to specify both which objects are created and what their values are. Specifically, by the renamings in the *EditorState* trait, we define a picture to be a collection of box and arrow *objects*, so that we can use properties of *_Obj* sorts to assure the uniqueness of each box and arrow in a picture, while still being able to change the *values* of existing boxes or arrows. The **private** variable *e* is an implicit input parameter to

[2]The renamings for the object, box, and arrow sorts are needed to allow us to manipulate *objects* whose *values* are O, B, and A, respectively, at the interface level.

all operations except the initialization operation, where it is an implicit return parameter. The **invariant** specifies properties that must be true after every operation and before all operations except those named in the **initialized by** clause. The invariants state that the editor maintains the well-formedness of a picture, and that the set of selected objects is always a subset of the objects in the current picture.

```
object miro_editor
   initialized by CreateEditor

   using EditorState
   type Cp based on CoordPair from BandASorts
   type Str based on Str from String
   type Bt based on BoxType from BandASorts
   type Bl based on BL from PicUnion
   type Box based on B from PicUnion
   type Arrow based on A from PicUnion
   type ObjType based on O from PicUnion
   type ObjSet based on OS from PicUnion
   type Value based on Value from ChangeAttr
   type Label based on Label from ChangeAttr
   type Picture based on P from PicUnion
   type Editor based on Ed from EditorState

   private e : Editor

   invariant (e.selected_objs ⊆ objects(e.picture))
   invariant (well_formed(e.picture))
```

Figure 27: First part of the editor interface specification

CreateEditor (Figure 28) is the operation that gets things started. Its effect is to initialize a new editor object with the default initial modes. The **ensures** clause states that a new object is created whose value is the result of the [...] term. For a parameter or global variable e, we distinguish between the values before and after the operation with the notations e and e', respectively.

DrawBox and *DrawArrow* require that the editor's current object type be *box* and *arrow* respectively. *DrawArrow* (Figure 29) has additional clauses requiring the parameters $cp1$ and $cp2$ to be coordinates on some boxes in the picture. The effects of each operation are to create a new object (box or arrow) and insert it into the picture. The values for the parameters of the *create_arrow* operator come from either the parameters of the operation (e.g., $cp1$, $cp2$), or from the editor state

```
operation CreateEditor (posn, size : Cp)
   requires true
   ensures  newobj ($e_{obj}$) ∧
            e′ = [posn, size, create_picture(inst_pic),
                  inst_pic, box, syn, positive, thin, false,{}:OS]
```

Figure 28: *CreateEditor* operation

(e.g., *e.thickness*). The **ensures** clause requires that we create a new object, *a*, whose value is that of the *create_arrow* term, and that the value of the editor picture is the result of inserting *a* in the previous picture[3]. The **newobj** term states that *a* is a new object; it exists after the operation but did not exist before. By using **newobj** in the **ensures** clauses of all operations that create new boxes and arrows (i.e., *DrawBox*, *DrawArrow*, and *CopyObjs*), we ensure the uniqueness of all boxes and arrows in the picture.

```
operation DrawArrow (cp1, cp2 : Cp, label : Str)
   requires e.object_type = arrow ∧
            is_on_a_box(cp1,e.picture) ∧is_on_a_box(cp2,e.picture)
   modifies ($e_{obj}$)
   ensures
      ∃ a:Arrow
            newobj (a) ∧
            a!post = create_arrow(e.picture_type, box_at(cp1, e.picture), box_at(cp2, e.picture),
                  e.arrow_parity, label, e.arrow_kind, e.thickness, e.starred) ∧
            e′.picture = insert_arrow(e.picture, a)
```

Figure 29: *DrawArrow* operation

In the editor, there are three different ways to select or unselect objects: individual select/unselect, sweep select, and global unselect. Each of these correspond to an operation at the interface level and effect changes to the *selected_objs* field of the editor. *Select* takes an object, *obj*, as a parameter. If that object is already selected, it becomes unselected, otherwise it becomes selected. *GroupSelect* adds a set of objects, *os*, to the currently selected set. If an object in *os* is already selected, it remains selected. Thus, the **ensures** clause states that the value of *e.selected_objs* after the operation is the value of *e.selected_objs* before the operation unioned with *os*. *Unselect* is very straightforward:

[3]We use *e′* for the value of the editor and *a!post* for the value of the arrow because GCIL's notation differs for parameters and quantified variables.

there are no preconditions, and the effect of the operation is that the *e.selected_objs* set is empty. Figure 30 shows the specifications of all three select operations.

```
operation  Select (obj : ObjType)
   requires  (obj ∈ objects(e.picture))
   modifies  (e_obj)
   ensures   (e'.selected_objs = toggle_in(e.selected_objs, obj))

operation  GroupSelect (os : ObjSet)
   requires  (os ⊆ objects(e.picture))
   modifies  (e_obj)
   ensures   (e'.selected_objs = (e.selected_objs ∪ os))

operation  Unselect ( )
   requires  true
   modifies  (e_obj)
   ensures   (e'.selected_objs = {}:OS)
```

Figure 30: *Select*, *GroupSelect*, and *Unselect* operations

MoveBoxes(delta) moves each box in the set of selected objects by *delta* (arrows remain attached to the same boxes). For each box object *b* that is in *boxes(selected_objs)*, the value of *b* after *MoveBoxes* is the result of setting the position of the box to its previous position plus *delta*. In the editor, once one of these operations is performed, all objects are unselected. This is reflected in the second clause of the **ensures** in each operation, which states that the set of currently selected objects in the editor state after the operation is empty. The specification of *ResizeBox* is similar, but operates only on a single box; it takes a box as a parameter and requires that box be the only object selected (*box_to_O* is a coercion operator). The ensures clause changes both the position and size of the box and unselects the box. Figure 31 shows the *MoveBoxes* and *ResizeBox* operations.

In *DeleteObjs*, shown in Figure 3.2.2, the operator *delete_objs* (defined in the *PicUnion* trait) modifies *e.picture* by removing all of the selected objects. Thus, the new picture is a picture without any of those objects, and the set of selected objects becomes empty.

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 specifies a new picture object, *newpic*, whose value is the result of copying the well-formed subset of the selected objects of *e.picture*. The second clause states that each object in *newpic* must be a distinct new

```
operation MoveBoxes (delta : Cp)
   requires true
   modifies ($e_{obj}$)
   ensures
      ∀ b:Box
            (b ∈ boxes(e.selected_objs) =>
                  b!post = set_pos(b!pre, (b!pre).pos+delta)) ∧
            e′.selected_objs = {}:OS

operation ResizeBox (b : Box, pos : Cp, size : Cp)
   requires e.selected_objs = {box_to_O(b)}
   modifies ($e_{obj}$)
   ensures  b′ = set_size(set_pos(b,pos),size) ∧
            e′.selected_objs = {}:OS
```

Figure 31: *MoveBoxes* and *ResizeBox* operations

```
operation DeleteObjs ( )
   requires true
   modifies ($e_{obj}$)
   ensures  e′.picture = delete_objs(e.picture, e.selected_objs) ∧
            e′.selected_objs = {}:OS
```

Figure 32: *DeleteObjs* operation

object. *e′.picture* is then the result of combining the existing picture with *newpic*, which has been moved by *delta*. Like the other operations, *CopyObjs* also unselects all objects. Figure 33 shows the *CopyObjs* operation.

Although complicated at the trait level, the *ChangeAttribute* operation becomes one of the "cleanest" at the interface level (Figure 34). It simply requires that its arguments "type check" (i.e., that *attr* is a valid attribute of the object *o*, and that *val* is a valid value for *attr*). The **ensures** clause simply states that the value of *attr* of *o* becomes *val*.

Finally, the operation *Clear* (Figure 35) "erases" the current picture by setting *e.picture* to the empty picture (of appropriate type). The set of currently selected objects also becomes empty.

```
operation CopyObjs (delta : Cp)
   requires true
   modifies ($e_{obj}$)
   ensures
      ∃ newpic : Picture ∀ o:ObjType
               newobj (newpic) ∧
               newpic!post = copy_picture(extract_wf(e.picture,e.selected_objs)) ∧
               (o ∈ objects(newpic!post) => newobj (o)) ∧
               e'.picture = pic_union(e.picture,
                     move_all_boxes(newpic!post, delta)) ∧
               e'.selected_objs = {}:OS
```

Figure 33: *CopyObjs* operation

```
operation ChangeAttribute (o:ObjType, attr:Label, val:Value)
   requires (valid_attr(attr, o)) ∧(valid_value(val, attr))
   modifies ($o_{obj}$)
   ensures  (o' = change_attr(o,attr,val))
```

Figure 34: *ChangeAttr* operation

```
operation Clear ( )
   requires true
   modifies ($e_{obj}$)
   ensures  (e'.picture = create_picture(e.picture_type)) ∧
           (e'.selected_objs = {}:OS)
```

Figure 35: *Clear* operation

4 Making Assertions

The Larch Shared Language 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. The Generic Interface Language provides an **invariant** clause. The **invariant** specifies properties that must be true after every operation and before all operations except those named in the **initialized by** clause. This section describes some of the assertions we make about our specification through the **implies** and **invariant** clauses.

4.1 Stating Consequences

As a simple example of the kinds of implications we can write in the Larch Shared Language, consider the **implies** clause of the *BasicPicture* trait, shown in Figure 36. The first two equations in the clause assert that *copy_picture* copies all the objects in the 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. We add the equations about *copy_picture* to the **implies** clause in order to record this decision explicitly. Note that 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.

The **converts** clause in Figure 36 claims that the trait includes sufficient information to fully define the operators listed, excluding the terms listed in the **exempting** clause. This means that, if we fix the interpretations of the other operators and of terms matching *delete_box(create_picture,b)* and *delete_arrow(create_picture,a)*, there are unique interpretations for the operators listed in the **converts** clause.

Figure 37 shows the **implies** clause from the *WFPicture* trait. In *WFPicture*, we add an assumption about the appearance of our pictures. Namely, that they are "well-formed" in the sense that all arrows are attached to boxes at both ends. The trait introduces new operations that assume they are operating on well-formed pictures to produce well-formed pictures. One operator that behaves differently because of the well-formedness assumption is *delete_objs*. In the case where we are deleting a box, we must also delete all arrows attached to that box in order to preserve well-formedness. The last two equations in the **implies** clause state which boxes and arrows are removed by *delete_objs*. Although this information is available in the equations for *delete_objs*, stating this explicitly in the **implies** clause allows a specification reader to confirm his or her

```
implies
    ∀ p : Pic, delta : CoordPair
        size(boxes(copy_picture(p))) == size(boxes(p))
        size(arrows(copy_picture(p))) == size(arrows(p))
    converts move_all_boxes, copy_picture, pic_union,
        delete_box, delete_arrow, boxes,
        arrows, arrows_attached_to_box, arrows_attached_to_boxes, is_on_a_box
         exempting ∀ b : Bx, a : Ax
        delete_box(create_picture, b), delete_arrow(create_picture, a)
```

Figure 36: **implies** clause from *BasicPicture* trait

understanding and also can serve as a reminder to the specifier in the event the specification needs modification.

The other assertion in the *WFPicture* **implies** clause states that a well-formed picture sort, *Pic*, is a graph where boxes are nodes and arrows are edges with an appropriate renaming of the operators. The *Graph* trait is defined in the Larch Handbook.

```
implies
    Graph(Bx, Ar, Pic, create_picture  for empty, insert_box  for addNode,
        insert_arrow  for addEdge, boxes  for nodes, arrows  for edges)

    ∀ pic : Pic, objs : ObjSet
        boxes(delete_objs(pic, objs)) == boxes(pic) − boxes(objs)
        arrows(delete_objs(pic, objs)) ==
            arrows(pic) − (arrows(objs) ∪
                arrows_attached_to_boxes(pic, boxes(objs)))
```

Figure 37: **implies** clause from *WFPicture* trait

The *WFInstancePic* trait adds an additional assertion about well-formedness and *delete_objs*. Since we define the well-formedness operator in this trait, we also add an implication stating that *delete_objs* maintains well-formedness:

```
    ∀ pic : IPic, objs : IOSet
        (well_formed(pic) ∧ objs ⊆ objects(pic))
            ⇒ well_formed(delete_objs(pic, objs))
```

We could also extend on our assertion in *WFPicture* that a picture is a graph. Although the editor does not currently enforce it (and we do not specify it), another well-formedness condition for instance pictures is that arrows go from *user* boxes to *file* boxes. If we add this requirement to our definition of well-formedness, we can assert that a well-formed instance picture is a bipartite graph.

4.2 Stating Invariants

At the interface level, we state two invariants. An **invariant** in GCIL specifies a property that must be true after every operation and before all operations except those named in the **initialized by** clause. The first invariant states that the set of selected objects in the picture is a subset of the set of objects in the picture. It is fairly easy to see that the editor operations maintain this property. Many of the operations (*CreateEditor*, *Unselect*, *MoveBoxes*, *ResizeBox*, *DeleteObjs*, *CopyObjs*, and *Clear*) specify in their **ensures** clauses that $e'.selected = \{\}$. *DrawBox*, *DrawArrow*, and *ChangeAttribute* do not change the set of selected objects, and either increase or do not change the set of objects in the picture. And for *Select* and *GroupSelect*, if the invariant holds before the operation, the **requires** clause ensures that it holds after. The selection invariant will be useful in proving the second invariant, which states that the editor picture is always well-formed. Many of the other Miró tools assume that the picture they are manipulating is well-formed; thus, this is an important property.

invariant (e.selected_objs $\subseteq$ objects(e.picture))
invariant (well_formed(e.picture))

5 Discussion

In this section, we discuss two limitations of Larch that arose out of this specification exercise, some general lessons learned from having done the specification after the implementation, and related work.

5.1 Why Tuples?

One important design issue was how to represent each of the graphical objects of the Miró languages. What are the essential characteristics of a box? It has such attributes as location, size and type. Operations on boxes include creating new boxes, changing attributes of a box, copying a box, and checking to see whether a coordinate is on a box. Larch's **tuple** shorthand provides much of the functionality we want (the attributes of a box are fields of a tuple, and tuple operators include a generator and operators to change each field). It is possible that knowing how the editor was implemented influenced this choice, but we note that other graphical specifications also use tuples. For example, the extended example in [GGH90] defines a window sort as a tuple.

One drawback to using tuples is that Larch does not permit "subtyping" of tuples. For example, a more general approach to specifying Miró objects would have been to define traits for a base set of generic graphical objects (box, text, line, arrow, ...), and then build the Miró object traits by including the appropriate generic objects. But since we cannot specify that the Miró box sort tuple is a subtype of the generic box sort tuple with the additional fields *type* and *label*, we would instead have to nest tuples. The Miró box would be a tuple that has a generic box as one of its fields (say, *box*)). Then, to talk about the location of a Miró box, we would have to say *b.box.location.*

A second example of the usefulness of tuple subtyping is in defining instance and constraint boxes. Some of the fields in the box trait are only meaningful for constraint boxes (e.g., *thick* and *starred*). Yet we could not define an instance box sort as a subtype of the box sort with those additional fields. Instead, we included those fields in the general Miró box sort, even though they are meaningless for instance boxes.

5.2 Union Sorts

Another difficulty in the specification was in moving from operators on specific sorts to operators on a more general sort (i.e., from boxes and arrows to objects, and from instance pictures and constraint pictures to pictures). There are places in the specification where it does not matter whether an object is a box or an arrow, we just want to delete it (and similarly for instance/constraint, since the editor can operate on either type of picture). So, if the operators *copy_picture:* $P \rightarrow P$, *copy_picture:* *IPic* $\rightarrow$ *IPic* and *copy_picture:* *CPic* $\rightarrow$ *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 (i.e., allow operators to be overloaded). The solution we use in this specification is to define union sorts for objects and pictures. This makes for two somewhat tedious traits (*Obj* and *PicUnion*), but a fairly "readable" specification overall (the alternative was to add explicit sort checking in the interface specification in order to call the correct operator). For *copy_picture(pic)*, this means we must first determine whether *pic* is an instance (or a constraint) picture and then coerce it 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.

Another variation of this problem arose from specifying the *change_attr* operator. Intuitively, *change_attr* should take as arguments an object, an attribute and a value, and return a new object that differs from the old only in the value of the attribute. But different objects have different attributes, and different attributes have different sorts, so *change_attr* turns out to be a long nested if-then-else statement that uses union types and coercion (we discuss this in detail in Section 3.1.6).

5.3 Lessons Learned

We began writing formal specifications in parallel with designing the Miró languages and designing and implementing of the editor. We wrote three major versions of the specification where the last version (this one) was written after the implmentation was running. The current version itself went through at least eight 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 clearly evident. We found both to be true in our case.

Having already implemented a version of the editor before completing the specification, we had that model of the languages and editor in mind, which led to some very implementation-biased versions of the 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 the implementation details that we removed from the specification is the correlation between mouse actions and interface procedures. One of the issues in specifying the editor is dealing with interactive user input. What is the best way to model a mouse? What level of detail should be represented? An initial specification modeled "mouse clicks" and a state-transition diagram that kept track of the current state. For each state (e.g., Waiting, Moving, Resizing) and each mouse action (e.g., RightUp, MiddleDown), there was a rule about what the effects of the mouse action would be. This was a bit too much detail for this specification, since we did not want to commit a particular operation to a particular mouse button (or series of mouse operations). Thus, in the current specification, we assume that the mouse operations produce some coordinate information, which is used to produce calls to the operations specified in the interface. The Miró Editor User's Guide[Zar90] defines the correlation between particular mouse operations and the operations defined in the interface level.

Another example of initial bias toward the implementation 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 constraints on arrow objects. So in a previous version of the specification, the arrow sort A had fields for the coordinate positions of the head and tail of the arrow. The *WFPicture* trait defined an operator called *adjust_arrows*, which took a picture and a box and adjusted those fields based on the position of the boxes at the head and tail of the arrow. But at a more abstract level, the coordinate information for an arrow is not important (since the boxes it is attached to are also fields of an arrow, and the coordinates can be derived from them), and it is more accurate to define the well-formedness property at the trait level but enforce it at the interface level, as the current specification does.

The second observation about doing the specification after the implementation is that doing the reverse would probably have resulted in a better implementation. As partially illustrated in the previous examples, the specification helped us to understand the languages and editor more clearly, and to realize the essential characteristics of the languages and editor. Particularly in the case of the editor, it is our further belief that if the work had instead progressed in the opposite order, this understanding would have helped to create a "cleaner" editor that would have been easier to implement and later modify.

One example to support this argument is our experience implementing multiple selection in the editor. The initial implementation of the editor allowed at most one object to be selected at any time. We extended the editor to allow selecting multiple objects while we were also working on the specification. So we wrote an informal specification of how the set of selected objects would be affected by each editor operation *before* we implemented multiple selection. The result was an implementation of multiple selection that was clean, consistent and relatively easy to add to the editor.

This specification was also useful for exploring the power and limitations of Larch as a tool for specifying a large interactive graphical application. We discovered two limitations that made the specification more difficult: the lack of tuple subtyping and operator overloading (Sections 5.1 and 5.2). But overall, we found Larch to be powerful enough to express the properties of our application. The tools (LC and the GCIL checker) proved invaluable in locating the minor errors in our specification through sort- and type-checking. We also found that writing such a specification taught us to think about program behavior in a more structured and property-oriented way that we hope will be useful in future projects.

5.4 Related Work

There are several Larch Shared Language specifications to read and learn from. The largest collection of Larch traits of which we are aware is the Larch Handbook[GHW85]. The extended example in[GGH90] specifies some traits for a simple windowing system. Larch (both trait and interface levels) has also been used to specify properties of objects in a transaction-based distributed system ([Win88] and [Ler91]).

In our specification, we assumed details about how Miró pictures are represented on the screen and what keyboard and mouse inputs activate the specified procedures. These are difficult problems, but have been addressed by others: basic properties of window systems have been specified in both Larch [GGH90] and Z [Bow89]; basic picture primitives as well as a method for specifying user interaction are defined in [Mal82]; a display-based text editor is specified in [Suf82]. Thus, we chose to focus at the next level of detail: properties of Miró pictures and the the editor. We have not seen any other specifications that deal with an interactive graphical application.

6 Further Work and Conclusions

The specification as is depends on the structure of Miró objects and the particular behavior of the editor. There are several axes along which it could be generalized. One possibility would be to move from specific Miró graphical objects to more generic objects, such as boxes, lines, and text, which could then be composed and extended to generate the particular objects for Miró. We could use these same graphical object specifications as a general library on top of which many different graphical languages could be specified.

A second issue is how the specification would change if we were to add or change features of the language or editor. Adding characteristics to objects should be fairly easy, since that is mainly a matter of extending the appropriate tuple. But in the current *Arrow* trait, the only way to identify ends of an arrow is by the boxes to which it is attached (there are no arrow coordinates). What if we change the editor to allow unattached arrows? How difficult would it be to include additional conditions in our definition of well-formed? How would our definitions of additional characteristics of Miró pictures such as ambiguity and legality translate into Larch?

Although we have checked all parts of the specification shown in this paper with the LSL and GCIL checkers, we have not tried using the Larch Prover (LP) to prove any of our assertions. This would serve both to demonstrate the correctness of parts of this specification and to explore the powers and limitations of LP. A final "link" would be to show that the specification models the actual editor implementation.

In conclusion, this specification has been an instructive exercise. We have learned more about Miró and we now have a formal definition of both the language and editor. Additionally, we have explored the Larch languages as a tool for specification.

I am particularly grateful to my advisor, Jeannette Wing, for her patience and guidance on this work. I would also like to thank Rick Lerner for the many conversations about LSL and GCIL, and Allan Heydon, Mark Maimone and Doug Tygar for their extremely helpful comments on earlier drafts. I am also indebted to Brad Myers and the rest of the Garnet group for help in implementing the Miró editor.

A Miró Editor Specification

% ——————————
% BOX AND ARROW ATTRIBUTE SORTS
% ——————————
BandASorts : **trait**

includes *Integer*

CoordPair **tuple of** $x : Int, y : Int$ % Coordinate Point
LineThickness **enumeration of** *thin*, *thick* % Line Thickness
BoxType **enumeration of** *user*, *file*, *unknown* % Box Type
Parity **enumeration of** *positive*, *negative* % Arrow Parity
ArrowKind **enumeration of** *syn*, *sem*, *con* % Arrow Kind

introduces
$\_\_ + \_\_ : CoordPair, CoordPair \rightarrow CoordPair$

asserts $\forall\ x_1, y_1 : Int, cp_1, cp_2 : CoordPair$
$cp_1 + cp_2 == [(cp_1.x) + (cp_2.x), (cp_1.y) + (cp_2.y)]$

% ——————————
% BOX
% ——————————
Box(Bx) : **trait**
includes *BandASorts*

% pos is bottom left corner of box. size is width and height.
Bx **tuple of** *pos* : *CoordPair*, *size* : *CoordPair*, *b_label* : *BoxLabel*, *thickness* : *LineThickness*,
starred : *Bool*, *box_type* : *BoxType*

introduces
$copy\_box : Bx \rightarrow Bx$
$is\_on\_box : CoordPair, Bx \rightarrow Bool$

asserts
$\forall\ b : Bx$
% Specify Copy_Box with rules for each field.
% b_label intentionally not specified here.
$copy\_box(b).pos == b.pos$
$copy\_box(b).size == b.size$
$copy\_box(b).thickness == b.thickness$
$copy\_box(b).starred == b.starred$
$copy\_box(b).box\_type == b.box\_type$

```
% ————————
% INSTANCE BOX
% ————————
InstanceBox(IB) : trait
    includes String(Str for C, null_string for new), Box(IB, Str for BoxLabel)

    asserts
      ∀ ib : IB
          % Additional rules for Copy_Box:
          % Do not copy label, instead set to empty string.
          copy_box(ib).b_label == null_string


% ————————
% CONSTRAINT BOX
% ————————
ConstraintBox(CB) : trait
    includes Box(CB, BoxDesc for BoxLabel)


% ————————
% ARROW
% ————————
Arrow(Ar) : trait
    includes BandASorts
    assumes Box(Bx)

    Ar tuple of kind : ArrowKind, a_label : Str, parity : Parity, thickness : LineThickness,
       starred : Bool, from_box : Bx, to_box : Bx

    introduces
       copy_arrow : Ar → Ar

    asserts
      ∀ a : Ar
          % Copy Arrow:
          % a_label intentionally not specified here.
          copy_arrow(a).kind == a.kind
          copy_arrow(a).parity == a.parity
          copy_arrow(a).thickness == a.thickness
          copy_arrow(a).starred == a.starred
          copy_arrow(a).from_box == copy_box(a.from_box)
          copy_arrow(a).to_box == copy_box(a.to_box)
```

% ————————
% INSTANCE ARROW
% ————————

InstanceArrow : **trait**
    includes *InstanceBox*, *Arrow*(*IA*, *IB* **for** *Bx*)

% ————————
% CONSTRAINT ARROW
% ————————

ConstraintArrow : **trait**
    includes *ConstraintBox*, *Arrow*(*CA*, *CB* **for** *Bx*)

% ————————
% BASIC PICTURE
% ————————

BasicPicture(*Pic*) : **trait**

    includes *Box*, *Set*(*Bx*, *BxSet*), *Arrow*, *Set*(*Ar*, *ArSet*)

    introduces
        create_picture :→ *Pic*
        insert_box : *Pic*, *Bx* → *Pic*
        insert_arrow : *Pic*, *Ar* → *Pic*
        move_all_boxes : *Pic*, *CoordPair* → *Pic*
        copy_picture : *Pic* → *Pic*
        pic_union : *Pic*, *Pic* → *Pic*
        delete_box : *Pic*, *Bx* → *Pic*
        delete_arrow : *Pic*, *Ar* → *Pic*

        boxes : *Pic* → *BxSet*   % obse
        arrows : *Pic* → *ArSet*
        arrows_attached_to_box : *Pic*, *Bx* → *ArSet*
        arrows_attached_to_boxes : *Pic*, *BxSet* → *ArSet*
        is_on_a_box : *CoordPair*, *Pic* → *Bool*
        box_at : *CoordPair*, *Pic* → *Bx*

    asserts

        Pic **generated by** *create_picture*, *insert_box*, *insert_arrow*
        Pic **partitioned by** *boxes*, *arrows*

        ∀ pic, pic_1, pic_2 : *Pic*, *cp*, *delta* : *CoordPair*, b, b_1 : *Bx*, a, a_1 :
            Ar, *bs* : *BxSet*, *as* : *ArSet*

            % Move All Boxes in the picture by delta.

```
$move\_all\_boxes(create\_picture, delta) == create\_picture$
$move\_all\_boxes(insert\_box(pic, b), delta) ==$
    $insert\_box(move\_all\_boxes(pic, delta), set\_pos(b, b.pos + delta))$
$move\_all\_boxes(insert\_arrow(pic, a), delta) ==$
    $insert\_arrow(move\_all\_boxes(pic, delta), a)$


% Copy Picture: copy each object.
$copy\_picture(create\_picture) == create\_picture$
$copy\_picture(insert\_box(pic, b)) ==$
    $insert\_box(copy\_picture(pic), copy\_box(b))$
$copy\_picture(insert\_arrow(pic, a)) ==$
    $insert\_arrow(copy\_picture(pic), copy\_arrow(a))$

% Union of two pictures.
$pic\_union(create\_picture, pic_2) == pic_2$
$pic\_union(insert\_box(pic, b), pic_2) == pic\_union(pic_1, insert\_box(pic_2, b))$
$pic\_union(insert\_arrow(pic, a), pic_2) == pic\_union(pic_1, insert\_arrow(pic_2, a))$


% Deleting a box or arrow is exempt for empty pictures.
$delete\_box(insert\_box(pic, b), b_1) ==$
     if $b = b_1$ then $pic$
     else $insert\_box(delete\_box(pic, b_1), b)$
$delete\_box(insert\_arrow(pic, a), b) ==$
    $insert\_arrow(delete\_box(pic, b), a)$

$delete\_arrow(insert\_arrow(pic, a), a_1) ==$
     if $a = a_1$ then $pic$
     else $insert\_arrow(delete\_arrow(pic, a_1), a)$
$delete\_arrow(insert\_box(pic, b), a) ==$
    $insert\_box(delete\_arrow(pic, a), b)$

% Return the set of boxes and set of arrows in picture.
$boxes(create\_picture) == \{\}$
$boxes(insert\_box(pic, b)) == insert(boxes(pic), b)$
$boxes(insert\_arrow(pic, a)) == boxes(pic)$

$arrows(create\_picture) == \{\}$
$arrows(insert\_box(pic, b)) == arrows(pic)$
$arrows(insert\_arrow(pic, a)) == insert(arrows(pic), a)$

% Arrows_Attached_To_Box:
% Find all arrows attached to a box – look at each arrow in
% picture to see whether to or from b.
$arrows\_attached\_to\_box(create\_picture, b_1) == \{\}$
$arrows\_attached\_to\_box(insert\_box(pic, b), b_1) ==$
```

```
        $arrows\_attached\_to\_box(pic, b_1)$
    $arrows\_attached\_to\_box(insert\_arrow(pic, a), b_1) ==$
        **if** $(((a.from\_box) = b_1) \vee$
        $((a.to\_box) = b_1))$ **then**
        $insert(arrows\_attached\_to\_box(pic, b_1), a)$
        **else** $arrows\_attached\_to\_box(pic, b_1)$


    % Arrows_Attached_To_Boxes:
    % Find all arrows attached to a set of boxes – union of sets of
    % arrows attached to each box.
    $arrows\_attached\_to\_boxes(pic, \{\}) == \{\}$
    $arrows\_attached\_to\_boxes(pic, insert(bs, b)) ==$
        $arrows\_attached\_to\_boxes(pic, bs) \cup$
        $arrows\_attached\_to\_box(pic, b)$

    % Is_On_A_Box returns true if there exists a box b in the picture.
    % such that is_on_box(cp,b) is true.
    $is\_on\_a\_box(cp, create\_picture) == false$
    $is\_on\_a\_box(cp, insert\_box(pic, b)) ==$
        $is\_on\_box(cp, b) \vee is\_on\_a\_box(cp, pic)$
    $is\_on\_a\_box(cp, insert\_arrow(pic, a)) == is\_on\_a\_box(cp, pic)$

    % Box_at returns the box b such that is_on_box(cp,b) is true
    % if such a box exists.
    $box\_at(cp, insert\_box(pic, b)) ==$
        **if** $is\_on\_box(cp, b)$ **then** $b$
        **else** $box\_at(cp, pic)$
    $box\_at(cp, insert\_arrow(pic, a)) == box\_at(cp, pic)$


**implies**
  $\forall\ p : Pic, delta : CoordPair$
    % Copy_pic copies _all_ objects (this won't necessarily be true at wf-pic level)
    $size(boxes(copy\_picture(p))) == size(boxes(p))$
    $size(arrows(copy\_picture(p))) == size(arrows(p))$

  **converts** $move\_all\_boxes, copy\_picture, pic\_union,$
    $delete\_box, delete\_arrow, boxes,$
    $arrows, arrows\_attached\_to\_box, arrows\_attached\_to\_boxes, is\_on\_a\_box$
     **exempting** $\forall\ b : Bx, a : Ar$
    $delete\_box(create\_picture, b), delete\_arrow(create\_picture, a)$



% ————————
% OBJECT
% ——————————
Obj : trait

    includes Set(Ob, ObjSet)
    assumes BasicPicture(Pic)

    Ob  union  of box : Bx, arrow : Ar

    introduces
        objects : Pic → ObjSet
        boxes : ObjSet → BxSet
        arrows : ObjSet → ArSet
        toggle_in : ObjSet, Ob → ObjSet

    asserts
        ∀ b : Bx, bs : BxSet, a : Ar, as : ArSet, obj : Ob, os : ObjSet, pic : Pic
            % Form the set of all objects by recursing through the sets of boxes and arrows.
            objects(create_picture) == {}
            objects(insert_box(pic, b)) == insert(objects(pic), box(b))
            objects(insert_arrow(pic, a)) == insert(objects(pic), arrow(a))

            % Boxes/Arrows: extract box and arrow sets from set of objects.
            boxes({}) == {}
            boxes(insert(os, obj)) ==
                if tag(obj) = box  then insert(boxes(os), obj.box)
                else boxes(os)

            arrows({}) == {}
            arrows(insert(os, obj)) ==
                if tag(obj) = arrow  then insert(arrows(os), obj.arrow)
                else arrows(os)

            % Toggle membership in set of objects (used in keeping set of
            % selected objects).
            toggle_in(os, obj) ==
                if obj ∈ os  then os − {obj}
                else os ∪ {obj}

    implies
        converts objects, boxes : ObjSet → BxSet, arrows : ObjSet → ArSet, toggle_in


% ——————————
% CHANGE ATTRIBUTE
% ——————————
ChangeAttr : trait
```

```
assumes $Box, Arrow, Obj$

$Label$ enumeration of $b\_label, a\_label, thickness, starred, pos, size,$
    $parity, from\_box, to\_box, kind, box\_type$

$Value$ union of $bool : Bool, cp : CoordPair, box\_label : BoxLabel, str : Str,$
    $b : Bx, arrow\_kind : ArrowKind, line\_thickness : LineThickness,$
    $parity : Parity, bt : BoxType$

introduces
    $valid\_attr : Label, Ob \rightarrow Bool$
    $valid\_value : Value, Label \rightarrow Bool$
    $change\_attr : Ob, Label, Value \rightarrow Ob$

asserts
    $\forall\ fieldname : Label, value : Value, obj : Ob$
        % Valid Attribute if label is an attribute in object
        % valid attrs for box = ( pos, size, b_label, thickness,starred, box_type)
        % valid attrs for arrow = (a_label, parity,from_box,
        % to_box, kind, thickness, starred)
        $valid\_attr(fieldname, obj) ==$
            if $tag(obj) = box$ then
            $((fieldname = pos) \vee (fieldname = size) \vee$
            $(fieldname = b\_label) \vee (fieldname = thickness) \vee$
            $(fieldname = starred) \vee (fieldname = box\_type))$
            else                                                    % tag(obj)=arrow
            $((fieldname = a\_label) \vee (fieldname = parity)$
            $\vee (fieldname = from\_box) \vee (fieldname = to\_box) \vee (fieldname = kind)$
            $\vee (fieldname = thickness) \vee (fieldname = starred))$

        % Valid Value if value is correct type for label
        $valid\_value(value, fieldname) ==$
            % valid labels for bool = (starred)
            if $tag(value) = bool$ then
            $(fieldname = starred)$
            % valid labels for line_thickness = thickness
            else if $tag(value) = line\_thickness$ then
            $(fieldname = thickness)$
            % valid labels for parity = parity
            else if $tag(value) = parity$ then
            $(fieldname = parity)$
            % valid labels for cp = (pos, size)
            else if $tag(value) = cp$ then
            $((fieldname = pos) \vee (fieldname = size))$
            % valid labels for box_label = (b_label)
            else if $tag(value) = box\_label$ then
    (($fieldname = b\_label$))
    % valid labels for str = (a_label)
     else if $tag(value) = str$ then
    (($fieldname = a\_label$))
    % valid labels for b = (from_box, to_box)
     else if $tag(value) = b$ then
    (($fieldname = from\_box$) $\vee$ ($fieldname = to\_box$))
    % valid labels for arrow_kind = (kind)
     else if $tag(value) = arrow\_kind$ then
    ($fieldname = kind$)
    % valid labels for bt = (box_type)
     else                                                        % tag(value) = bt
    ($fieldname = box\_type$)

% Change Attribute assumes valid arguments. Just huge case
% statement, first on object type, then on label
$change\_attr(obj, fieldname, value)$ ==
    % boxes
     if ($tag(obj) = box$) then
     if ($fieldname = pos$) then
    $box(set\_pos(obj.box, value.cp))$
     else if ($fieldname = size$) then
    $box(set\_size(obj.box, value.cp))$
     else if ($fieldname = b\_label$) then
    $box(set\_b\_label(obj.box, value.box\_label))$
     else if ($fieldname = thickness$) then
    $box(set\_thickness(obj.box, value.line\_thickness))$
     else if ($fieldname = starred$) then
    $box(set\_starred(obj.box, value.bool))$
     else                                           % (fieldname=box_type) then
    $box(set\_box\_type(obj.box, value.bt))$
     else                                                        % tag(obj)=arrow
     if ($fieldname = kind$) then
    $arrow(set\_kind(obj.arrow, value.arrow\_kind))$
     else if ($fieldname = a\_label$) then
    $arrow(set\_a\_label(obj.arrow, value.str))$
     else if ($fieldname = parity$) then
    $arrow(set\_parity(obj.arrow, value.parity))$
     else if ($fieldname = thickness$) then
    $arrow(set\_thickness(obj.arrow, value.line\_thickness))$
     else if ($fieldname = starred$) then
    $arrow(set\_starred(obj.arrow, value.bool))$
     else if ($fieldname = from\_box$) then
    $arrow(set\_from\_box(obj.arrow, value.b))$
     else                                            % (fieldname=to_box) then
    $arrow(set\_to\_box(obj.arrow, value.b))$
    implies
        converts valid_attr, valid_value


% ————————
% WELL-FORMED PICTURE
% ————————
WFPicture(Pic) : trait
    includes BasicPicture(Pic), Obj, ChangeAttr

    introduces
        extract_wf : ObjSet → Pic
        delete_objs : Pic, ObjSet → Pic
        delete_wf_box : Pic, Bx → Pic
        delete_wf_arrow : Pic, Ar → Pic
        delete_arrows : Pic, ArSet → Pic

        arrows_attached : Pic, ArSet → Bool
        arrow_attached : Pic, Ar → Bool
        well_formed : Pic → Bool                    % defined in WF instance/constraint

    asserts
        ∀ pic : Pic, b : Bx, a : Ar, as : ArSet, obj : Ob, os : ObjSet

            % Extract WF: keep all boxes and arrows in os whose boxes are also in os.
            boxes(extract_wf(os)) == boxes(os)
            a ∈ arrows(extract_wf(os)) ==
                (a ∈ arrows(os))∧
                ((a.to_box) ∈ boxes(os))∧
                ((a.from_box) ∈ boxes(os))

            % Delete a set of objects.
            delete_objs(pic, {}) == pic
            delete_objs(pic, insert(os, obj)) ==
                if tag(obj) = box then
                delete_objs(delete_wf_box(pic, obj.box), os)
                else                                              % is an arrow
                delete_objs(delete_wf_arrow(pic, obj.arrow), os)

            % When deleting box, delete all attached arrows first.
            % If box is not in picture, just return picture.
            delete_wf_box(pic, b) ==
                if b ∈ boxes(pic) then
delete_box(delete_arrows(pic, arrows_attached_to_box(pic, b)), b)
                else pic

            % Check to see if arrow is in picture first.
            delete_wf_arrow(pic, a) ==
                if a ∈ arrows(pic)  then delete_arrow(pic, a)
                else pic

            % Delete a set of arrows.
            delete_arrows(pic, {}) == pic
            delete_arrows(pic, insert(as, a)) ==
                delete_arrows(delete_wf_arrow(pic, a), as)

            % arrows_attached(pic,as) iff each arrow in as (set of arrows) is attached.
            arrows_attached(pic, {}) == true
            arrows_attached(pic, insert(as, a)) ==
                arrow_attached(pic, a) ∧ arrows_attached(pic, as)

            % An arrow is attached in a picture if both of its boxes are in the picture.
            arrow_attached(pic, a) ==
                (((a.to_box) ∈ boxes(pic)) ∧ ((a.from_box) ∈ boxes(pic)))

    implies
        % A well-formed picture is a graph.
        Graph(Bx, Ar, Pic, create_picture  for empty, insert_box  for addNode,
        insert_arrow  for addEdge, boxes  for nodes, arrows  for edges)

        ∀ pic : Pic, objs : ObjSet
            % The result of delete is a picture with all boxes in objs deleted,
            % and all arrows attached to boxes in objs, as well as all arrows
            % in objs, deleted.
            boxes(delete_objs(pic, objs)) == boxes(pic) − boxes(objs)
            arrows(delete_objs(pic, objs)) ==
                arrows(pic) − (arrows(objs)∪
                arrows_attached_to_boxes(pic, boxes(objs)))

        converts arrows_attached, arrow_attached


% ——————————
% WELL-FORMED INSTANCE PICTURE
% ——————————
WFInstancePic :  trait
    includes InstanceBox, InstanceArrow,
        WFPicture(IPic, create_instance_pic  for create_picture, IB  for Bx,
        IBSet  for BxSet, Str  for BoxLabel, IA  for Ar, IASet  for ArSet,
        IO  for Ob, IOSet  for ObjSet)

    introduces
        create_ibox : CoordPair, CoordPair, Str, BoxType → IB
        create_iarrow : IB, IB, Parity, Str → IA
    ambiguous : IPic → Bool

  asserts
    ∀ ipic : IPic, cp₁, cp₂ : CoordPair, parity : Parity, label : Str, b, b₁ : IB, bt : BoxType

        % Default values for thickness (thin), starred (false).
        create_ibox(cp₁, cp₂, label, bt) ==
            [cp₁, cp₂, label, thin, false, bt]

        % default values for thick (thin), starred (false), and kind (syn)
        create_iarrow(b, b₁, parity, label) ==
            [syn, label, parity, thin, false, b, b₁]

        % Well Formed.
        well_formed(ipic) == arrows_attached(ipic, arrows(ipic))

  implies

    ∀ pic : IPic, objs : IOSet

        % delete_objs maintains well-formedness
        (well_formed(pic) ∧ objs ⊆ objects(pic))
        ⇒ well_formed(delete_objs(pic, objs))

     converts create_ibox, create_iarrow, well_formed




% ————————
% WELL-FORMED CONSTRAINT PICTURE
% ————————
WFConstraintPic : trait
   includes ConstraintBox, ConstraintArrow,
      WFPicture(CPic,
      create_constraint_pic for create_picture,
      CB for Bx, CBSet for BxSet, BoxDesc for BoxLabel,
      CA for Ar, CASet for ArSet, CO for Ob, COSet for ObjSet)

   introduces
      create_cbox : CoordPair, CoordPair, BoxDesc, LineThickness, Bool, BoxType → CB
      create_carrow :
      CB, CB, Parity, Str, ArrowKind, LineThickness, Bool → CA

   asserts
     ∀ cpic : CPic, cp₁, cp₂ : CoordPair, parity : Parity, thickness : LineThickness,
            starred : Bool, kind : ArrowKind, boxlabel : BoxDesc,
            arrowlabel : Str, b, b1 : CB, bt : BoxType

            create_cbox(cp1, cp2, boxlabel, thickness, starred, bt) ==
                [cp1, cp2, boxlabel, thickness, starred, bt]

            create_carrow(b, b1, parity, arrowlabel, kind, thickness, starred) ==
                [kind, arrowlabel, parity, thickness, starred, b, b1]

            % Well Formed. Right now, just require that all arrows are attached.
            well_formed(cpic) == arrows_attached(cpic, arrows(cpic))

        implies
            converts create_cbox, create_carrow, well_formed


% ————————
% PICTURE UNION
% ————————

PicUnion : trait
        includes WFInstancePic, WFConstraintPic, Set(A, ASet),
            Set(B, BSet), Set(O, OS)

    PicType  enumeration  of inst_pic, const_pic                    % picture type

    B  union  of ibox : IB, cbox : CB                               % box union
    A  union  of iarrow : IA, carrow : CA                           % arrow union
    O  union  of iobj : IO, cobj : CO                               % object union
    P  union  of instance : IPic, constraint : CPic                 % picture union
    BL  union  of ilabel : Str, clabel : BoxDesc                    % box label

        introduces
            % explicit coersion functions for sets of objects
            % OS ~ union of iobjset:IOSet, cobjset:COSet
            __.iobjset : OS → IOSet
            __.cobjset : OS → COSet
            iobjset : IOSet → OS
            cobjset : COSet → OS

            box_to_O : B → O

            set_pos : B, CoordPair → B
            __.pos : B → CoordPair
            set_size : B, CoordPair → B
            __.size : B → CoordPair
```

% operators from BasicPicture
$create\_picture : PicType \rightarrow P$
$insert\_box : P, B \rightarrow P$
$insert\_arrow : P, A \rightarrow P$
$copy\_picture : P \rightarrow P$
$pic\_union : P, P \rightarrow P$
$is\_on\_a\_box : CoordPair, P \rightarrow Bool$
$box\_at : CoordPair, P \rightarrow B$

% operators from Picture
$objects : P \rightarrow OS$
$boxes : OS \rightarrow BSet$
$toggle\_in : OS, O \rightarrow OS$
$valid\_attr : Label, O \rightarrow Bool$
$change\_attr : O, Label, Value \rightarrow O$

$create\_box : PicType, CoordPair, CoordPair, BL, LineThickness, Bool, BoxType \rightarrow B$
$create\_arrow :$
$PicType, B, B, Parity, Str, ArrowKind, LineThickness, Bool \rightarrow A$

% operators from WFPicture
$move\_all\_boxes : P, CoordPair \rightarrow P$
$delete\_objs : P, OS \rightarrow P$
$extract\_wf : PicType, OS \rightarrow P$
$well\_formed : P \rightarrow Bool$

asserts
OS **generated by** $iobjset, cobjset$

$\forall\ pt : PicType, p, p_1 : P, b, b_1 : B, a : A, cp, cp_1 : CoordPair, bs : BSet, as : ASet,$
$obj : O, os : OS, labelvar : Label, value : Value, bl : BL, parity : Parity,$
$thickness : LineThickness, starred : Bool, str : Str, kind : ArrowKind,$
$bt : BoxType, ib : IB, ibs : IBSet, cb : CB, cbs : CBSet, ios : IOSet, io : IO,$
$cos : COSet, co : CO$

% explicit coersion functions to convert a set of objects to a set
% of instance or constraint objects
$(\{\}).iobjset == \{\}$
$(insert(os, obj)).iobjset == insert(os.iobjset, obj.iobj)$
$(\{\}).cobjset == \{\}$
$(insert(os, obj)).cobjset == insert(os.cobjset, obj.cobj)$
$iobjset(\{\}) == \{\}$
$iobjset(insert(ios, io)) == insert(iobjset(ios), iobj(io))$
$cobjset(\{\}) == \{\}$
$cobjset(insert(cos, co)) == insert(cobjset(cos), cobj(co))$

% explicit coersion from a (union) box to a (union) object

```
$box\_to\_O(b) ==$
    if $tag(b) = ibox$  then $iobj(box(b.ibox))$
    else $cobj(box(b.cbox))$

% explicit "tuple" operators for box union sort.
$set\_pos(b, cp) ==$
    if $tag(b) = ibox$  then $ibox(set\_pos(b.ibox, cp))$
    else $cbox(set\_pos(b.cbox, cp))$
$b.pos ==$
    if $tag(b) = ibox$  then $b.ibox.pos$
    else $b.cbox.pos$
$set\_size(b, cp) ==$
    if $tag(b) = ibox$  then $ibox(set\_size(b.ibox, cp))$
    else $cbox(set\_size(b.cbox, cp))$
$b.size ==$
    if $tag(b) = ibox$  then $b.ibox.size$
    else $b.cbox.size$


% for each of these operators, just do coersions based on whether
% it's an instance or constraint

$create\_picture(pt) ==$
    if $(pt = inst\_pic)$  then $instance(create\_instance\_pic)$
    else $constraint(create\_constraint\_pic)$

$insert\_box(p, b) ==$
    if $tag(p) = instance$  then $instance(insert\_box(p.instance, b.ibox))$
    else $constraint(insert\_box(p.constraint, b.cbox))$

$insert\_arrow(p, a) ==$
    if $tag(p) = instance$  then $instance(insert\_arrow(p.instance, a.iarrow))$
    else $constraint(insert\_arrow(p.constraint, a.carrow))$

$copy\_picture(p) ==$
    if $tag(p) = instance$  then $instance(copy\_picture(p.instance))$
    else $constraint(copy\_picture(p.constraint))$

$pic\_union(p, p_1) ==$
    if $tag(p) = instance$  then $instance(pic\_union(p.instance, p_1.instance))$
    else $constraint(pic\_union(p.constraint, p_1.constraint))$

$is\_on\_a\_box(cp, p) ==$
    if $tag(p) = instance$  then $(is\_on\_a\_box(cp, p.instance))$
    else $(is\_on\_a\_box(cp, p.constraint))$

$box\_at(cp, p) ==$
```

```
    if $tag(p) = instance$ then $ibox(box\_at(cp, p.instance))$
    else $cbox(box\_at(cp, p.constraint))$

% Can't tell from os whether instance or constraint, so have to
% define boxes recursively.
$boxes(\{\} : OS) == \{\}$
$boxes(insert(os, obj)) ==$
    if $tag(obj) = iobj$ then
    if $tag(obj.iobj) = box$ then
    $insert(boxes(os), ibox(obj.iobj.box))$
    else $boxes(os)$
    else
    if $tag(obj.cobj) = box$ then
    $insert(boxes(os), cbox(obj.cobj.box))$
    else $boxes(os)$

$toggle\_in(os, obj) ==$
    if $tag(obj) = iobj$ then $iobjset(toggle\_in(os.iobjset, obj.iobj))$
    else $cobjset(toggle\_in(os.cobjset, obj.cobj))$

$valid\_attr(labelvar, obj) ==$
    if $tag(obj) = iobj$ then $valid\_attr(labelvar, obj.iobj)$
    else $valid\_attr(labelvar, obj.cobj)$

$change\_attr(obj, labelvar, value) ==$
    if $tag(obj) = iobj$ then $iobj(change\_attr(obj.iobj, labelvar, value))$
    else $cobj(change\_attr(obj.cobj, labelvar, value))$

$create\_box(pt, cp, cp_1, bl, thickness, starred, bt) ==$
    if $(pt = inst\_pic)$ then
    $ibox(create\_ibox(cp, cp_1, bl.ilabel, bt))$
    else
    $cbox(create\_cbox(cp, cp_1, bl.clabel, thickness, starred, bt))$

$create\_arrow(pt, b, b_1, parity, str, kind, thickness, starred) ==$
    if $(pt = inst\_pic)$ then
    $iarrow(create\_iarrow(b.ibox, b_1.ibox, parity, str))$
    else
    $carrow(create\_carrow(b.cbox, b_1.cbox, parity, str, kind, thickness, starred))$

$move\_all\_boxes(p, cp) ==$
    if $tag(p) = instance$ then $instance(move\_all\_boxes(p.instance, cp))$
    else $constraint(move\_all\_boxes(p.constraint, cp))$

$objects(p) ==$
    if $tag(p) = instance$ then $iobjset(objects(p.instance))$
    else $cobjset(objects(p.constraint))$
```

```
        delete_objs(p, os) ==
            if tag(p) = instance  then instance(delete_objs(p.instance, os.iobjset))
            else constraint(delete_objs(p.constraint, os.cobjset))

        extract_wf(pt, os) ==
            if pt = inst_pic  then instance(extract_wf(os.iobjset))
            else constraint(extract_wf(os.cobjset))

        well_formed(p) ==
            if tag(p) = instance  then well_formed(p.instance)
            else well_formed(p.constraint)

    implies
      converts box_to_O, create_picture : PicType → P,
        copy_picture : P → P, is_on_a_box : CoordPair, P → Bool, boxes : OS → BSet,
        objects : P → OS, move_all_boxes : P, CoordPair → P, well_formed : P → Bool

        % not converted because doesn't check all coercions: insert_box, insert_arrow,
        % pic_union, toggle_in, valid_attr, change_attr, create_box, create_arrow,
        % delete_objs, extract_wf

        % box_at is not converted because it assumes is_on_a_box


% ———————
% PIXEL MAP
% ———————
PixelMap : trait

   PixelValue enumeration of on, off

   includes BandASorts, FiniteMapping(PixelMap, CP, PixelValue)


% ———————
% EDITOR STATE
% ———————

EditorState : trait
   includes PicUnion(ObjType_Obj for O, Box_Obj for B, Arrow_Obj for A), PixelMap

   OT enumeration of box, arrow                                    % object type

   Ed tuple of
```

```
    pos : CoordPair, size : CoordPair,                    % position info
    picture : P,                                          % graphical objects
    picture_type : PicType,                               % picture type
    object_type : OT,                                     % object mode info
    arrow_kind : ArrowKind,
    arrow_parity : Parity,
    thickness : LineThickness,
    starred : Bool,
    selected_objs : OS                                    % selection

  introduces
    display_window : Ed → PixelMap                        % not defined here


object miro_editor
  initialized by CreateEditor

  using EditorState
  type Cp based on CoordPair from BandASorts
  type Str based on Str from String
  type Bt based on BoxType from BandASorts
  type Bl based on BL from PicUnion
  type Box based on B from PicUnion
  type Arrow based on A from PicUnion
  type ObjType based on O from PicUnion
  type ObjSet based on OS from PicUnion
  type Value based on Value from ChangeAttr
  type Label based on Label from ChangeAttr
  type Picture based on P from PicUnion

  type Editor based on Ed from EditorState
  private e : Editor

  invariant e.selected_objs ⊆ objects(e.picture)

  invariant well_formed(e.picture)

  operation CreateEditor (posn, size : Cp)
    requires true
    ensures  newobj (e_obj) ∧
             e' = [posn, size, create_picture(inst_pic),
                   inst_pic, box, syn, positive, thin, false,{}:OS]

  operation DrawBox (cp1, cp2 : Cp, label : Bl, bt : Bt)
    requires e.object_type = box
    modifies (e_obj)
    ensures
    ∃ b:Box
            **newobj** (b) ∧
            b!post = create_box(e.picture_type,cp1, cp2, label, e.thickness, e.starred, bt) ∧
            e′.picture = insert_box(e.picture, b)

**operation** DrawArrow (cp1, cp2 : Cp, label : Str)
   **requires** e.object_type = arrow ∧
            is_on_a_box(cp1,e.picture) ∧is_on_a_box(cp2,e.picture)
   **modifies** ($e_{obj}$)
   **ensures**
      ∃ a:Arrow
            **newobj** (a) ∧
            a!post = create_arrow(e.picture_type, box_at(cp1, e.picture), box_at(cp2, e.picture),
                  e.arrow_parity, label, e.arrow_kind, e.thickness, e.starred) ∧
            e′.picture = insert_arrow(e.picture, a)

**operation** Select (obj : ObjType)
   **requires** obj ∈ objects(e.picture)
   **modifies** ($e_{obj}$)
   **ensures** e′.selected_objs = toggle_in(e.selected_objs, obj)

**operation** GroupSelect (os : ObjSet)
   **requires** os ⊆ objects(e.picture)
   **modifies** ($e_{obj}$)
   **ensures** e′.selected_objs = (e.selected_objs ∪os)

**operation** Unselect ( )
   **requires** true
   **modifies** ($e_{obj}$)
   **ensures** e′.selected_objs = {}:OS

**operation** MoveBoxes (delta : Cp)
   **requires** true
   **modifies** ($e_{obj}$)
   **ensures**
      ∀ b:Box
            (b ∈ boxes(e.selected_objs) =>
                  b!post = set_pos(b!pre, (b!pre).pos+delta)) ∧
            e′.selected_objs = {}:OS

**operation** ResizeBox (b : Box, pos : Cp, size : Cp)
   **requires** e.selected_objs = {box_to_O(b)}
   **modifies** ($e_{obj}$)
   **ensures** b′ = set_size(set_pos(b,pos),size) ∧
            e′.selected_objs = {}:OS

**operation** DeleteObjs ( )
```

```
    requires  true
    modifies  ($e_{obj}$)
    ensures   e′.picture = delete_objs(e.picture, e.selected_objs) ∧
              e′.selected_objs = {}:OS

operation  CopyObjs (delta : Cp)
    requires  true
    modifies  ($e_{obj}$)
    ensures
        ∃ newpic : Picture ∀ o:ObjType
                 newobj (newpic) ∧
                 newpic!post = copy_picture(extract_wf(e.picture_type,e.selected_objs)) ∧
                 (o ∈ objects(newpic!post) => newobj (o)) ∧
                 e′.picture = pic_union(e.picture,
                        move_all_boxes(newpic!post, delta)) ∧
                 e′.selected_objs = {}:OS

operation  ChangeAttribute (o:ObjType, attr:Label, val:Value)
    requires  valid_attr(attr, o) ∧valid_value(val, attr)
    modifies  ($o_{obj}$)
    ensures   o′ = change_attr(o,attr,val)

operation  Clear ( )
    requires  true
    modifies  ($e_{obj}$)
    ensures   e′.picture = create_picture(e.picture_type) ∧
              e′.selected_objs = {}:OS
```

References

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

[Che89] Jolly Chen. The Larch/Generic Interface Language. S.B. Thesis, MIT, May 1989.

[GGH90] Stephen J. Garland, John V. Guttag, and James J. Horning. Debugging Larch Shared Language specifications. Technical report, DEC-SRC, 1990.

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

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

[HMT+90] Allan Heydon, Mark W. Maimone, J.D. Tygar, Jeannette M. Wing, and Amy Moormann Zaremski. Miró: Visual specification of security. *IEEE Transactions on Software Engineering*, 16(10):1185–1197, October 1990.

[Ler91] Richard Lerner. Modular specifications of concurrent programs. Ph.D. thesis, Carnegie Mellon (to be published), 1991.

[Mal82] William R. Mallgren. *Formal Specification of Interactive Graphics Programming Languages.* ACM Distinguished Dissertation Series. The MIT Press, Cambridge, Massachusetts, London, England, 1982.

[MTW90] Mark W. Maimone, J. D. Tygar, and Jeannette M. Wing. Formal semantics for visual specification of security. In S.K. Chang, editor, *Visual Languages and Visual Programming.* Plenum Publishing Corporation, 1990. A preliminary version of this paper appeared in *Proceedings of the 1988 IEEE Workshop on Visual Languages*, October, 1988, pages 45–51.

[Suf82] Bernard Sufrin. Formal specification of a display-oriented text editor. *Science of Computer Programming*, 1(2):157–202, May 1982.

[Win88] Jeannette M. Wing. Specifying Avalon objects in Larch. Technical Report CMU-CS-88-208, CMU, 1988.

[Zar90] Amy Moormann Zaremski. The Miró editor: A user's guide. Miró Note 7, Carnegie Mellon University, School of Computer Science, 1990.