# Unintrusive Ways to Integrate Formal Specifications in Practice

Jeannette M. Wing and Amy Moormann Zaremski

February 25, 1991
CMU-CS-91-113

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Submitted to *VDM '91*.

**Abstract**

Formal methods can be neatly woven in with less formal, but more widely-used, industrial-strength methods. We show how to integrate the Larch two-tiered specification method [GHW85a] with two used in the waterfall model of software development: Structured Analysis [Ros77] and Structure Charts [YC79]. We use Larch traits to define data elements in a data dictionary and the functionality of basic activities in Structured Analysis data-flow diagrams; Larch interfaces and traits to define the behavior of modules in Structure Charts. We also show how to integrate loosely formal specification in a prototyping model by discussing ways of refining Larch specifications as code evolves. To provide some realism to our ideas, we draw our examples from a non-trivial Larch specification of the graphical editor for the Miró visual languages [HMT+90]. The companion technical report, CMU-CS-91-111, contains the entire specification.

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

**Keywords:** formal methods, formal specification,software engineering, Structured Analysis, Structure Charts, Larch

## 1. A Different Picture of Software Development

Most pictures of the software development process are variations of the one shown in Figure 1(a) where we name the standard phases *requirements analysis*, *design*, *implementation*, and *validation*. Each phase results in at least one tangible product, e.g., a requirements specification, a design notebook, executable code, and test suites.

The software engineering community is beginning to acknowledge the need to use and the potential benefits obtained from using more formal methods for developing software. How then do formal methods fit in the picture? In 1980 Guttag and Horning proposed that formal specifications be used in program design [GH80]. This idea suggests revising Figure 1(a) to that in Figure 1(b) by inserting the phase *formal specification*. This revised picture implies that given a set of informally described requirements, a *system specifier* (a brand new player on the software development team [Win85]) writes a formal specification of the system to be built. After the system is built, formal *verification* is possible since in principle the implementation can be proven correct with respect to the formal specification. We call this overall development process the *waterfall* model.

Zooming in at the design phase of the waterfall model gives rise to another common picture shown in Figure 2. Here, some initial formal specification codifies the informal requirements, shown as a cloud in the picture. Then, by applying a series of well-defined correctness-preserving transformations, the system specifier refines a specification from one step to the next, perhaps generating proof obligations along the way. The transformation process ends when an implementation with acceptable performance is reached. Discharging proof obligations in parallel with applying stepwise refinement dispenses with the need of a separate verification phase. A proof of correctness is generated as the program is developed [Dij76, Gri81, Jon86]. We call this development process the *refinement* model.

Software engineers who interpret at face value these pictures of software development might get the wrong impression that formal specifications have a fixed, well-defined, and rigid role in the software development process. The problem with the waterfall model picture is that it implies that a formal specification should be written before implementation begins. It implies that without doing verification, the developer would never be certain that an implementation is correct. The problem with the refinement model picture is that it implies that the design process can be accomplished completely systematically through stepwise refinement. The only creativity needed is to choose what to refine at each step and which of many provably correct transformations to apply. It requires that such transformations exist. These are all strong requirements and implications that can result from forgetting that these pictures are just abstractions of what occurs in practice. They idealize the software development process by hiding false starts, feedback loops, and parallel development strategies.

We propose a different picture shown in Figure 3. Here, starting from a set of informally described requirements, four iterative activities occur in parallel: formal specification, implementation, validation and verification, and writing informal documentation[1]. Unrolling each of the parallel activities (e.g., see Figure 4) shows that iterations of each need not occur in lockstep and that each can affect subsequent iterations of another. Our picture accommodates the waterfall and refinement models. If we ignore the documentation branch, we get the waterfall model by unrolling each of the three leftmost branches once, going down the leftmost branch and moving to the right across the other branches as time progresses. We get the refinement model by unrolling the leftmost branch multiple times and then moving right at the last refinement step to get an implementation. Our picture also accommodates a prototyping model where we follow the implementation branch that has been unrolled multiple times and ignore the other branches; none of the previous pictures directly reflect this prototyping model.

Our picture not only accommodates different software development models but also reflects more realistically how software is developed in practice. It explicitly shows iterations through specifications and implementations, and how results of one phase can influence and provide feedback to others. It makes more explicit the role of informal documentation rather than relegating it to be an invisible by-product of each phase. It allows the first use of formal methods to occur at any time during the development process: before, during, or after the system is built. The point of our picture is to convey to software engineers that applying formal methods need not be an intimidating or burdensome task. Applying formal methods is not something done by a group of people sitting off in a corner by themselves, but

[1] In order not to clutter the picture, we do not draw the implicit arrows between non-adjacent boxes, e.g., between formal specification and informal documentation.

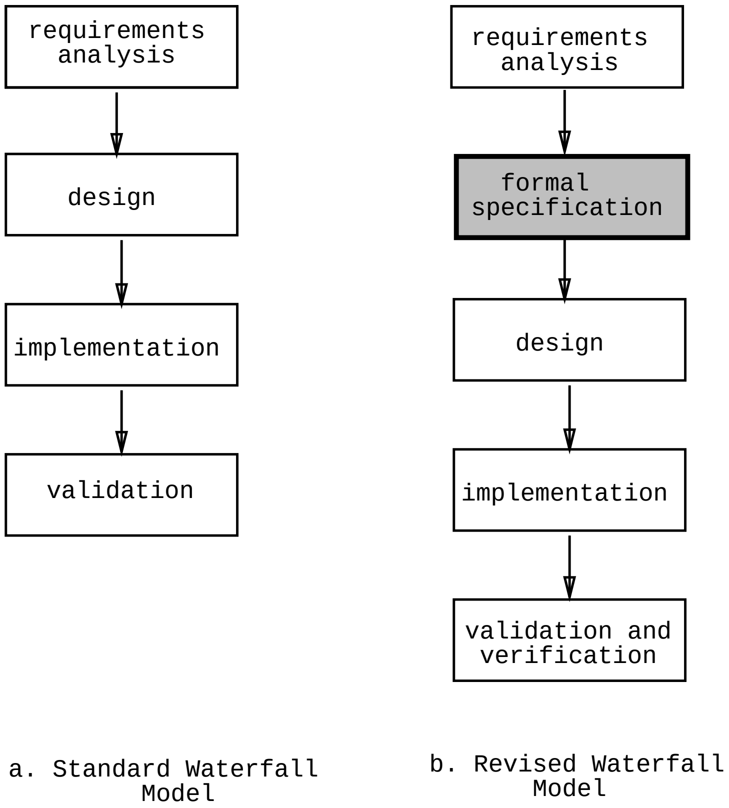


Figure 1: The Waterfall Model

rather, more importantly, could and should be done alongside conventional software development activities. More significantly, with care and foresight specific formal methods can be neatly woven in with less formal, but more widely-used, industrial-strength methods. Thus, we use our picture to help illustrate natural and convenient ways of unintrusively integrating formal methods in the software development process: the main subject and contribution of this paper. Henceforth we will focus on the interplay between specification and implementation, though many of our remarks hold for the interplay between the other phases, e.g., formal specification and validation and verification.

Specifically, we present two ways to integrate formal methods in the current practice of software engineering. First, we show in Section 3 how to integrate the Larch two-tiered specification method [GHW85a] with methods used in a traditional waterfall model. Larch specifications fit in with Structured Analysis [Ros77], a typical requirements analysis method, and Structure Charts [YC79], a typical design method. Next in Section 4, we explain how we can integrate Larch in a prototyping model where versions of specifications and implementations evolve in parallel. Although we make our points using specific techniques, our approach is generally applicable to other requirements analysis and design methods such as SADT [Ros85] and JSM [Jac83] and other formal methods such as VDM [Jon86] and Z [Spi88]. To provide some realism to our ideas, instead of illustrating them using small examples, we extract pieces from a non-trivial Larch specification, that of the Miró graphical editor [Zar91]. We begin with an overview of the specificand ( Miró ) and specification language (Larch) in the next section. We conclude in Section 5 with some general words of advice to interested practitioners.

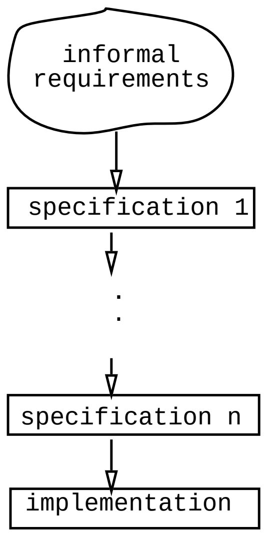


Figure 2: The Refinement Model

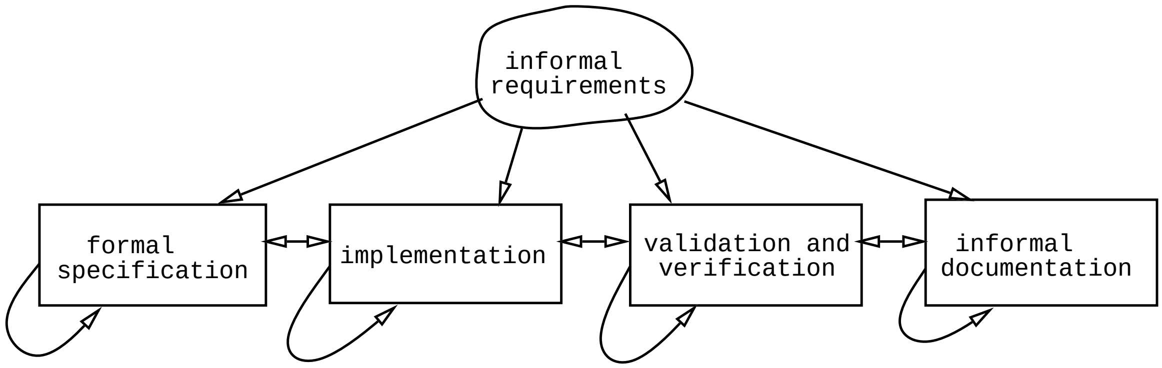


Figure 3: A Parallel-Iterative Model of Software Development

## 2. Miró and Larch

### 2.1. Specificand: Miró

The goal of the Miró Project is to provide language and tool support for letting users specify formally through pictures the security configuration of file systems (i.e., which users have access to which files) and general security policy constraints (i.e., rules to which a configuration must conform).

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

The basic elements in the instance language are *boxes* and *arrows*. Boxes that contain no other boxes represent users and files. Boxes can contain other boxes to indicate groups of users and directories of files. User group boxes

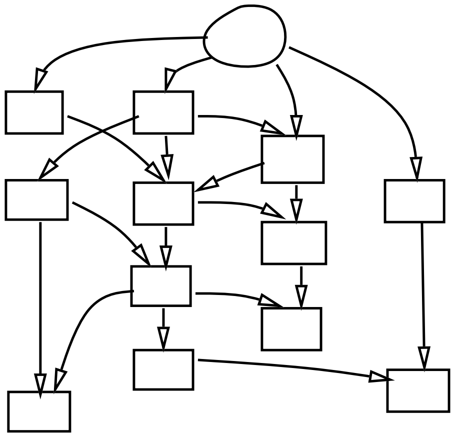

Figure 4: An Unrolling of the Parallel-Iterative Model

may overlap to indicate a user is in more than one group. Labeled arrows go from user (group) boxes to file (group) boxes; the label indicates the access mode, e.g., read or write. Access rights are inherited by corresponding pairs of boxes contained within boxes connected by arrows, thus minimizing the number of arrows necessary to draw. Arrows may be negated to indicate the denial of the labeled access.

Figure 5 shows an instance picture, as drawn in the Miró editor. The positive arrow from `Alice` to `Alice's files` indicates that Alice has read and write access to her files. The positive arrow from `Alice's friends` to Alice's `schedule` file indicates that both Bob and Charlie have read access to Alice's schedule. By default, since there are no arrows between Alice's friends and her other files, Bob and Charlie do not have read access to Alice's mail file. We could also show this property with an explicit negative arrow between Alice's friends and her mail file. The presence of both positive and negative arrows can lead to possibly *ambiguous* pictures, a property formally defined in [MTW90].

The *constraint* language also consists of boxes and arrows, but here the objects have different meanings. A box labeled with an expression defines a set of instance boxes. E.g., the left-hand box in Figure 6 denotes the set of instance boxes of type User. There are three sorts of arrows, allowing us to describe three different relations between boxes in an instance picture, *IP*: syntactic (solid horizontal) – whether an arrow explicitly appears in *IP*; semantic (dashed horizontal) – whether an access right exists in the matrix denoted by *IP*; and containment (solid vertical with head inside box) – whether a box is nested within another in *IP*. Additionally, the *thickness* attribute of each constraint object is key in defining a constraint picture's meaning: in general, for each set of instance objects that matches the thick part of the constraint, there must be another set of objects (disjoint from the set matching the thick part) that matches the thin part. Figure 6 shows a constraint picture that specifies that users who have write access to a file must have read access to it as well.

The Miró editor provides facilities to create, view, and modify both instance and constraint pictures. The left-hand side of the window in Figure 5 displays a menu from which users can select the type of picture and object they wish to draw. The editor also serves as an interface to other Miró tools: an *ambiguity checker* that determines whether an instance picture is ambiguous, a *constraint checker* that determines whether an instance picture matches a constraint picture, a *prober* that creates a representation of an instance picture for a given file system, and a *verifier* that determines whether the result of the prober matches an instance picture. We revisit the relationship among these tools in Section 3.

### 2.2. Specification Language: Larch

We present a brief refresher of Larch here and give further details as necessary. See [GHW85b, GHM90] for more details.

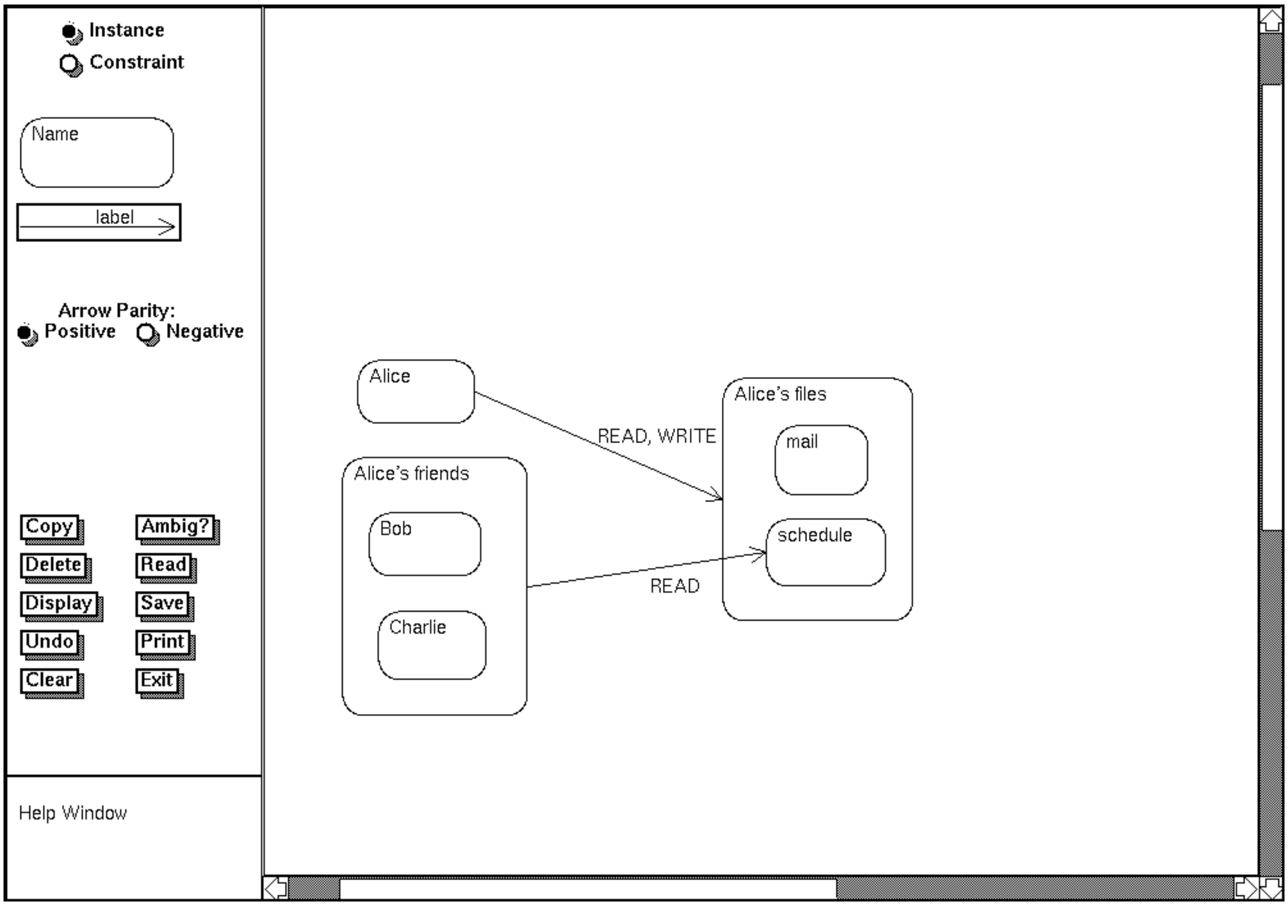


Figure 5: The Miró editor and a sample instance picture

Larch provides a "two-tiered" approach to specification. In one tier, the specifier writes *traits* in the Larch Shared Language (LSL) to assert state-independent properties of a program. Each trait introduces *sorts* and *operators* and defines equality between terms composed of the operators (and variables of the appropriate sorts). E.g., the *Box* trait in Figure 7 introduces the sort *B*, which is a **tuple of** six fields, and the operator *copy_box*. By our use of LSL's **tuple of** constructor, we implicitly introduce four other sorts (*CoordPair*, *Label*, *LineThickness*, and *BoxType*[2]) and the *__field_name* and *set_field_name* operators for each of the six *field_name*s in the tuple. Five equations constrain the meaning of *copy_box* and together imply that a box *b* is not necessarily equal to *copy_box(b)* since they may differ in the values set in their corresponding *b_label* fields.

In the second tier, the specifier writes *interfaces* in a Larch interface language (here, we use Lerner's extensions [Ler91] to the Generic Interface Language (GIL) [Che89]), to describe state-dependent effects of a program. An *operation interface* includes the operation's header and a body with three clauses: a **requires** clause states the operation's pre-condition; a **modifies** clause lists those objects whose value the operation may possibly change; an **ensures** clause states the operation's post-condition. The assertion language for the pre- and post-conditions is drawn from LSL traits.

An *object interface* contains a **based on** clause and a set of operation interfaces, one for each operation exported by the object. Through **based on** clauses, Larch interfaces link 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. An object interface also may contain **invariant** clauses that state properties of the object preserved by its operations.

[2] *Bool* is built-in to all LSL traits.

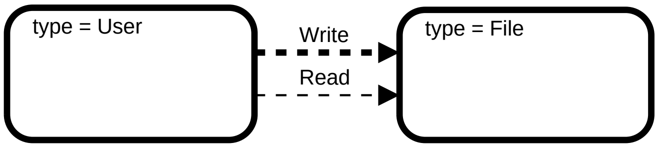


Figure 6: A sample constraint picture

*Box*(*B*) : **trait**

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

**introduces**
$copy\_box : B \rightarrow B$

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

Figure 7: The Box Trait

Part of the interface specification for the Miró editor shown in Figure 8 defines the type *Editor*, which is based on the sort *Ed* introduced in the EditorState trait. The **private** clause declares an object *e* of *Editor* type that is local to this interface. It is treated as an implicit argument and result of each operation in the object's interface. ResizeBox's pre-condition requires that the set of selected objects be exactly the singleton set containing ResizeBox's argument *b* of Box type (*box_to_O* is an operator that coerces a box sort to a more generic object sort). Its post-condition updates the size and position of the box being resized (using *set_size* and *set_pos*) and unselects all objects. In a post-condition an undecorated formal, *e*, stands for the initial value of the object; a primed one, *e'*, stands for the final value. The **modifies** clause states that MoveBoxes may change only the editor and no other object.

We used the LSL Checker and the GIL Checker to check all specifications in this paper for syntactic and sort/type correctness. The Appendix contains the full specifications for the excerpts we give in this paper; see [Zar91] for the full specification of the Miró editor.

## 3. Formal Methods Integrated in the Waterfall Model

We assume some familiarity with the two software development techniques discussed in this section, Structured Analysis and Structure Charts, and show how we can integrate Larch with both.

**object** miro_editor
. . .
    **type** Editor **based on** Ed **from** EditorState
    **private** e : Editor

    **invariant** . . .

    **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) $\wedge$ e'.selected_objs = $\{\}$

. . .

Figure 8: Part of the Miro Editor Interface Specification

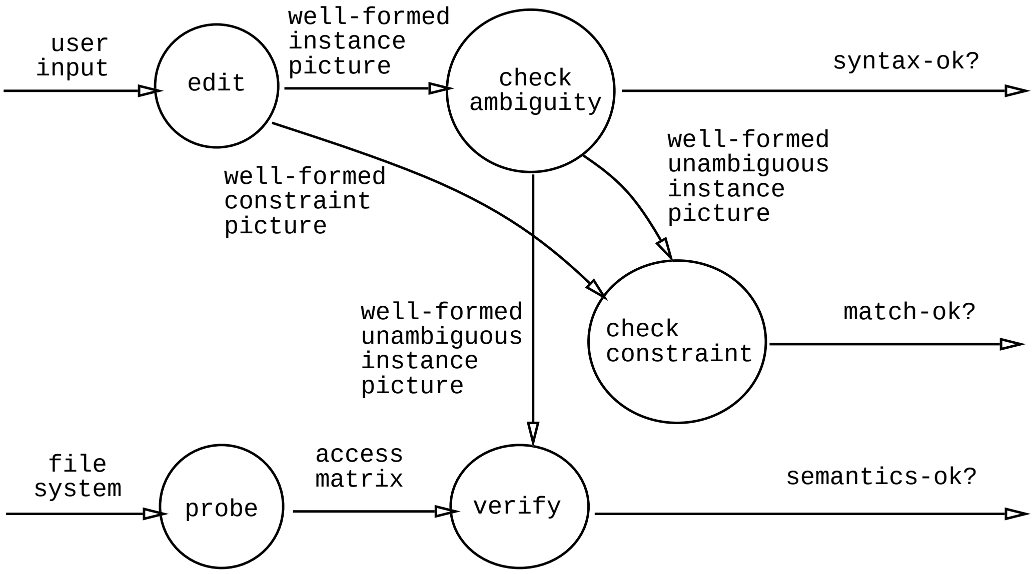


Figure 9: Data-Flow Diagram of the Miró Tools

### 3.1. Requirements Phase: Larch and Structured Analysis

Structured Analysis, developed by DeMarco [DeM78], is a common technique used to record a system's functional requirements. There are three components to a structured analysis document: data-flow diagrams, data dictionaries, and activity specifications. In a data-flow diagram, circles represent activities (processing elements) and arrows drawn between circles represent data elements flowing between their corresponding activities. A system specifier refines each activity in a data-flow diagram to a lower-level data-flow diagram; refinement continues until a set of basic activities is reached. A data dictionary provides the definition of all data elements flowing in a data-flow diagram (at each level of refinement). Activity specifications provide the definition of each basic activity.

For example, Figure 9 shows a top-level data-flow diagram of some of the functionality supported by the Miró tools. The *editor* takes user commands and creates well-formed instance pictures and well-formed constraint pictures. The *ambiguity checker* outputs well-formed unambiguous pictures that the *constraint checker* can check against given constraint pictures. The *prober* extracts an access matrix from a given file system directory that the *verifier* can then check against a given well-formed unambiguous instance picture.

Whereas notation and its interpretation for data-flow diagrams are standard, they are not for data dictionaries nor activity specifications. Here is one explicit point where Larch can be integrated in Structured Analysis:

| | |
|---|---|
| Use the Larch Shared Language to define | (1) data elements in a data dictionary and<br>(2) the functionality of basic activities. |

### 3.1.1. Data Dictionaries Written in LSL

A data dictionary defines the data elements in a data-flow diagram. Many software engineering textbooks [Myn90, Pfl91, ED89] recommend using BNF-like notation, such as + for concatenation and {} for repetition, to describe how to form new data elements out of basic ones. For example,

Miro picture = Instance picture | Constraint picture
Instance picture = { Box | Arrow }

indicates that a Miro picture can be either an instance picture or constraint picture and an instance picture can be a collection of boxes and arrows. BNF-like notation forces the specifier to model data completely in terms of a few basic sorts of data elements like numbers, characters, and strings. This approach is at best inconvenient, e.g., how would we say that a box is the "concatenation" of four lines, but at worst too low-level in that the model forces the specifier to make design decisions prematurely. For example, we do not care that a box is composed of four lines since the implementor could choose to represent it as a set of pixels; rather we want to specify only those attributes of boxes that are necessary to describe the behaviors of the activities that manipulate boxes. In short, we cannot use BNF-like notation to describe the abstract properties of the data elements, e.g., those that would distinguish constraint boxes from instance boxes.

Instead, we can use the Larch Shared Language to write the data dictionary by specifying the sorts of data elements labeling the arrows in a data-flow diagram such as the one in Figure 9. Let us build up to a trait that describes well-formed instance pictures by beginning with one that describes basic pictures.

Figure 10 shows the sorts and operators introduced in the BasicPicture trait. A picture (sort *Pic*) is a collection of boxes and arrows. The renamings of sort identifiers in the two uses of the *Set* trait (from the Larch Handbook [GHW85b]) gives us the sorts *BSet* and *ASet* for sets of boxes and arrows, and, among others, the operators $\{\}\!:\to BSet$ and $\{\}\!:\to ASet$ to denote respectively the empty set of boxes and the empty set of arrows. The **generated by** clause says that all terms denoting values of pictures are expressed in terms of the operators *create_picture*, *insert_box*, and *insert_arrow*. The **partitioned by** clause says that if for two pictures their corresponding sets of *boxes* and sets of *arrows* are the same, then the pictures are the same. Through an **implies** clause, LSL provides a place to record explicitly consequences of the first-order theory denoted by a trait. The **converts** ... **exempting** clause states a property about the trait related to *sufficient-completeness* [Gut75]; it says that a term using any operator listed in the **converts** clause can be shown equal to either a term not using any convertible operator or an **exempt** term.

The equational axioms defining the operators are straightforward and given in the standard style of "algebraic" specifications (define the meaning of each non-constructor operator in terms of each constructor operator). For example, as shown in the equations below, we define *move_all_boxes(pic,delta)* to move each box in the picture *pic* by some amount *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). The second equation states that the result of moving all boxes in a picture *pic* to which we have just inserted a box *b* is the same as (recursively) moving all boxes in *pic* and adding to it the result of changing *b*'s position by *delta*. The third equation similarly recurses through a picture's boxes without changing any inserted arrows.

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

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

**includes** *Box*, *Set*(*B*, *BSet*), *Arrow*, *Set*(*A*, *ASet*)

**introduces**
$\mathit{create\_picture} : \rightarrow \mathit{Pic}$
$\mathit{insert\_box} : \mathit{Pic}, B \rightarrow \mathit{Pic}$
$\mathit{insert\_arrow} : \mathit{Pic}, A \rightarrow \mathit{Pic}$
$\mathit{move\_all\_boxes} : \mathit{Pic}, \mathit{CoordPair} \rightarrow \mathit{Pic}$
$\mathit{copy\_picture} : \mathit{Pic} \rightarrow \mathit{Pic}$
$\mathit{delete\_box} : \mathit{Pic}, B \rightarrow \mathit{Pic}$
$\mathit{delete\_arrow} : \mathit{Pic}, A \rightarrow \mathit{Pic}$

$\mathit{boxes} : \mathit{Pic} \rightarrow \mathit{BSet}$
$\mathit{arrows} : \mathit{Pic} \rightarrow \mathit{ASet}$
$\mathit{arrows\_attached\_to\_box} : \mathit{Pic}, B \rightarrow \mathit{ASet}$
$\mathit{arrows\_attached\_to\_boxes} : \mathit{Pic}, \mathit{BSet} \rightarrow \mathit{ASet}$

**asserts**
*Pic* **generated by** $\mathit{create\_picture}, \mathit{insert\_box}, \mathit{insert\_arrow}$
*Pic* **partitioned by** $\mathit{boxes}, \mathit{arrows}$
$\ldots <$ Equations go here. $> \ldots$
**implies**
**converts** $\mathit{move\_all\_boxes}, \mathit{copy\_picture}, \mathit{delete\_box}, \mathit{delete\_arrow},$
$\mathit{boxes}, \mathit{arrows}, \mathit{arrows\_attached\_to\_box}, \mathit{arrows\_attached\_to\_boxes}$
**exempting** $\forall\ b : B, a : A$
$\mathit{delete\_box}(\mathit{create\_picture}, b), \mathit{delete\_arrow}(\mathit{create\_picture}, a)$

Figure 10: Part of the BasicPicture Trait

$\mathit{insert\_arrow}(\mathit{move\_all\_boxes}(\mathit{pic}, \mathit{delta}), a)$

Next we define a trait that builds upon basic pictures to define well-formedness for pictures. The *WFPicture* trait in Figure 11 introduces the *well-formed* operator among others. A well-formedness condition that is common to both instance and constraint pictures is that all arrows must be attached to boxes (no dangling arrows). We define the *arrows_attached* operator to capture this condition. *Arrows_attached* checks that a picture's set of arrows is attached by using *arrow_attached* to check each arrow in the set. An arrow is attached in a picture if the boxes at each of its ends are both in the picture.

$\mathit{arrows\_attached}(\mathit{pic}, \{\}) == \mathit{true}$
$\mathit{arrows\_attached}(\mathit{pic}, \mathit{insert}(\mathit{as}, a)) == \mathit{arrow\_attached}(\mathit{pic}, a) \wedge \mathit{arrows\_attached}(\mathit{pic}, \mathit{as})$

$\mathit{arrow\_attached}(\mathit{pic}, a) == (((a.\mathit{to\_box}) \in \mathit{boxes}(\mathit{pic})) \wedge ((a.\mathit{from\_box}) \in \mathit{boxes}(\mathit{pic})))$

We then succinctly define the well-formedness condition common to instance and constraint pictures as:

$\mathit{well\_formed}(\mathit{pic}) == \mathit{arrows\_attached}(\mathit{pic}, \mathit{arrows}(\mathit{pic}))$

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. We define elsewhere the sort ($O$) for objects to be a **union of** the sorts ($B$ and $A$) for boxes and arrows. For each object in the set of objects to be deleted, *delete_objs* checks whether the object is a box, i.e., *tag(obj) = box*, or an arrow and then uses the appropriate operator, *delete_wf_box* or *delete_wf_arrow*. Deleting just a box may violate the well-formedness condition since it could result in dangling arrows. Hence, if the box, $b$, being deleted is in the picture, *delete_wf_box* must delete all attached arrows before deleting $b$. *Delete_wf_arrow* operator deletes the arrow $a$ from the picture only if $a$ is in the picture.

$\mathit{delete\_objs}(\mathit{pic}, \{\}) == \mathit{pic}$
$\mathit{delete\_objs}(\mathit{pic}, \mathit{insert}(\mathit{os}, \mathit{obj})) ==$ **if** $\mathit{tag}(\mathit{obj}) = \mathit{box}$

$WFPicture(Pic)$ : **trait**
**includes** $BasicPicture(Pic), Obj$

**introduces**
$arrows\_attached : Pic, ASet \to Bool$
$arrow\_attached : Pic, A \to Bool$
$well\_formed : Pic \to Bool$

$delete\_objs : Pic, ObjSet \to Pic$
$delete\_wf\_box : Pic, B \to Pic$
$delete\_wf\_arrow : Pic, A \to Pic$
$delete\_arrows : Pic, ASet \to Pic$

$extract\_wf : ObjSet \to Pic$
**asserts** $\forall$ $pic : Pic$
$well\_formed(pic) == arrows\_attached(pic, arrows(pic))$
. . . < Other equations go here. > . . .

Figure 11: Part of the WFPicture Trait

**then** $delete\_objs(delete\_wf\_box(pic, obj.box), os)$
**else** $delete\_objs(delete\_wf\_arrow(pic, obj.arrow), os)$

$delete\_wf\_box(pic, b) ==$ **if** $b \in boxes(pic)$
**then** $delete\_box(delete\_arrows(pic, arrows\_attached\_to\_box(pic, b)), b)$
**else** $pic$
$delete\_wf\_arrow(pic, a) ==$ **if** $a \in arrows(pic)$ **then** $delete\_arrow(pic, a)$ **else** $pic$

*Extract_wf* returns a picture that is the maximal well-formed subset of a set of objects. The interface specification for the editor uses *extract_wf* to describe the behavior of the editor operation that copies objects (CopyObjs). The set of objects to be copied is a sub-picture, which may or may not be well-formed. The result of *extract_wf(os)* is a well-formed picture that contains all the objects of *os* except the dangling arrows, i.e., arrows that are not attached to boxes in *os*. We define *extract_wf* indirectly by defining what *boxes* and *arrows* are in the resulting picture:

$WFInstancePic$ : **trait**
**includes** $WFPicture(IPic, create\_instance\_pic$ **for** $create\_picture)$
**introduces**
$users\_to\_files : IPic, IASet \rightarrow Bool$
$user\_to\_file : IPic, IA \rightarrow Bool$

**asserts** $\forall$ $ipic : IPic$
$users\_to\_files(ipic, \{\}) == true$
$users\_to\_files(ipic, insert(as, a)) ==$
$user\_to\_file(ipic, a) \wedge users\_to\_files(ipic, as)$
$user\_to\_file(ipic, a) ==$
$(a.from\_box).box\_type = User \wedge (a.to\_box).box\_type = File$
$well\_formed(ipic) == users\_to\_files(ipic, arrows(ipic))$

Figure 12: Part of the WFInstancePic Trait

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

Instance pictures have an additional well-formedness condition that constraint pictures do not: each arrow must start from a box of type User (see the sixth field labeled *box_type:BoxType* of Figure 7) and end at a box of type File. Figure 12 shows the WFInstancePic trait that includes the WFPicture trait and adds this additional well-formedness condition. This example shows how one trait (WFInstancePic) can place further constraints on an operator, e.g., *well_formed*, introduced and defined elsewhere (WFPicture).

Returning to Figure 9 we now know more formally what "well-formed instance picture" means: in all pictures each arrow is attached to boxes at both ends and each goes from a User to a File box. We define other traits for defining distinguishing characteristics of constraint pictures and unambiguous (instance) pictures similarly and omit the details here. In summary, we see that LSL traits give a more precise definition of the sorts of data elements labeling the arrows in data-flow diagrams.

### 3.1.2. Activity Specifications Written in LSL

Activity specifications are usually written in English, pseudo-code, or using decision tables. Since we can interpret each basic activity as a mathematical function that takes data elements as input and returns them as output, we can write for an activity specification of function *f* an LSL trait that introduces and defines through equations the function *f*. For example, after refining the *edit* activity of the data-flow diagram of Figure 9, we may find we need to define a basic activity called *adjust_arrows* whose effect is to ensure that the well-formedness property of instance pictures is maintained.

We define *adjust_arrows* on pictures with the following equations, where *set_tail_pos* (*set_tail_head*) sets the position information for an arrow's tail (head) and *find_tail_pos* (*find_tail_head*), given a picture and two boxes, determines the appropriate coordinate pair of the tail (head) of the arrow being adjusted.
$adjust\_arrows(create\_picture, b) == create\_picture$
$adjust\_arrows(insert\_box(pic, b1), b) == insert\_box(adjust\_arrows(pic, b), b1)$
$adjust\_arrows(insert\_arrow(pic, a), b) ==$
**if** $a.from\_box = b$ **then**
$insert\_arrow(adjust\_arrows(pic, b), set\_tail\_pos(a, find\_tail\_pos(pic, b, a.to\_box)))$
**else if** $a.to\_box = b$ **then**
$insert\_arrow(adjust\_arrows(pic, b), set\_head\_pos(a, find\_head\_pos(pic, b, a.to\_box)))$
**else** $insert\_arrow(adjust\_arrows(pic, b), a)$

In general, for any basic activity we can write its functional definition since LSL provides a conditional construct,

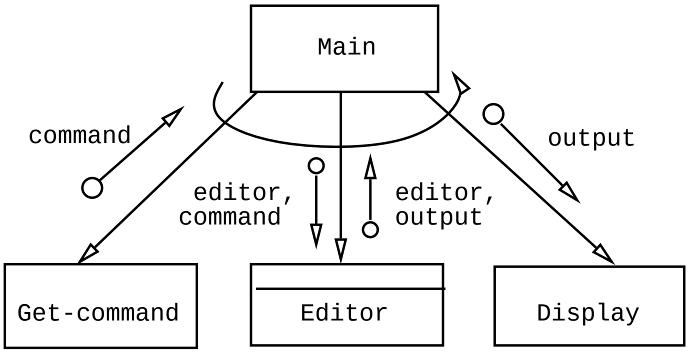


Figure 13: A Structure Chart Using the Editor Module

function composition, and recursion.

### 3.2. Design Phase: Larch and Structure Charts

Yourdon and Constantine developed a structured design method that lets software developers express system designs using *Structure Charts* [YC79]. Developers may even choose to transform data-flow diagrams created in the requirements analysis phase to produce Structure Charts in the design phase. A Structure Chart is a hierarchical diagram in which rectangular boxes represent modules in a system and arrows connecting boxes represent the calling relationships among the modules.

For example, the Structure Chart in Figure 13 shows the top-level control structure between four modules we have identified for the Miró editor[3]. The Main module loops receiving input commands from the user through the Get-command module, calling an Editor module that accepts different user commands initializing or transforming the editor state accordingly, and calling a Display module to output the transformed editor state. The short, labeled arrows represent the direction in which data flows between modules. We show the Editor module as a *data abstraction module* by drawing a double horizontal bar; more typically, we could depict it as a *procedural abstraction module* (single horizontal bar) that serves as a dispatcher (denoted by a diamond superimposed on the lower horizontal bar) to each of the operations exported by the data abstraction.

Structure Charts and Larch interface specifications are ideal complements of each other. Larch interface specifications are appropriate for specifying the behavior of each module. Larch provides no means of showing the interconnection among modules; structure charts do. Here is the explicit point where Larch can be integrated in the software development technique based on Structure Charts:

| Use Larch interface language specifications to describe the behavior of modules in a Structure Chart. |
|---|

To flesh out the editor example in more detail, we present pieces of the Larch interface specification of the editor data abstraction specified as a GIL object interface. We begin by describing editor state at the trait level and then describe the editor's operations at the interface level.

#### 3.2.1. LSL Specification of Editor State

The *EditorState* trait (Figure 14) introduces the tuple sort *Ed* to stand for 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

[3]Here, we focus on just the "edit" activity in the data-flow diagram of Figure 9.

*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* $\rightarrow$ *PixelMap* % not defined here

Figure 14: The EditorState Trait

*P* (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 (which is relevant only for constraint pictures); the rest of the attributes are self-explanatory. We introduce only one operator, *display_window*, but leave it unspecified here; we could use it to define a mapping from the abstract editor state to an actual mapping of screen pixels.

### 3.2.2. Part of the Larch Interface for the Editor

In Figure 8 we presented part of the Miró editor interface specification. We explain it in more detail here. We first establish using the **based on** clause a correspondence between the type of object being specified and the sort of values the object ranges over. We give elsewhere object specifications for other types, e.g., object sets (ObjSet) and coordinate pairs (Cp), on which the editor operation's depends. An **invariant** specifies properties that must be true after every procedure and before all procedures except those named in the **initialized by** clause. In this case, the first invariant states that the editor maintains the well-formedness of a picture and the second states that the set of selected objects must be a subset of all objects in the editor's picture.

**object** miro_editor
**type** Editor **based on** Ed **from** EditorState
**private** e : Editor

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

**initialized by** CreateEditor

. . . < Interface specifications of the twelve exported operations go here. > . . .

In the Appendix we give specifications of the twelve operations in the editor interface. The operations include creating an editor (CreateEditor), drawing a box (DrawBox), drawing an arrow (DrawArrow), selecting an object (Select), selecting a group of objects (GroupSelect), unselecting objects (Unselect), moving boxes (MoveBoxes), resizing boxes (ResizeBox), deleting objects (DeleteObjs), copying objects (CopyObjs), changing an object's attribute (ChangeAttribute), and clearing the editor (Clear). We present only three here (another was given in Section 2).

CreateEditor's effects are to return a new object (**newobj**) of type editor and to initialize various fields of the tuple representing the editor's state. ( [ . . . ] is the tuple constructor operator.)

**operation** CreateEditor (posn, size : Cp)
  **requires** true
  **ensures** (**newobj**($e_{obj}$)) $\wedge$
    (e' = ([posn, size, create_picture(inst_pic),
      inst_pic, box, syn, positive, thin, false,{}]))

Miró editor users can select or unselect objects in three ways: individual select/unselect, group select, and global unselect. For example, GroupSelect requires that its argument, *os*, be a subset of objects in the picture and has the effect of adding *os* to the currently selected set. If an object in *os* is already selected, it remains selected.

**operation** GroupSelect (os : ObjSet)
  **requires** (os $\subseteq$ objects(e.picture))
  **modifies** ($e_{obj}$)
  **ensures** (e'.selected_objs = (e.selected_objs $\cup$ os))

The copy operation in the editor is somewhat complex because of the well-formedness constraint. Copy operates on only 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*. *e'.picture* is then the result of combining the existing picture with *newpic*, which has been moved by *delta*[4]. CopyObjs has the side effects of creating a new object for each of the copied objects in the well-formed subset and of unselecting all objects.

**operation** CopyObjs (delta : Cp)
  **requires** true
  **modifies** ($e_{obj}$)
  **ensures**
    $\exists$ newpic : Picture $\forall$ o:ObjType
      **newobj** (newpic) $\wedge$
      newpic!post = copy_picture(extract_wf(e.picture_type,e.selected_objs)) $\wedge$
      (o $\in$ objects(newpic!post) => **newobj** (o)) $\wedge$
      e'.picture = pic_union(e.picture,
        move_all_boxes(newpic!post, delta)) $\wedge$
      e'.selected_objs = {}

### 3.3. Larch Tools and CASE Tools

In teaching the undergraduate software engineering course during the Spring 1991 semester, we loosely integrated the Larch Shared Language Checker and a Larch/C Checker with the CASE tool, *Software Through Pictures* (STP) [5]. STP supports graphical editors for many software analysis and design methods including Structured Analysis and Structure Charts. Where "annotations" are normally attached to data elements, e.g., in a data-flow diagram, users can in addition attach LSL specifications. Similarly, users can attach Larch/C specifications as annotations of modules in Structure Charts. A similar or even tighter integration is possible with many other CASE tools.

[4] GIL notation requires us to represent the final value of an object *a* as *a'* if *a* is a parameter or global variable and as *a*!post if *a* is a quantified variable.

[5] Copyright 1990. IDE Software Through Pictures is a registered trademark of Interactive Development Environment.

## 4. Formal Methods Integrated in a Prototyping Model

### 4.1. A Loose (Unintrusive) Integration

The prototyping model of software development has not reached the same maturity as the waterfall model. Software developers typically design and build prototypes in the implementation language itself rather than choosing a separate language. Alternatively, they can write prototypes in an executable formal specification language, e.g., PAISley [Zav72] and OBJ [FGJM85], and then rewrite the "final" prototype in a more efficient implementation language. This approach suggests a tight coupling between the specification and implementation, at least between the last specification and the first "efficient" implementation; it is roughly similar to the refinement model where the prototypes are specifications.

Although the use of an executable formal specification language is a valid way to integrate some classes of formal methods in a prototyping model, we explore here a looser integration in which a specification and an implementation are each written in their own notation and refined in different ways. This approach treats both the specification and implementation as dynamically changing by-products of each of the specifying and coding phases. Developing a specification and an implementation in parallel means that they can directly affect each other's next version. Each newer version of the specification (implementation) is influenced by earlier versions of the implementation (specification) and can influence the later versions.

When we developed both the Larch specification and the implementation of the Miró editor we followed a prototyping approach rather than the conventional waterfall or refinement approaches. Since our particular specificand has a heavy user interface component, prototyping was an appropriate and useful means of exploring and validating the graphical and device handling aspects of the Miró editor.

The unrolling shown in Figure 4 matches the actual development of the Miró editor. We began writing formal specifications in parallel with designing and implementing the editor. We wrote three major versions of the specification where the last version was written after the implementation had already passed into the maintenance phase. We specified in the first version a subset of the editor's functionality and handled only instance pictures. The second version was more comprehensive and also dealt with constraint pictures, but we invented new syntactic constructs to make the Larch traits more concise. Finally, we added yet more functionality in the current version, but reverted to standard Larch syntax by translating our syntactic inventions. The current version itself went through at least eight minor iterations. The entire development effort, which also included changes to the instance and constraint languages, lasted two years.

### 4.2. Larch and Prototyping

Since there are no well-known or widely-used counterparts to Structured Analysis and Structure Charts for a prototyping model of software development, we cannot as concretely describe as we did for the waterfall model how Larch or any similar formal method fits in with prototyping. A specification, whether written in Larch or not, can evolve in many ways: by making it more complete, abstracting, generalizing, and improving its presentation. In this section we focus on these four kinds of changes in the context of Larch and point out places where cross-fertilization between specification and implementation can occur.

#### 4.2.1. Make it more complete.

*Write signatures of operations; introduce new types.*

One way to start writing a Larch interface specification is to define the signatures of each operation. The signatures determine what other types of objects, hence data abstractions, need to be specified and implemented. For example, the GroupSelect operation takes a set of objects as its argument, introducing the need to implement an ObjSet (set of objects) type. This type information in the specification imposes constraints on other parts of specification too, in particular what is required by the **based on** trait: a type is based on some sort that must be introduced by some trait.

*Fill in bodies of interfaces.*

Once we have an operation's signature we can flesh out its body by writing **requires/modifies/ensures** clauses. Each clause has a direct influence on the implementation. The inclusion of a non-trivial **requires** clause (something other than "true") says the implementor need not check the system's state for whether the pre-condition holds since the client code is responsible for checking. The omission of a **modifies** clause (equivalent to saying **modifies nothing**) is a strong assertion that the implementor must uphold. Not only may no input argument be modified, but no global variable may be modified. All operations in the editor specification except CreateEditor may modify the editor's state. An **ensures** clause summarizes three important effects of an operation: what updates are made to any object listed in the **modifies** clause, what new objects are created, and what exceptional termination conditions are possible (error handling). We see from their specifications (in the Appendix) that CreateEditor, DrawBox, DrawArrow, and CopyObjs are the only editor operations that add new objects to the system's state. In contrast, ResizeBox does not delete an old box and create a new one; it just changes the properties of the old one.

*Fill in trait information.*

Just as interfaces can be fleshed out from one specification step to the next; traits can be fleshed out in a similar way. Signatures for trait operators can be given first; equations defining their meaning later. **Generated by**, **partitioned by**, and **converts** clauses can be added later, as necessary. Filling in details at the trait level may also affect the implementation. For example, **exempt** terms of a sort S usually stand for "error" values, which should be handled by a pre-condition in any operation referring to an object whose type is based on S. Alternatively, they could be handled by raising some exception (see discussion in Section 4.2.3 on removing pre-conditions).

### 4.2.2. Abstract.

An extreme example of applying abstraction is to consider the implementation as a specification. This viewpoint often makes sense when following the approach of developing a specification in parallel with an implementation. We use the implementation as an appropriate starting point for a specification and then abstract from it until we reach a high-level enough description of the system; this abstraction process goes in the opposite direction of that followed in the refinement model. The danger with this approach is that without sufficient abstraction the specification may reflect too faithfully the original implementation model and perhaps exclude other acceptable implementations.

For example, our implementation is written in a frame-based knowledge representation language and so we naturally represent each type of object in the editor as a *frame* with various *slots*. This record-like representation greatly influenced our initial specification in which we used LSL **tuple**s for boxes and arrows and drew a one-to-one correspondence between slots in a frame and fields in the corresponding tuple. We include, for example, a field called *selected* used to determine whether that box or arrow is currently selected. Every operation in the editor's interface except CreateEditor tested and toggled this field explicitly. Later we removed the *selected* field from the tuple upon realizing that we already maintained the relevant information with the set of selected objects in the editor state (*selected_objs*).

One way to abstract from a concrete model is to state a property of the model and ask whether it is an essential behavioral characteristic of the desired system. If the property is irrelevant then it can be left out, making the specification more abstract. Removing irrelevant detail from a specification may sometimes result in a complete redesign of the abstractions themselves. Again, influenced by the record representation for all objects, we used **tuples** for describing pictures; later, upon realizing that the essential properties of pictures in which we are interested are similar to those for graphs, we changed the BasicPicture trait to build upon sets of boxes (nodes) and sets of arrows (edges). We recursively generate values for pictures with *create_picture*, *insert_box* and *insert_arrow* instead of a **tuple of** constructor. This change also made it easier to prove a consequence of this trait stated in its **implies** clause (see discussion in Section 4.2.4 on **implies** clauses).

#### 4.2.3. Generalize.

*Remove pre-conditions; handle error conditions.*

Removing a pre-condition implies that an operation can be invoked from more states than before. An earlier specification of the MoveBoxes operation had a pre-condition that only one object be selected; hence moving objects had to be done by selecting and moving them one at a time. Removing the pre-condition allows MoveBoxes to operate on a set of objects, e.g., the set of selected objects. Removing a pre-condition in general may imply that the implementation now needs to check explicitly for a case that was safely ignored before. It often results in introducing an exceptional termination case in the post-condition, which now the implementor must **signal** under the appropriate condition.

*Add type genericity.*

An earlier specification had an operation each for deleting boxes, deleting arrows, copying boxes, copying arrows, changing attributes of a box, and changing attributes of an arrow. Realizing that deleting, copying, and changing attributes are operations whose effects are independent of whether their argument is a box or arrow led us to add a more generic object type to stand for boxes and arrows. This change led to the current version which has only three generic operations instead of six specific ones. In reality, the actual implementation influenced our addition of type genericity to these operations: Garnet [M+89], the graphical system on which Miró runs, supports the notion of an *object* on which many common editing operations can be performed.

In this instance, adding type genericity had the side effect of changing the underlying traits as well. We introduced a union sort for objects to stand for the corresponding box or arrow sorts. This change actually complicated the trait equations since we had to define some trait operators as lengthy "case" statements depending on an object's tag (recall the equations for *delete_objs* in Section 3.1); however, this complexity in the trait is completely hidden at the interface level in the specification. For example, what appears in the ChangeAttribute operation as a simple function call to the *change_attr* trait operator actually expands to its half-page long, nested **if-then-else** definition.

*Add parameters to operations/operators.*

We add parameters to an operation at the interface level, and similarly an operator at the trait level, to capture intentional incompleteness in a specification (delaying a design decision till binding time) and to add a degree of generality to the specification. For example, in the first version of the specification we explicitly modeled the mouse as an input device that provided coordinate pair information. In post-conditions of the earlier versions of DrawBox and DrawArrow we referred explicitly to this mouse's coordinate information to determine where the box or arrow is to be drawn. In contrast, our current version of the entire editor specification makes no reference to a mouse at all. DrawBox and DrawArrow each take coordinate pairs as parameters and do not care what or how that information is generated.

#### 4.2.4. Improve presentation.

*Add invariants.*

Adding an invariant to a Larch interface specification is a way to capture explicitly in one place information that may be either implicit or scattered throughout the rest of the interface specification. In the Editor specification the invariant that all pictures be well-formed arose from factoring out what in an earlier version of the specification was in each of the operation's pre- and post-conditions. It is an example of an explicit statement of a property that the Miró editor is required to maintain. Adding this invariant had the side benefit of making the specification tidier.

*Add* **implies** *to traits.*

Stating explicit consequences of a trait is similar to stating explicit **invariant**s in an interface. After writing a barebones trait with just equations and perhaps **generated by** and **partitioned by** clauses, often we add an **implies** clause as a way of formally documenting what else a trait intends to state. For example, since we can think of each Miró picture as being a graph where boxes are nodes and arrows are edges, we add the following **implies** clause to the WFPicture trait:

**implies**

$$Graph(B, A, Pic, create\_picture \textbf{ for } empty, insert\_box \textbf{ for } addNode,$$
$$insert\_arrow \textbf{ for } addEdge, boxes \textbf{ for } nodes, arrows \textbf{ for } edges)$$

where *Graph* is a Larch Handbook trait. This implication is a strong assertion. In terms of LSL semantics, it says that the first-order theory of graphs is a subset of the theory of well-formed instance pictures; any property of graphs holds for well-formed instance pictures.

We can take this one step further in our WFInstancePic trait. Since we add the well-formed requirement that each arrow goes from a User box to a File box, we can also add the implication that a well-formed instance picture is a bipartite graph. Here is a case where explicitly stating this as an implication caused us to realize a discrepancy between the specification and the implementation; the editor does not currently enforce this well-formedness property.

*Make more readable and modular.*

Many of the minor iterations of each major version of the specification improved the readability of the specification. Factoring large traits into smaller, reusable ones yields more modular specifications. For example, we define single traits each for boxes, arrows, and well-formed pictures; we reuse each twice: once for their version for instance pictures and once for constraint pictures. We also introduce many trait operators solely to shorten interface pre- and post-conditions.

## 5. Further Work and Final Remark

Based on our more realistic model of software development drawn in Figure 3, we suggest two practical directions for the software engineering community to pursue: (1) Begin to use formal methods now. The earlier the better, but it's never too late. Many specification case studies published are *post facto* specification exercises (e.g., [Win90, MS84]). A project can gain benefits from using formal methods no matter when they are introduced in the overall project's development plan. (2) Expand CASE technology to include tools that support formal methods. Integrating syntax and type checkers is easy; integrating semantic analyzers like proof checkers, proof debuggers, and theorem provers is harder, but possibly where the greatest payoff from using formal methods lies.

Our purpose in writing this paper is to provide concrete advice to practitioners on how formal methods can be integrated unintrusively in software engineering today. With the growing concern by the software engineering community to alleviate the "software crisis" by trying formal methods and with the growing interest by industry in formal methods, we hope our advice helps to bridge the gap between inventors of formal methods and their intended users.

## 6. Acknowledgments

We are grateful to Rick Lerner for his implementation of the GIL checker; to Rick, Allan Heydon, Mark Maimone, and Doug Tygar for their helpful comments on the Miró editor specification; and to Rod Nord for integrating the Larch tools with STP.

## A Miró Editor Specification

This appendix contains in their entirety the traits and interface specification for the Miró editor discussed in the paper. Additional traits included by those traits are not listed, but can be found in [Zar91].

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

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

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

  **asserts**
    $\forall\ b : B$

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

% ————————
% BASIC PICTURE
% ————————
*BasicPicture*(*Pic*) : **trait**

  **includes** *Box*, *Set*(*B*, *BSet*), *Arrow*, *Set*(*A*, *ASet*)

  **introduces**
    $create\_picture : \rightarrow Pic$
    $insert\_box : Pic, B \rightarrow Pic$
    $insert\_arrow : Pic, A \rightarrow Pic$
    $move\_all\_boxes : Pic, CoordPair \rightarrow Pic$
    $copy\_picture : Pic \rightarrow Pic$
    $pic\_union : Pic, Pic \rightarrow Pic$
    $delete\_box : Pic, B \rightarrow Pic$
    $delete\_arrow : Pic, A \rightarrow Pic$

    $boxes : Pic \rightarrow BSet$ % observers
    $arrows : Pic \rightarrow ASet$
    $arrows\_attached\_to\_box : Pic, B \rightarrow ASet$
    $arrows\_attached\_to\_boxes : Pic, BSet \rightarrow ASet$

$is\_on\_a\_box : CoordPair, Pic \rightarrow Bool$
$box\_at : CoordPair, Pic \rightarrow B$

**asserts**

$Pic$ **generated by** $create\_picture, insert\_box, insert\_arrow$
$Pic$ **partitioned by** $boxes, arrows$

$\forall\ pic, pic_1, pic_2 : Pic, cp, delta : CoordPair, b, b_1 : B, a, a_1 : A, bs : BSet, as : ASet$

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

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

% Union 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.
$delete\_box(insert\_box(pic, b), b_1) ==$
$\quad$ **if** $b = b_1$ **then** $pic$
$\quad$ **else** $insert\_box(delete\_box(pic, b_1), b)$
$delete\_box(insert\_arrow(pic, a), b) ==$
$\quad insert\_arrow(delete\_box(pic, b), a)$

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

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

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

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

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

$\mathit{box\_at}(\mathit{cp}, \mathit{insert\_arrow}(\mathit{pic}, a)) == \mathit{box\_at}(\mathit{cp}, \mathit{pic})$

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

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

% ——————————
% OBJECT
% ——————————
$\mathit{Obj}$ : **trait**

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

$Ob$ **union of** $box : B, arrow : A$

**introduces**

$objects : Pic \rightarrow ObjSet$
$boxes : ObjSet \rightarrow BSet$
$arrows : ObjSet \rightarrow ASet$
$toggle\_in : ObjSet, Ob \rightarrow ObjSet$

**asserts**
$\forall\ b : B, bs : BSet, a : A, as : ASet, obj : Ob, os : ObjSet, pic : Pic$
% 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** $obj \in os$ **then** $os - \{obj\}$
**else** $os \cup \{obj\}$

**implies**
**converts** $objects, boxes : ObjSet \rightarrow BSet, arrows : ObjSet \rightarrow ASet, toggle\_in$

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

**introduces**
$arrows\_attached : Pic, ASet \rightarrow Bool$
$arrow\_attached : Pic, A \rightarrow Bool$
$well\_formed : Pic \rightarrow Bool$

$delete\_objs : Pic, ObjSet \rightarrow Pic$
$delete\_wf\_box : Pic, B \rightarrow Pic$
$delete\_wf\_arrow : Pic, A \rightarrow Pic$
$delete\_arrows : Pic, ASet \rightarrow Pic$

$extract\_wf : ObjSet \rightarrow Pic$

**asserts**

$\forall\ pic : Pic, b : B, a : A, as : ASet, obj : Ob, os : ObjSet$

% arrows_attached(pic,as) iff each arrow in as (set of arrows) is attached .
$arrows\_attached(pic, \{\}) == true$
$arrows\_attached(pic, insert(as, a)) ==$
$\quad arrow\_attached(pic, a) \wedge arrows\_attached(pic, as)$
% an arrow is attached in a picture if both of its boxes are in the picture.
$arrow\_attached(pic, a) ==$
$\quad (((a.to\_box) \in boxes(pic)) \wedge ((a.from\_box) \in boxes(pic)))$

$well\_formed(pic) == arrows\_attached(pic, arrows(pic))$

% Delete a set of objects.
$delete\_objs(pic, \{\}) == pic$
$delete\_objs(pic, insert(os, obj)) ==$
$\quad$ **if** $tag(obj) = box$ **then**
$\quad delete\_objs(delete\_wf\_box(pic, obj.box), os)$
$\quad$ **else** % is an arrow
$\quad delete\_objs(delete\_wf\_arrow(pic, obj.arrow), os)$

% When deleting a box, delete all attached arrows first.
% If box is not in picture, just return picture.
$delete\_wf\_box(pic, b) ==$
$\quad$ **if** $b \in boxes(pic)$ **then**
$\quad delete\_box(delete\_arrows(pic,$
$\quad arrows\_attached\_to\_box(pic, b)), b)$
$\quad$ **else** $pic$

% Check to see if arrow is in picture first.
$delete\_wf\_arrow(pic, a) ==$
$\quad$ **if** $a \in arrows(pic)$ **then** $delete\_arrow(pic, a)$
$\quad$ **else** $pic$

% Delete a set of arrows.
$delete\_arrows(pic, \{\}) == pic$
$delete\_arrows(pic, insert(as, a)) ==$
$\quad delete\_arrows(delete\_wf\_arrow(pic, a), as)$

% Extract WF: Keep all boxes and arrows in os whose boxes are also in os.
$boxes(extract\_wf(os)) == boxes(os)$
$a \in arrows(extract\_wf(os)) ==$
$\quad (a \in arrows(os)) \wedge$
$\quad ((a.to\_box) \in boxes(os)) \wedge$
$\quad ((a.from\_box) \in boxes(os))$

**implies**

% a well-formed picture is a graph
$Graph(B, A, Pic, create\_picture$ **for** $empty, insert\_box$ **for** $addNode,$
$insert\_arrow$ **for** $addEdge, boxes$ **for** $nodes, arrows$ **for** $edges)$

$\forall\ pic : Pic, objs : ObjSet$

% 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.
$boxes(delete\_objs(pic, objs)) == boxes(pic) - boxes(objs)$
$arrows(delete\_objs(pic, objs)) ==$
$\quad arrows(pic) - (arrows(objs) \cup$
$\quad arrows\_attached\_to\_boxes(pic, boxes(objs)))$

% Delete_objs maintains well-formedness .
$(well\_formed(pic) \wedge objs \subseteq objects(pic))$
$\Rightarrow well\_formed(delete\_objs(pic, objs))$

**converts** *arrows_attached*, *arrow_attached*

% ——————————
% WELL-FORMED INSTANCE PICTURE
% ——————————
*WFInstancePic* : **trait**
**includes** *InstanceBox*, *InstanceArrow*,
*WFPicture*(*IPic*, *create_instance_pic* **for** *create_picture*, *IB* **for** *B*,
*IBSet* **for** *BSet*, *Str* **for** *Label*, *IA* **for** *A*, *IASet* **for** *ASet*,
*IO* **for** *Ob*, *IOSet* **for** *ObjSet*)

**introduces**
$create\_ibox : CoordPair, CoordPair, Str, BoxType \rightarrow IB$
$create\_iarrow : IB, IB, Parity, Str \rightarrow IA$
$users\_to\_files : IPic, IASet \rightarrow Bool$
$user\_to\_file : IPic, IA \rightarrow Bool$

**asserts**
$\forall\ ipic : IPic, cp_1, cp_2 : CoordPair, parity : Parity, label : Str,$
$\quad b, b_1 : IB, bt : BoxType, as : IASet, a : IA$

% Default values for thickness (thin), starred (false).
$create\_ibox(cp_1, cp_2, label, bt) ==$
$\quad [cp_1, cp_2, label, thin, false, bt]$

% Default values for thick (thin), starred (false), and kind (syn).
$create\_iarrow(b, b_1, parity, label) ==$
$\quad [syn, label, parity, thin, false, b, b_1]$

$users\_to\_files(ipic, \{\}) == true$
$users\_to\_files(ipic, insert(as, a)) ==$
$\quad user\_to\_file(ipic, a) \wedge$
$\quad users\_to\_files(ipic, as)$
$user\_to\_file(ipic, a) == (a.from\_box).box\_type = User \wedge$
$\quad (a.to\_box).box\_type = File$

% An additional well-formedness condition for instance pictures.
$well\_formed(ipic) == users\_to\_files(ipic, arrows(ipic))$

**implies**
**converts** *create_ibox*, *create_iarrow*, *well_formed*

```
% ———————
% 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


%——————————————————————-
% The Editor Interface Specification
%——————————————————————-

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 well_formed(e.picture)
```

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

**operation** CreateEditor (posn, size : Cp)
**requires** true
**ensures** **newobj** ($e_{obj}$) $\wedge$
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**
$\exists$ b:Box
**newobj** (b) $\wedge$
b!post = create_box(e.picture_type,cp1, cp2, label, e.thickness, e.starred, bt) $\wedge$
e'.picture = insert_box(e.picture, b)

**operation** DrawArrow (cp1, cp2 : Cp, label : Str)
**requires** e.object_type = arrow $\wedge$
is_on_a_box(cp1,e.picture) $\wedge$ is_on_a_box(cp2,e.picture)
**modifies** ($e_{obj}$)
**ensures**
$\exists$ a:Arrow
**newobj** (a) $\wedge$
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) $\wedge$
e'.picture = insert_arrow(e.picture, a)

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

**operation** GroupSelect (os : ObjSet)
**requires** os $\subseteq$ objects(e.picture)
**modifies** ($e_{obj}$)
**ensures** e'.selected_objs = (e.selected_objs $\cup$os)

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

**operation** MoveBoxes (delta : Cp)
**requires** true
**modifies** ($e_{obj}$)
**ensures**
$\forall$ b:Box
(b $\in$ boxes(e.selected_objs) =>
b!post = set_pos(b!pre, (b!pre).pos+delta)) $\wedge$
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) $\wedge$
e'.selected_objs = $\{\}$:OS

**operation** DeleteObjs ( )
**requires** true
**modifies** ($e_{obj}$)
**ensures** e'.picture = delete_objs(e.picture, e.selected_objs) $\wedge$
e'.selected_objs = $\{\}$:OS

**operation** CopyObjs (delta : Cp)
**requires** true
**modifies** ($e_{obj}$)
**ensures**
$\exists$ newpic : Picture $\forall$ o:ObjType
**newobj** (newpic) $\wedge$
newpic!post = copy_picture(extract_wf(e.picture_type,e.selected_objs)) $\wedge$
(o $\in$ objects(newpic!post) => **newobj** (o)) $\wedge$
e'.picture = pic_union(e.picture,
move_all_boxes(newpic!post, delta)) $\wedge$
e'.selected_objs = $\{\}$:OS

**operation** ChangeAttribute (o:ObjType, attr:Label, val:Value)
**requires** valid_attr(attr, o) $\wedge$ 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) $\wedge$
e'.selected_objs = $\{\}$:OS

## References


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

[DeM78] T. DeMarco. *Structured Analysis and System Specification*. Yourdon Press, New York, 1978.

[Dij76] E.W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[ED89] C. Easteal and G. Davies. *Software Engineering: Analysis and Design*. McGraw-Hill, London, 1989.

[FGJM85] K. Futatsugi, J.A. Goguen, J.-P. Jouannaud, and J. Meseguer. Principles of OBJ2. In *Proceedings of ACM POPL*, pages 52–66, 1985.

[GH80] J.V. Guttag and J.J. Horning. Formal specification as a design tool. In *Proceedings of the Seventh Symposium on Principles of Programming Languages*, pages 251–261, Las Vegas, Nevada, January 1980.

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

[GHW85a] John V. Guttag, James J. Horning, and Jeannette M. Wing. The Larch family of specification languages. *IEEE Software*, pages 24–36, September 1985.

[GHW85b] J.V. Guttag, J.J. Horning, and J.M. Wing. Larch in five easy pieces. Technical report, DEC-SRC, 1985.

[Gri81] D. Gries. *The Science of Programming*. Springer-Verlag, New York, 1981.

[Gut75] J.V. Guttag. *The Specification and Application to Programming of Abstract Data Types*. PhD thesis, University of Toronto, Toronto, Canada, September 1975.

[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.

[Jac83] M. Jackson. *System Development*. Prentice-Hall, Englewood Cliffs, 1983.

[Jon86] C.B. Jones. *Systematic Software Development Using VDM*. Prentice-Hall International, 1986.

[Ler91] R.A. Lerner. *Modular Specifications of Concurrent Programs*. PhD thesis, Carnegie Mellon University, 1991.

[M+89] B.A. Myers et al. The Garnet toolkit reference manuals: Support for highly-interactive , graphical user interfaces in Lisp. Technical Report CMU-CS-89-196, Carnegie Mellon University, November 1989.

[MS84] C. Morgan and B. Sufrin. Specification of the UNIX filing system. *IEEE TSE*, 10(2):128–142, 1984.

[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.

[Myn90] B.T. Mynatt. *Software Engineering with Student Project Guidance*. Prentice-Hall, Englewood Cliffs, 1990.

[Pfl91] S.L. Pfleeger. *Software Engineering: The Production of Quality Software*. Macillan, New York, 1991. Second edition.

[Ros77] D.T. Ross. Structured analysis (SA): A language for communicating ideas. *IEEE TSE*, pages 16–34, January 1977.

[Ros85] D.T. Ross. Applications and extensions of SADT. *IEEE Computer*, pages 25–34, April 1985.

[Spi88] J.M. Spivey. *Introducing Z: a Specification Language and its Formal Semantics*. Cambridge University Press, 1988.

[Win85] J.M. Wing. Specification firms: A vision for the future. In *Proceedings of the Third International Workshop on Software Specification and Design*, pages 241–243, 1985.

[Win90] J. Wing. Using Larch to specify Avalon/C++ objects. *IEEE TSE*, 16(9):1076–1088, September 1990.

[YC79] E. Yourdon and L. Constantine. *Structured Design*. Prentice-Hall, Englewood Cliffs, 1979.

[Zar91] A.M. Zaremski. A Larch specification of the Miro editor. Technical Report CMU-CS-91-111, CMU, 1991.

[Zav72] P. Zave. An operational approach to requirements specification for embedded systems. *IEEE Trans. Software Eng.*, 8(3):250–269, May 1972.