# A Framework for Specifying and Visualizing Architectural Designs

Richard C. Holt and Spiros Mancoridis
Department of Computer Science
University of Toronto
Canada

**Abstract**

Architectural designs specify the components of a software system, their interfaces, and their interrelationships. Module Interconnection Languages (MILs) are useful for specifying architectural designs, but lack an intuitive visual representation similar to the visual design notations found in CASE tools. This paper presents a framework for formally defining the syntax and semantics of languages for specifying and visualizing architectural designs. Also described are an instance and prototype implementation of this framework consisting of two languages: one for specifying designs and one for visualizing them.

## 1 Background

One of the phases of the software development process is the specification of the system's components, their interfaces, and their interrelationships. Throughout this paper, we will refer to the product of this phase as the *architectural design*.

Our definition of architectural design is close to the definition of *software architecture* given by Schwanke, Altucher, and Platoff [15], who define software architecture as the permitted or allowed set of connections among components. More recent research in software architecture has led to definitions that subsume the one given by Schwanke et al. More specifically, Perry and Wolf [13] argue that there is more to software architecture than simply components and their connections. This argument is consistent with the approach of Garlan and Shaw [5], who define software architecture as the design and specification of the overall system structure. Among their extensive list of structural issues, which includes protocols for communication, physical distribution, and performance, Garlan and

Shaw mention the composition of design elements. In this paper, we use the term architectural design, instead of software architecture, and concentrate on components and their interrelations, rather than on interfaces.

In current practice, architectural designs are commonly specified informally in English text accompanied by diagrams. Although these specifications act as a useful reference for developers and maintainers, their informality implies that they cannot be mechanically checked for syntactic and semantic consistency.

*Module Interconnection Languages (MILs)*, such as those defined by DeRemer and Kron [3], Cooprider [2], and Tichy [18], represent early attempts to define languages for specifying architectural designs. MILs are layered on top of common programming languages. Their advantage is that they closely couple the design specification to the source code, making the specification amenable to mechanical processing. (Specifications can be compiled and executed.) MILs, however, lack the intuitive visual representation that designers are accustomed to using.

Visual notations[1], such as those found in CASE tools, evolved separately from MILs as a set of diagrammatic conventions to support a rich class of design approaches. Examples of such notations are Jackson's System Design [7], Booch's Object–Oriented Design [1], and Data–Flow Diagrams from Yourdon and Constantine [20]. These visual notations are intuitive but often informal, closely tied to particular software development process, and detached from the implementation programming language.

There has been an attempt to formalize some of these visual notations using the *theory–model* paradigm. In particular, Ryman, Lamb, and Jain [14] specified a theory for the Jackson System Design notation and subsequently verified that examples (models) of designs satisfied this theory. Their long–term goal is to formalize a large set of these visual notations and include these formalisms in a library of design theories.

Recently, there have been efforts to formally define visual notations that are more like MILs and less like the visual CASE notations. These notations are not tied to a particular software development process, and have explicit mechanisms for coupling design diagrams to source code. Examples of such notations are Penny's *Software Landscape* [11], with related work by Holt, Godfrey, and Mancoridis [12, 6], and *ARCH* defined by Schwanke, Altucher, and Platoff [15].

# 2 Overview of the Framework

Our work can be thought of as a formal approach to MILs, design theories, and visual CASE notations, which integrates these into a single framework. This framework is illustrated in Figure 1.

A particular MIL has a syntax, given in the oval at the top left of Figure 1. A design theory corresponds to the lower left box. The semantics of a MIL specification is given by mapping its syntax to a model in the design theory[2]. The design theory, expressed in

[1]We assume that *languages* are always textual; hence, we use the term *notations* when referring to diagrammatic or visual representations.

[2]In this paper, we consider that the semantics of a MIL is given by translating each MIL specification to a corresponding model in the design theory. At a separate level, not addressed in this paper, one must

mathematical set theory, is used to describe the components, their interrelationships, and their constraints.

Separately, we consider diagrammatic approaches, which are shown on the right in Figure 1. The oval on the top right corresponds to a drawing language in which we specify how to draw CASE diagrams. The semantics of the drawing language is given in terms of a simple set–based theory, which corresponds to the bottom right box in Figure 1. Models in this theory have an obvious interpretation as CASE diagrams consisting of boxes and arrows.

The ovals in Figure 1, which give the syntax of the MIL and the drawing language, are connected by an arrow, indicating a two–way mapping between them. This mapping allows us to interpret a CASE diagram as a MIL specification and hence to give it semantics in design theory. Conversely, it allows us to mechanically produce a CASE diagram from a MIL specification.

Our framework provides the advantage of neatly separating our concerns about various aspects of architectural design. We can concentrate on theories of design with its questions, such as the details of rules for visibility among modules, without concern for the syntax of a MIL or CASE diagrammatic conventions. We can separately design the syntax of a MIL, which many programmers seem to feel is the natural way to express higher level constraints on their code. We can separately investigate CASE notations that are intuitive and practical for capturing a designer's intentions. And we can separately deal with the mechanics of producing diagrams on a computer screen and the means of updating and navigating through these diagrams. We hope that our framework will help to show how to use formal approaches in each of these areas of concern in the design phase of software development.

As an example of a framework instance, we define a language for specifying architectural designs called SIL (Subsystem[3] Interconnection Language), as well as a drawing language called DL (Drawing Language), for specifying diagrams comprising annotated, colored boxes and arrows. A number of the properties of SIL and DL have evolved from ideas described in Penny's thesis [11]. Our long–term goal is to define more elaborate languages and semantic theories for specifying architectural designs (as outlined in Section 9.)

Thus far we have given the motivation behind our work, described its relationship to previous research, and presented an overview of our framework. The rest of the paper is structured as follows: Section 3 presents a diagrammatic example of an architectural design, and then presents this example as a SIL and a DL specification. Section 4 presents an overview of the mathematical notations used in our formalisms. Sections 5 and 6 present the syntax and semantics of SIL and DL, respectively. Section 7 defines the syntactic mapping between them. Section 8 presents an overview of the prototype implementation to support them. Finally, Section 9 presents a summary of the paper and our directions for further research.

also give the semantics of the program (effect of execution) that is based on a particular design model.

[3] We will refer to composite modules as *subsystems* and atomic units as *modules*. Hence, our preference for the term SIL rather than the more traditional MIL.

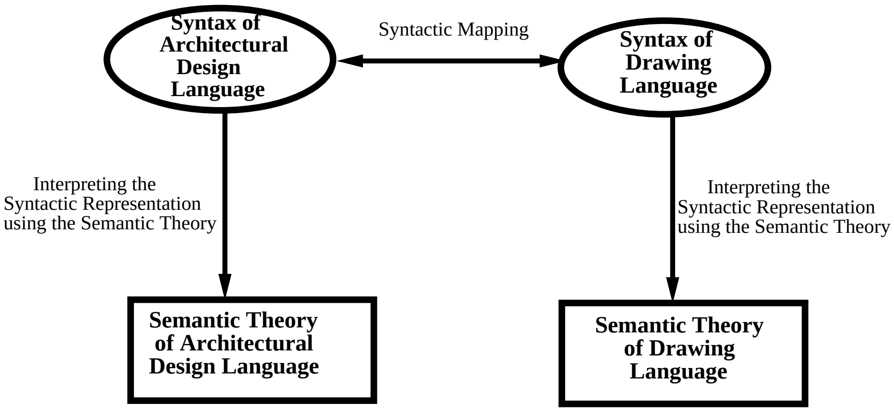


Figure 1: **Overview of the Framework**

# 3 An Example

In this section we present an example architectural design depicted in Figure 2. White boxes represent *subsystems*, dark[4] boxes represent *modules*, dark arrows represent *import* relations, light arrows represent *use* relations, nested boxes represent *contain* relations, and light frames around boxes represent *export* relations.

Nested boxes are used as an aggregation mechanism so that hierarchies of subsystems can be specified. Dependencies between subsystems are specified with the *import* relation. The *export* relation is used as an information hiding mechanism. Finally, the *use* relation is employed to specify dependencies between atomic entities.

This architectural design consists of a subsystem *MAIN* that contains two subsystems: *DEMO* and *LIB*. The *stackDemo* module in *DEMO* uses the exported *stack* module in *LIB*. For this usage to be allowed, the parent subsystem of *stackDemo (DEMO)* must import the stack module. This constraint is imposed by the semantics of our SIL, presented in Section 5.2. The *LIB* subsystem contains the *node* and *stack* modules. The arrow between them indicates that *stack* uses *node*.

In the diagram, each box has a coordinate system, and all of the values[5] specified in the diagram are in local coordinates. Each coordinate system has an adjustable origin and a scaling factor. The origin is the bottom left corner of the box, plus an offset to facilitate the movement (panning) of its contents. The scaling factor is used for enlarging or reducing (zooming) the size of its contents. Both the adjustable origin and scale factor are used for layout and navigation purposes (panning and zooming), which are features of our prototype visual editor for specifying designs similar to the one in Figure 2.

The boldface numbers in brackets specify the $x$ and $y$ coordinates of the bottom left corner of each box. The numbers on the sides of the boxes represent the height and width of the box. Because we do not discuss panning and zooming in this paper, for simplicity of presentation, we assume that the scaling factors are all 1.0 and the offset of the origin

[4]In a color diagram the dark boxes would have been blue, light arrows blue, and light frames green.
[5]These values do not appear in actual visualizations but are shown here for illustration purposes.

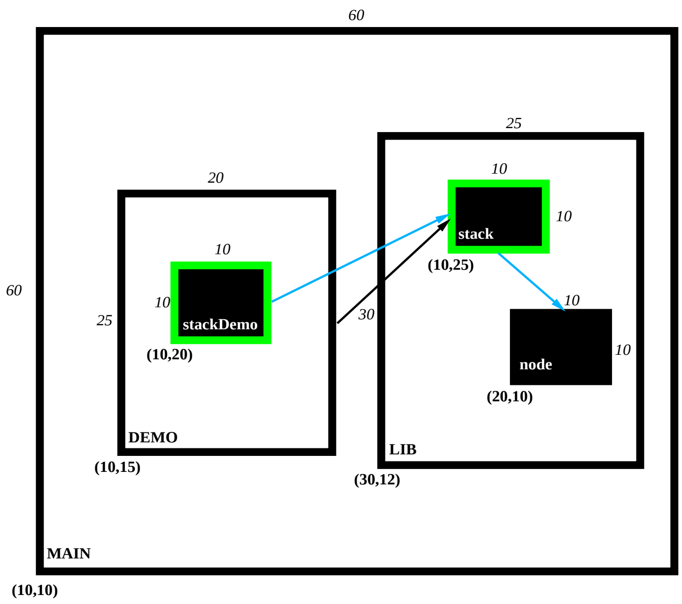


Figure 2: **Example of a Visualization of an Architectural Design**

of each coordinate system is 0.

## 3.1 Example in SIL

The architectural design in Figure 2 is represented below as a SIL specification. Each box in the figure corresponds to either a `subsystem` (white box) or a `module` (dark box). Note that modules have no contents, other than a possible `use` list. The contents of these modules are source statements written in a programming language. Because our languages are concerned with the architectural design, the source statements have been omitted for simplicity.

```
subsystem MAIN
    contain DEMO, LIB
end MAIN

module stackDemo
    use stack
end stackDemo

subsystem LIB
    export stack
    contain stack, node
end LIB

subsystem DEMO
    import stack
    export stackDemo
    contain stackDemo
end DEMO

module stack
    use node
end stack

module node
end node
```

## 3.2 Example in DL

We now present the DL specification of the design in Figure 2. Each box in the figure has a corresponding `box` clause in the DL specification. Nested boxes are specified using the `contain` clause, and arrows between boxes are specified using the `edge` clause.

For example, the *DEMO* box in Figure 2 corresponds to a box clause in the DL specification. The *stackDemo* box appears as a contain clause within box DEMO. The x, y, width, height attributes of the contain clause are used for positioning the contained box within the container box's coordinate system. Hence, box DEMO specifies that the bottom left corner of the stackDemo box is at position (10,20). The attributes of the edge clause are used for specifying: the color of the edge (edgecolor), the identifier of the box that the edge points to (target), and the side of the box from which the edge exits and enters (exit/enter side) along with the position of contact between the box and the ends of the edge. This position is given as a percentage of the side of a box (that is, 0.5 means that the edge contacts the middle of a box side.) Following is the complete DL specification:

```
box MAIN
    xorigin 0 yorigin 0
    scale 1
    boxcolor white
    titlecolor black

    contain DEMO
        x 10 y 15
        width 20 height 25
        framecolor black
    end DEMO

    contain LIB
        x 30 y 12
        width 25 height 30
        framecolor black
    end LIB
end MAIN
```

```
box DEMO
    xorigin 0 yorigin 0
    scale 1
    boxcolor white
    titlecolor black

    contain stackDemo
        x 10 y 20
        width 10 height 10
        framecolor green
    end stackDemo

    edge
        edgecolor black
        target stack
        exit right side 0.5
        entry left side 0.5
end DEMO
```

```
box LIB
    xorigin 0 yorigin 0
    scale 1
    boxcolor white
    titlecolor black

    contain stack
        x 10 y 25
        width 10 height 10
        framecolor green
    end stack

    contain node
        x 20 y 10
        width 10 height 10
        framecolor black
    end node
end LIB
```

```
box stackDemo
    xorigin 0 yorigin 0
    scale 1
    boxcolor blue
    titlecolor white

    edge
        edgecolor blue
        target stack
        exit side right position 0.5
        entry side left position 0.5
end stackDemo
```

```
box stack
    xorigin 0 yorigin 0
    scale 1
    boxcolor blue
    titlecolor white

    edge
        edgecolor blue
        target node
        exit side bottom position 0.5
        entry side top position 0.5
end stack
```

```
box node
    xorigin 0 yorigin 0
    scale 1
    boxcolor blue
    titlecolor white

end node
```

Before proceeding with the definitions of the languages, we give an overview of the mathematical notations used in our formalisms.

# 4 Mathematical Notations

We use regular expressions for specifying the symbols and extended Backus–Naur Form (BNF) for specifying the grammars of both the SIL and DL languages. In formalizing the semantics of the languages, we will use mathematical set theory. Entities in both languages will then be sets, and interrelations between entities will be modelled as relations among sets. An appropriate alternative would have been to formalize the semantics of the languages using a formal specification language based on set theory, such as $Z$ [16].

If $A$ and $B$ are sets, $A \times B$ will denote their cartesian product. It is convenient to reference the different sets that participate in a cross product. To do so, we will specify the cross product together with a naming convention for the product operands, when necessary. Thus, if set $A$ is to be called the first attribute, *firstAttr*, of the cross product and set $B$ is to be called the second attribute, $secondAttr$, we will specify the cross product as *firstAttr:A* $\times$ *secondAttr:B*. The attribute names can then be used to access the attribute values for a specific member $t$ of the cross product; *t.firstAttr* refers to the first value of $t$, and $t.secondAttr$ refers to the second value. Next, we formally define the syntax and semantics of the SIL language.

# 5 Subsystem Interconnection Language (SIL)

The need for languages to support architectural design specification was first addressed by DeRemer and Kron [3] in their seminal paper on MILs. The authors stated that the activity of producing source code in a programming language is called *Programming–in–the–Small (PitS)*, whereas the activity of specifying the interconnections between PitS entities (that is, procedures, variables, types) is called *Programming–in–the–Large (PitL)*.

This section describes the syntax and semantics of our MIL, which is called SIL. Although most of the MILs described in the introduction have mechanisms for composing systems made up of nested modules, the only relationships between the modules are what PitS entities are *required* by a module and what PitS entities are *provided* by a module to other modules. SIL relations do not conform to the provides–requires paradigm because they are not based on relationships between PitS elements but are, rather, based on relationships between PitL elements such as whole modules and subsystems. Another difference is that SIL is not restricted to only two kinds of relations (provides and requires). It currently supports four kinds of relations *import*, *export*, *use*, and *contain* and potentially more, as mentioned in Section 9.

## 5.1 Syntax of SIL

The syntax of SIL is described using a set of symbols and a context–free grammar for constructing strings from these symbols.

### 5.1.1 Symbols

Following is the set of allowable symbols (*Tokens*) in the SIL language. Characters and character strings are written using the sans–serif font, names of regular expressions are written using the *italics* font. The vertical bar (|) symbol denotes disjunction, and the star symbol (*) denotes zero or more repetitions of an item.

$Token = SpecialSymbol \mid Keyword \mid Id$
$SpecialSymbol =$ ,
$Keyword =$ contain | end | export | import | module | subsystem | use
$Id = Letter\ (Letter \mid Digit)^*$
$Letter =$ a | ⋯ | z | A | ⋯ | Z
$Digit =$ 0 | ⋯ | 9

### 5.1.2 Grammar of SIL

Following is the context–free grammar of SIL, with the start symbol $S$. The words in *italics* represent non–terminal symbols, the words in sans–serif font represent terminal symbols, the lambda symbol ($\Lambda$) means the empty string, the vertical bar symbol (|) denotes disjunction, braces { } mean that an item can occur zero or more times, and brackets [ ] mean that an item is optional.

$S ::= subsystemDecl\ S \mid moduleDecl\ S \mid \Lambda$

*subsystemDecl* ::=
    subsystem *Id*
        [import *IdList*]
        [export *IdList*]
        [contain *IdList*]
    end *Id*

*moduleDecl* ::=
    module *Id*
        [use *IdList*]
    end *Id*

*IdList* ::= *Id* {, *Id*}

The next section describes how meaning can be attached to these syntactic constructs by giving semantic interpretations to them.

## 5.2 Semantics of SIL

Each SIL specification is mapped to a set of entities and relations called a *configuration*. The set of all possible configurations is the semantic theory for SIL. There are also logic formulas for imposing constraints on the entities of the configuration.

### 5.2.1 Entities in the Semantic Theory for SIL

Entities are the basic items of interest in the semantic theory. Each entity has a unique name, and can either be atomic or composite. In our formalism, we have a single kind of atomic entity called *module*[6]. Atomic entities cannot contain other entities. Entity containment is a unique characteristic of composite entities. The only kind of composite entity is the *subsystem*. We define the set of all entities $E$ as the following cross product:

$$E = \text{EntityId:}Id \times \text{EntityKind:}\{\mathsf{subsystem}, \mathsf{module}\}$$

In this context, $Id$ is the set of all identifiers defined by the regular expression given in Section 5.1.1. The attributes of an entity $e$ are meant to denote the following: e.EntityId is the unique identifier of the entity, and $e.EntityKind$ is the type (subsystem or module) of the entity.

We define the set of all atomic and composite entities, $A$ and $C$ respectively, as follows:

$$A = \{\text{e} \mid (\text{e} \in E) \wedge (\text{e.EntityKind} = \mathsf{module})\}$$
$$C = \{\text{e} \mid (\text{e} \in E) \wedge (\text{e.EntityKind} = \mathsf{subsystem})\}$$

Having defined the various kinds of entities, we proceed with the definition of a configuration.

### 5.2.2 Configurations

Each subsystem interconnection configuration $K$ is a tuple:

$$K = (Entity, Contain, Import, Export, Use)$$

where *Entity*, *Contain*, *Import*, *Export* and *Use* are defined as follows:

$$Entity \subseteq AtomicEntity \cup CompositeEntity$$
$$AtomicEntity \subseteq A$$
$$CompositeEntity \subseteq C$$
$$Import \subseteq CompositeEntity \times Entity$$
$$Contain,\ Export \subseteq CompositeEntity \times Entity$$
$$Use \subseteq AtomicEntity \times AtomicEntity$$

Note that only composite entities can contain, import, or export other entities and that the *Use* relation is only defined for atomic entities. The SIL specification given in Section 3.1 can be described as the following configuration:

*Entity* = { (stackDemo, module), (stack, module), (node, module),
(MAIN, subsystem), (DEMO, subsystem), (LIB, subsystem) }

*Contain* = { ((MAIN, subsystem), (DEMO, subsystem)),
((MAIN, subsystem), (LIB, subsystem)), ((DEMO, subsystem), (stackDemo, module)),

[6] Other possible kinds of atomic entities we could have considered in our applications of this work include *classes*, *monitors*, and *C* files.

((LIB, subsystem), (stack, module)), ((LIB, subsystem), (node, module)) }

*Import* = { ((DEMO, subsystem), (stack, module)) }

*Export* = { ((DEMO, subsystem), (stackDemo, module)), ((LIB, subsystem), (stack, module)) }

*Use* = { ((stackDemo, module), (stack, module)), ((stack, module), (node, module)) }

A mapping for translating SIL specifications into these sets is described later on in Section 5.2.4. The definitions given thus far describe the objects of the semantic theory, but say nothing about the constraints imposed upon them.

### 5.2.3 Semantic Constraints

Not all configurations are well formed. This section defines a collection of relations and first–order logic formulas that specify the semantic constraints on configurations. Note that $R^+$ and $R^*$ are the transitive and reflexive transitive closures of relation R, respectively. We define three auxiliary relations:

$$\mathit{CanSee},\ \mathit{SeeAsSibling},\ \mathit{SeeAsImport} \subseteq \mathit{Entity} \times \mathit{Entity}$$

An entity can see its siblings or entities imported into its scope:
$(a\ \mathit{CanSee}\ b) \equiv (a\ \mathit{SeeAsSibling}\ b) \vee (a\ \mathit{SeeAsImport}\ b)$

An entity can see another entity as a sibling if and only if they both have a common parent:
$(a\ \mathit{SeeAsSibling}\ b) \equiv \exists\ p \cdot (p\ \mathit{Contain}\ a) \wedge (p\ \mathit{Contain}\ b)$

An entity $a$ can see another entity $b$ as a result of an import if and only if the parent $p$ of entity $a$ imports entity $b$:
$(a\ \mathit{SeeAsImport}\ b) \equiv \exists\ p \cdot (p\ \mathit{Contain}\ a) \wedge (p\ \mathit{Import}\ b)$

A configuration is well formed if and only if the semantic constraint on entity uniqueness as well as the semantic constraints on the *Contain*, *Import*, *Export*, and *Use* relations are satisfied.

#### Constraints on *Entity*

Entities must have a unique identifier:
$(a \in \mathit{Entity}) \wedge (b \in \mathit{Entity}) \wedge \neg\,(a = b) \Rightarrow \neg\,(a.\mathrm{EntityId} = b.\mathrm{EntityId})$

#### Constraints on *Contain*

An entity cannot directly or indirectly contain itself; that is, *Contain* is acyclic:
$(a\ \mathit{Contain}^+\ b) \Rightarrow \neg\,(a = b)$

An entity cannot be contained in more than one distinct entity:
$(a_1\ \mathit{Contain}\ b) \wedge (a_2\ \mathit{Contain}\ b) \Rightarrow (a_1 = a_2)$

The last two constraints impose a forest structure on the *Contain* relation.

#### Constraints on *Import*

Entities cannot directly import themselves:

$(\mathrm{a}\ \mathit{Import}\ \mathrm{b}) \Rightarrow \neg\ (\mathrm{a} = \mathrm{b})$

An entity cannot import any of its ancestors or descendents in the containment hierarchy:

$(\mathrm{a}\ \mathit{Import}\ \mathrm{b}) \Rightarrow \neg\ (\mathrm{a}\ \mathit{Contain}^{+}\ \mathrm{b}) \wedge \neg\ (\mathrm{b}\ \mathit{Contain}^{+}\ \mathrm{a})$

An entity can import an entity that it can see, or is exported from an entity it can see:

$(\mathrm{a}\ \mathit{Import}\ \mathrm{b}) \Rightarrow \exists\ \mathrm{p} \cdot (\mathrm{a}\ \mathit{CanSee}\ \mathrm{p}) \wedge (\mathrm{p}\ \mathit{Export}^{*}\ \mathrm{b})$

#### Constraints on *Export*

An entity can only export entities it contains:

$(\mathrm{a}\ \mathit{Export}\ \mathrm{b}) \Rightarrow \exists\ \mathrm{c} \cdot (\mathrm{a}\ \mathit{Contain}\ \mathrm{c}) \wedge (\mathrm{c}\ \mathit{Export}^{*}\ \mathrm{b})$

#### Constraints on *Use*

An atomic entity can use another atomic entity that is exported from an entity it can see:

$(\mathrm{a}\ \mathit{Use}\ \mathrm{b}) \Rightarrow \exists\ \mathrm{p} \cdot (\mathrm{a}\ \mathit{CanSee}\ \mathrm{p}) \wedge (\mathrm{p}\ \mathit{Export}^{*}\ \mathrm{b})$

Having defined both the syntax and semantics of SIL, we now describe how to map the syntactic structures of SIL to their corresponding objects in the semantic theory.

### 5.2.4 Mapping the Syntax of SIL to Objects in the Semantic Theory

Below, we show which subsystem $S$ or module $M$ in a SIL specification corresponds to which element of the sets of the semantic theory.

| Syntax | Semantic Theory |
|---|---|
| subsystem $S$ | $(S, \textsf{subsystem}) \in \mathit{CompositeEntity}$ |
| import $I_1, \cdots$ | $((S, \textsf{subsystem}), (I_1, I_1.\mathrm{EntityKind})) \in \mathit{Import} \cdots$ |
| export $E_1, \cdots$ | $((S, \textsf{subsystem}), (E_1, E_1.\mathrm{EntityKind})) \in \mathit{Export} \cdots$ |
| contain $C_1, \cdots$ | $((S, \textsf{subsystem}), (C_1, C_1.\mathrm{EntityKind})) \in \mathit{Contain} \cdots$ |
| end $S$ | |
| module $M$ | $(M, \textsf{module}) \in \mathit{AtomicEntity}$ |
| use $U_1, \cdots$ | $((M, \textsf{module}), (U_1, \textsf{module})) \in \mathit{Use} \cdots$ |
| end $M$ | |

This concludes the definition of the syntax and semantics of SIL. In the next section, we present the definition of a language for specifying drawings and, later, show how this drawing language relates to SIL.

# 6 Drawing Language (DL)

We believe that visual representations of subsystem interconnections are more intuitive and easier to understand. Hence, many developers prefer to view their architectural design in a diagrammatic form. This section defines the syntax and semantics of a language, called DL, for specifying drawings made up of annotated colored boxes and arrows. Other examples of a drawing language are the Graph Exchange Format (GXF) [4] for visualizing hierarchical graphs, and the Graph Description Language (GDL) [19] for visualizing graphs in three dimensions.

## 6.1 Syntax of DL

In this section we describe the syntax of DL which comprises a collection of symbols and a context–free grammar for constructing strings from these symbols.

### 6.1.1 Symbols

Following is the set of allowable symbols (*Token*) in DL. The plus symbol (+) denotes one or more repetitions of an item, and the query symbol (?) denotes that an item is optional. The other symbols are described in Section 5.1.1.

$Token = Color \mid Id \mid Keyword \mid Real \mid Side$
$Color = \textsf{black} \mid \textsf{white} \mid \textsf{blue} \mid \textsf{green}$
$Keyword = \textsf{box} \mid \textsf{boxcolor} \mid \textsf{contain} \mid \textsf{edgecolor} \mid \textsf{entry} \mid \textsf{exit} \mid \textsf{framecolor} \mid \textsf{height} \mid$
$\quad \textsf{position} \mid \textsf{scale} \mid \textsf{side} \mid \textsf{target} \mid \textsf{titlecolor} \mid \textsf{width} \mid \textsf{x} \mid \textsf{xorigin} \mid \textsf{y} \mid \textsf{yorigin}$
$Real = (+ \mid \textsf{-})?\ (Digit)^*\ (\textsf{.})?\ (Digit)^+\ ((\textsf{E} \mid \textsf{e})\ (+ \mid \textsf{-})?\ (Digit)^+)?$
$Side = \textsf{bottom} \mid \textsf{left} \mid \textsf{right} \mid \textsf{top}$

The definitions for *Id*, *Letter*, and *Digit* are identical to those given for the symbols of SIL in Section 5.1.1.

### 6.1.2 Grammar of DL

The context–free grammar of DL, with a start symbol $S$, is the following:

$S ::= \{boxDecl\}$

$boxDecl ::=$
  box *Id*
    xorigin *Real* yorigin *Real*
    scale *Real*
    boxcolor *Color*
    titlecolor *Color*
    {*edgeDecl*}
    {*containDecl*}
  end *Id*

$edgeDecl$ ::=

edge

edgecolor $Color$

target $Id$

exit side $Side$ position $Real$

entry side $Side$ position $Real$

$containDecl$ ::=

contain $Id$

x $Real$ y $Real$

width $Real$ height $Real$

framecolor $Color$

end $Id$

Next, we define the semantics of DL.

## 6.2 Semantics of DL

In this section, we present a semantic theory for the drawing language, DL. The theory is defined by a number of interrelated sets that represent boxes and edges with their positions and other attributes.

### 6.2.1 Boxes in the Semantic Theory for DL

Boxes in the semantic theory for DL are defined as follows:

$$B = \text{BoxId}{:}Id \times \text{Xorigin}{:}Real \times \text{Yorigin}{:}Real \times \text{Scale}{:}Real \times \text{BoxColor}{:}Color \times \text{TitleColor}{:}Color$$

The attributes of a box $b \in B$ are meant to denote the following: $b$.BoxId is the unique identifier associated with the box; $b$.Xorigin and $b$.Yorigin denote the origin of $b$'s coordinate system (translations and other transformations on $b$ may change the value of its origin); $b$.Scale denotes the scale factor for enlarging or reducing the contents of $b$; $b$.BoxColor is the color of $b$ (one of black or blue); and, $b$.TitleColor is the color of the title of $b$.

### 6.2.2 Configurations

Each drawing configuration $K$ is a tuple:

$$K = (Box,\ Edge,\ Contain)$$

Where $Box$, $Edge$, and $Contain$ are defined as follows:

$Box \subseteq B$

$Edge \subseteq$ SrcBoxId:$Id \times$ EdgeColor:$Color \times$ DestBoxId:$Id \times$ ExitSide:$Side \times$
ExitPos:$Real \times$ EntrySide:$Side \times$ EntryPos:$Real$

$Contain \subseteq$ ParentBoxId:$Id \times$ BoxId:$Id \times$ X:$Real \times$ Y:$Real \times$ Width:$Real \times$
Height:$Real \times$ FrameColor:$Color$

In this context, *Id*, *Color*, and *Side* are the sets defined by their corresponding regular expressions given in Section 6.1.1. The set *Real*, used in this section, is not the set of all string that represent the real numbers, but the actual set of real numbers.

The attributes of an edge $e \in Edge$ are meant to denote the following: $e$.SrcBoxId is the unique identifier of the box from which the edge exits; $e$.EdgeColor is the color of the edge; $e$.DestBoxId is the unique identifier of the box to which the edge enters; $e$.ExitSide is the side of the source box from which the edge exits; $e$.ExitPos is the position of the side of the box, given as a percentage of the side, from which the edge exits; $e$.EntrySide is the side of the target box to which the edge enters; and $e$.EntryPos is the position of the side of the box to which the edge enters.

The attributes of a contained box $b \in Contain$ are meant to denote the following: $b$.ParentBoxId is the unique identifier associated with the container box; $b$.BoxId is the unique identifier associated with the contained box; $b$.X and $b$.Y represent the bottom left corner of the contained box in the coordinate system of the container; $b$.Width and $b$.Height represent the width and height of the contained box in the coordinate system of the container; and $b$.FrameColor represents the frame color of the contained box.

The complete DL specification of the example in Section 3.2 can be described as the following configuration:

$Box = \{$
(MAIN, 0, 0, 1, white, black), (DEMO, 0, 0, 1, white, black), (LIB, 0, 0, 1, white, black),
(stackDemo, 0, 0, 1, blue, white), (stack, 0, 0, 1, blue, white), (node, 0, 0, 1, blue, white)$\}$

$Edge = \{$
(DEMO, black, stack, right, .5, left, .5) (stackDemo, blue, stack, right, .5, left, .5)
(stack, blue, node, bottom, .5, top, .5)$\}$

$Contain = \{$
(MAIN, DEMO, 10, 15, 20, 25, black), (DEMO, stackDemo, 10, 20, 10, 10, green),
(MAIN, LIB, 30, 12, 25, 30, black), (LIB, stack, 10, 25, 10, 10, green),
(LIB, node, 20, 10, 10, 10, black)$\}$

A mapping from DL specifications into these sets is described later on in Section 6.2.4. The definitions given thus far describe the objects but say nothing about the semantic constraints imposed upon them.

### 6.2.3 Semantic Constraints

Not all drawing configurations are well formed. This section defines constraints that govern whether a particular drawing is well formed or not. We define the following auxiliary functions:

*corner*, *inside*: *Real* $\times$ *Real* $\times$ *Box* $\rightarrow$ *Boolean*
*enclose*, *overlap*, *contain*, *edge*: *Box* $\times$ *Box* $\rightarrow$ *Boolean*

The *corner* function returns true if and only if *(x, y)* is one of the corners of box *a*:

$$corner\ (x, y, a) \equiv \exists\ t \cdot ((t \in Contain) \wedge (t.BoxId = a.BoxId) \wedge (x = t.X \wedge y = t.Y) \vee (x = t.X + t.Width \wedge y = t.Y) \vee (x = t.X \wedge y = t.Y + t.Height) \vee (x = t.X + t.Width \wedge y = t.Y + t.Height))$$

The *inside* function, in a configuration $K$, returns true if and only if *(x, y)* lies within box *a*:

$$inside\ (x, y, a) \equiv \exists\ t \cdot ((t \in Contain) \wedge (t.BoxId = a.BoxId) \wedge (t.X < x < t.X + t.Width) \wedge (t.Y < y < t.Y + t.Height))$$

The *enclose* function returns true if and only if boxes *a* and *b* have no overlapping boundaries:

$$enclose\ (a, b) \equiv \forall\ x, y \cdot corner\ (x, y, b) \Rightarrow inside\ (x * a.Scale + a.Xorigin,\ y * a.Scale + a.Yorigin,\ a)$$

The *overlap* function returns true if and only if boxes *a* and *b* contain common points:

$$overlap\ (a, b) \equiv \exists\ x, y \cdot \mathrm{inside}\ (x, y, a) \wedge \mathrm{inside}\ (x, y, b)$$

The *contain* function returns true if and only if box *a* directly contains box *b*:

$$contain\ (a, b) \equiv \exists\ C \cdot (C \in Contain) \wedge (C.ParentBoxId = a.BoxId) \wedge (C.BoxId = b.BoxId)$$

The *edge* function returns true if and only if there is an edge between boxes *a* and *b*:

$$edge\ (a, b) \equiv \exists\ E \cdot (E \in Edge) \wedge (E.SrcBoxId = a.BoxId) \wedge (E.DestBoxId = b.BoxId)$$

### Constraints on drawings

Each child box must be enclosed by its parent box:

$$contain\ (a, b) \Rightarrow enclose\ (a, b)$$

The boundaries of sibling boxes cannot overlap:

$$contain\ (a, b) \wedge contain\ (a, c) \Rightarrow \neg\ overlap\ (b, c)$$

A box cannot directly or indirectly contain itself; that is, *Contain* is acyclic:

$$contain\ (a, b) \Rightarrow \neg\ (a = b)$$

A box cannot be contained in more than one distinct boxes:

$$contain\ (a_1, b) \wedge contain\ (a_2, b) \Rightarrow (a_1 = a_2)$$

### Constraints for referential integrity

If *a* contains *b*, then both *a* and *b* must be boxes:

$$contain\ (a, b) \Rightarrow (a \in Box) \wedge (b \in Box)$$

If there is an edge between *a* and *b*, then both *a* and *b* must be boxes:

$$edge\ (a, b) \Rightarrow (a \in Box) \wedge (b \in Box)$$

Having defined both the syntax and semantics of DL, we now describe how to map the syntactic structures of DL to their corresponding objects in the DL semantic theory.

### 6.2.4 Mapping the Syntax of DL to Objects in the Semantic Theory

Below, we show which objects in a DL specification are mapped to which objects in the semantic theory.

| Syntax | Semantic Theory |
|---|---|
| box $B$ | $(B,$ |
| xorigin $XOR$ yorigin $YOR$ | $XOR$, $YOR$, |
| scale $SC$ | $SC$, |
| boxcolor $BC$ | $BC$, |
| titlecolor $TC$ | $TC) \in Box$ |
| | |
| edge | $(B,$ |
| edgecolor $EC$ | $EC$, |
| target $B_i$ | $B_i$, |
| exit side $XS$ position $XPOS$ | $XS$, $XPOS$ |
| entry side $ES$ position $EPOS$ | $ES$, $EPOS) \in Edge$ |
| $\vdots$ | $\vdots$ |
| | |
| contain $B_j$ | $(B, B_j$ |
| x $X$ y $Y$ | $X$, $Y$ |
| height $H$ width $W$ | $H$, $W$, |
| framecolor $FC$ | $FC) \in Contain$ |
| end $B_j$ | |
| $\vdots$ | $\vdots$ |
| end $B$ | |

The next section describes how the SIL and DL can be used in concert for specifying and visualizing architectural designs.

# 7 Mapping Between SIL and DL

This section presents a syntactic mapping between SIL and DL. This mapping forms the basis for the automatic translation, described in section 8, between specifications written in either of the two languages. The syntactic mapping from a SIL to a DL specification is, in general, 1–to–many because SIL specifications have no layout or color information. The mapping assigns default values to fields with no counterpart in the other language. The syntactic mapping from a DL specification to a SIL one is many–to–1. Below is the mapping between a SIL subsystem and its corresponding DL box:

| SIL | DL |
|---|---|
| subsystem $S$ | box $S$ |
| | xorigin 0 yorigin 0 |
| | scale 0 |
| | boxcolor white |
| | titlecolor black |
| | |
| import $I_1, \cdots$ | edge |
| | edgecolor black |
| | target $I_1$ |
| | exit side 0 position 0 |
| | entry side 0 position 0 |
| export $E_1, \cdots$ | $\vdots$ |
| | |
| contain $E_1, \cdots$ | contain $E_1$ |
| | x 0 y 0 |
| | width 0 height 0 |
| | framecolor green |
| | end $E_1$ |
| | $\vdots$ |
| end $S$ | end $S$ |

Note that, when SIL source is mapped to DL source, exported items are represented in DL by green frame colors. This requires that the contain clause in DL corresponds to an export clause in SIL and that the contain clause in DL has the framecolor green clause. Other usage of colors are: black edges depict imports, black frames depict entities that are not exported, and white boxes depict subsystems. The mapping between a SIL module and its corresponding DL box is similar:

| SIL | DL |
|---|---|
| module $M$ | box $M$ |
| | xorigin 0 yorigin 0 |
| | scale 0 |
| | boxcolor blue |
| | titlecolor white |
| | |
| use $U_1, \cdots$ | edge |
| | edgecolor blue |
| | target $U_1$ |
| | exit side 0 position 0 |
| | entry side 0 position 0 |
| | $\vdots$ |
| end $M$ | end $M$ |

Note that `blue` edges are used to depict the `use` relation, and `blue` boxes are used to depict modules. Having given the definitions of the syntax and semantics of SIL and DL, as well as a mapping for translating between specifications written in either of these languages, we next present an overview of the prototype implementation to support these languages.

# 8 Overview of Prototype Implementation

Our decision to separate the syntax of the languages from their semantics is reflected in the structure of our prototype implementation shown in Figure 3. The three major components of the implementation are:

1. *Editors:* Any text editor can be used to create and modify the textual SIL and DL specifications. For DL specifications, however, users will probably prefer to use our visual editor that directly edits a diagram and saves it as the corresponding textual DL specification.

2. *Translators:* The Star system [9] acts as a translator between SIL and DL. Star has parsers and source code generators for both languages and can translate a specification written in one language into an equivalent specification of the other language. The translations between the SIL and DL specifications are based on the mapping described in Section 7. The main difference between this mapping and the implemented mapping is that Star uses the Sugiyama [17] graph layout algorithm to automatically add positioning information to translations from SIL to DL specifications. For the Star system to perform these translations, the specifications must be syntactically correct. The Star parsers can also translate SIL or DL specifications into a set of Prolog facts that can be used for verifying the semantic constraints. This translation is simple; for example, if $A$ contains $B$ in a SIL or DL specification, the generated Prolog fact is `contain('A', 'B')`.

3. *Theorem Provers:* A Prolog interpreter is used as a theorem prover[7] that verifies the semantic constraints of both SIL and DL specifications. The constraints of both languages are specified using Prolog rules. SIL and DL specifications are translated into a set of Prolog facts by the Star system.

This implementation framework suits our experimental purposes because it comprises easily replaced modular components. We wanted to be able to adjust the syntax and semantics of our languages without changing the tools very much. Hence, we chose to loosely integrate simple tools using the Star system. For example, changing the semantics of the SIL language requires only changes to the respective Prolog rules. Moreover, adjusting the syntax of SIL only implies updating a single module in the Star system.

The modular implementation framework, however, may not be practical in a non–academic setting because of the overhead associated with the translations. A production

[7]In another paper [9], we showed how other interpreters — such as the one in the ConceptBase Knowledge Representation Management System [8] — can be used instead of Prolog.

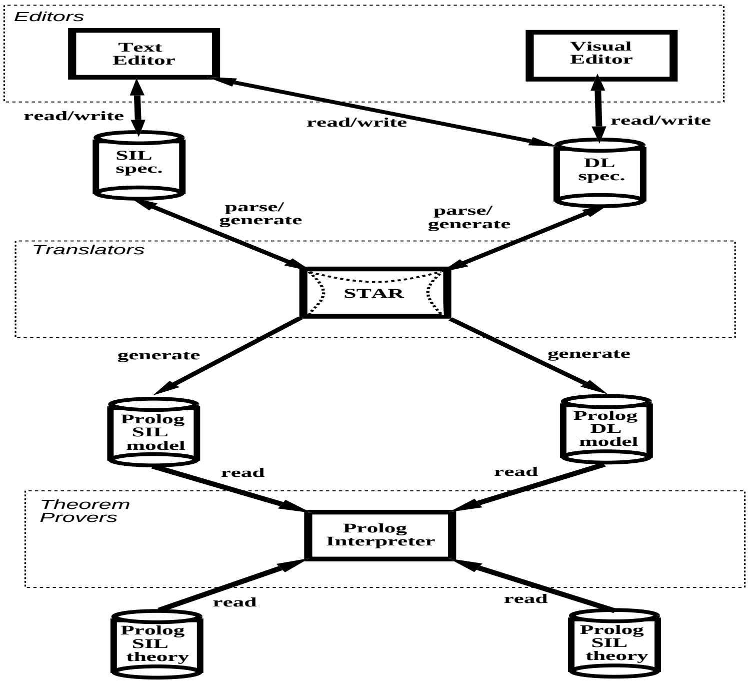


Figure 3: **Implementation Overview:** Boxes represent tools, cylinders represent persistent specifications, and arrows represent the relationships between tools and specifications.

quality version might integrate the components more tightly; it might merge SIL and DL into a single language, and would probably check the semantic constraints directly, rather than using a general interpreter such as Prolog.

# 9 Conclusions and Future Research

In this paper, we described a framework for formally defining languages for specifying and visualizing architectural designs. We explained how this framework was used to define the syntax and semantics of languages for specifying designs (SIL) and drawings (DL). Subsequently, we described the relationship between SIL and DL specifications by defining a syntactic mapping from one to another. Finally, we described our prototype implementation based on this framework.

We hope that this research will lead to languages and tools for architectural design that will be used with the same degree of confidence that conventional programming languages and compilers are used. From a specification point of view, this research shows that specifying architectural designs can be done both easily and unambiguously. From an implementation point of view, it shows that visually displaying and manipulating such designs as well as checking for their syntactic and semantic consistency can be done both mechanically and correctly.

We are currently working on using the framework to define languages for specifying and visualizing more general architectural designs that also accommodate object orientation, reuse libraries, configuration management, and data flow. In addition, we intend to extend the framework to permit algebraic manipulations of the objects in the semantic theory. We also plan to implement tightly integrated tools to support our languages. These tools will eventually be incorporated into the Object–Oriented Turing (OOT) [10] programming environment and used by developers and students at our university.

# References


[1] Booch, G. *Object–Oriented Design with Applications.* Benjamin/Cummings Publishing Company Inc., Redwood City, California, 1991.

[2] Cooprider, L. W. The Representation of Families of Software Systems. Tech. Rep. CMU–CS–79–116, Computer Science Department CMU, April 1979.

[3] DeRemer, F., and Kron, H. H. Programming–in–the–Large Versus Programming–in–the–Small. *IEEE Transactions on Software Engineering 2*, 2 (June 1976), 80–86.

[4] Eigler, F. C. GXF: A Graph Exchange Format. In *Declarative Database Visualization: Recent Papers from the Hy+/GraphLog Project*, A. Mendelzon, Ed. Technical report CSRI–285, University of Toronto, July 1993, pp. 91–107.

[5] Garlan, D., and Shaw, M. Architectures for Software Systems. In *tutorial given at ACM SIGSOFT '93: Symposium on the Foundations of Software Engineering* (Los Angeles, California, December 1993).

[6] Holt, R. C., Penny, D. A., and Mancoridis, S. Multicolour Programming and Metamorphic Programming: Object Oriented Programming–in–the–Large. In *Proceedings of the 1992 IBM CASCON Conference* (November 1992), pp. 43–58.

[7] Jackson, M. *Principles of Program Design.* Academic Press, New York, New York, 1975.

[8] Jarke, M. ConceptBase V3.0 User Manual. Tech. Rep. MIP–9106, Universitat Passau, March 1991.

[9] Mancoridis, S., Holt, R. C., and Godfrey, M. W. A Program Understanding Environment Based on the "Star" Approach to Tool Integration. In *Proceedings of the Twenty–Second ACM Computer Science Conference* (March 1994), pp. 60–65.

[10] Mancoridis, S., Holt, R. C., and Penny, D. A. A "Curriculum–Cycle" Environment for Teaching Programming. In *Proceedings of the Twenty–Fourth ACM SIGCSE Technical Symposium on Computer Science Education* (February 1993), pp. 15–19.

[11] Penny, D. A. *The Software Landscape: A Visual Formalism for Programming–in–the–Large.* PhD thesis, Department of Computer Science, University of Toronto, 1992.

[12] PENNY, D. A., HOLT, R. C., AND GODFREY, M. W. Formal Specification in Metamorphic Programming. In *S. Prehn and W. J. Toetenel (eds.),* VDM '91: Formal Software Development Methods, Proceedings of the 4th International Symposium of VDM Europe (October 1991), Springer–Verlag Lecture Notes in Computer Science no. 551.

[13] PERRY, D. E., AND WOLF, A. L. Foundations for the Study of Software Architectures. *Software Engineering Notes 17*, 4 (October 1992), 40–49.

[14] RYMAN, A., LAMB, D. A., AND JAIN, N. Theories and Models in Software Design. Tech. Rep. TR 74.081, IBM Canada Lab, October 1991.

[15] SCHWANKE, R. W., ALTUCHER, R. Z., AND PLATOFF, M. A. Discovering, Visualizing, and Controlling Software Structure. In *Proceedings of the Fifth International Workshop on Software Specification and Design* (Pittsburgh, Pennsylvania, May 1989), pp. 147–150.

[16] SPIVEY, J. M. *The Z Notation: A Reference Manual (2nd ed.).* Prentice Hall International, 1992.

[17] SUGIYAMA, K., TAGAWA, S., AND TODA, M. Methods for Visual Understanding of Hierarchical System Structures. *IEEE Transactions on Systems, Man, and Cybernetics 11*, 2 (February 1981), 109–125.

[18] TICHY, W. F. Software Development Control Based on System Structure Description. Tech. Rep. CMU–CS–80–120, Computer Science Department CMU, January 1980.

[19] WARE, C., HUI, D., AND FRANCK, G. Visualizing Object Oriented Software in Three Dimensions. In *Proceedings of the 1993 IBM CASCON Conference* (October 1993), pp. 612–620.

[20] YOURDON, E., AND CONSTANTINE, L. L. *Structured Design: Fundamentals of a Discipline of Computer Program and System Design.* Prentice Hall, Englewood Cliffs, New Jersey, 1985.