ISF: A Visual Formalism for Specifying Interconnection Styles for Software Design

Spiros Mancoridis
Department of Mathematics & Computer Science
Drexel University
3141 Chestnut Street,
Philadelphia, PA, USA 19104

e-mail: smancori@mcs.drexel.edu

Abstract

We have developed a framework for specifying high-level software designs. The core of the framework is a very simple visual notation. This notation enables designers to document designs as labelled rectangles and directed edges. In addition to the notation, our framework features a supporting formalism, called ISF (Interconnection Style Formalism). This formalism enables designers to customize the simple design notation by specifying the type of entities, relations, legal configurations of entities and relations, as well as scoping rules of the custom notation.

In this paper we present the formal definition of ISF and use ISF to specify two custom design notations. We also describe how ISF specifications, using deductive database technology, are used to generate supporting tools for these custom notations.

Keywords: *Visual Formalism, Software Design, Deductive Databases.*

1 Introduction

The structure of a large software system typically consists of hundreds, often thousands, of typed components and dependencies. For example, software structures feature procedures that call procedures, classes that inherit from other classes, interfaces that are implemented by modules, and so on.

Being able to understand the structure of a software system is crucial for the successful implementation, testing, and maintenance of that system. However, determining the most important structural aspects of a large system from the rich structural information present in the source code is not easy. Software developers have long appreciated the need for high-level abstractions to provide summaries of the numerous low-level components and dependencies present in the source code.

In practice, software developers create a mental model of what they believe to be the important aspects of a system's software structure. These models are typically represented as informal diagrams. Such diagrams, however, are often incorrect, out of date, and difficult to understand.

Research into the area of software structure aims at improving the ability of software developers to understand and manage the structure of large systems. This research has led to notations for specifying software structure, reverse engineering tools for extracting structural information from source code, tools for visualizing the structure of software, and so on.

Despite the availability of several design notations, most designers still document the structure of their systems using informal diagrams. We believe that a reason for the reluctance of designers to adopt one of the available design notations is the lack of flexibility of these notations.

Software designers create structural abstractions to communicate their designs to others. These abstractions differ depending on the kind of software system being designed. Examples of such abstractions are: server, database, subsystem, filter, pipe, and so on. It is unlikely that a designer will find a notation that will support all of the abstractions necessary to specify a given software system. Without this support, designers are obliged to specify structural abstractions in terms of the abstractions (*e.g.,* objects) supported by a general-purpose design notation.

Our research provides a framework that enables software designers to create custom, yet formal, notations for documenting the structure of software systems. These custom notations can be used to specify high-level design components and dependencies that are not supported by conventional programming languages and design notations.

Our Work

Our framework is based on a simple, yet customizable, visual notation for specifying high-level designs. This notation enables designers to create designs consisting of labelled rectangles and directed edges. Unlike other design notations, our notation does not support a fixed set of entities, relations, and scoping rules. Instead, we have developed a supporting visual formalism, called ISF (Interconnection Style Formalism), that enables designers to specify *Interconnection Styles*. We define an interconnection style to be a description of:

- the types of components present in the design (*e.g.,* module, subsystem),
- the types of dependencies present in the design (*e.g.,* import, export),
- syntactic composition rules for defining the set of all well-formed configurations of components and dependencies,
- the semantics of each well-formed configuration (*e.g.,* exported components are visible to external client components).

Using ISF, designers can customize our simple notation into one of the existing design notations or, instead, choose to customize the simple notation into a new design notation. Independent of the outcome of the customization process, our prototype environment are able to automatically generate tools to support each custom notation.

In this paper we use ISF to specify two interconnection styles. We chose two styles that are variations of the styles commonly found in design notations called Module Interconnection Languages (MILs) [12]. Our intent is to demonstrate the expressiveness of ISF as well as to describe how we use ISF and deductive database technology [15] to generate tools that support custom notations for high-level design. These tools are used to check, among other things, the well-formedness of software designs. We say that a design is *well-formed*, with respect to an ISF specification, when the configuration of the entities and relations that comprise the design do not violate any of the rules of the ISF specification.

Presentation Outline

The rest of this paper is structured as follows: First, we provide a brief survey of research that is related to our work. We then give two examples of software designs that follow specific interconnection styles. For each design, we use ISF to specify the style followed by the design.

The examples of ISF specifications are intended to give readers enough background to be able to follow the formal definition of ISF that follows. The formal definition shows how programs for checking the well-formedness of any style of software design can be generated from the style's ISF specification. We conclude the paper by summarizing the research contributions of this work.

2 State of the Art

Our research is related to three major research areas. Namely, those of deductive databases, visual formalisms, and software architecture.

Deductive Databases

Deductive database systems are based on a declarative language (*i.e.,* Datalog [15]) used to specify rules. A declarative language is used to define what a program wants to achieve rather than how to achieve it.

A deductive database consist of facts and rules. Facts specify data as relational tuples. Rules specify relations that are not actually stored but that can be formed from the facts by applying inferences on the rules of the database.

Deductive databases are related to both relational databases and logic programming languages such as Prolog [2]. The Datalog language is actually closely related to a subset of Prolog.

In our work, we use deductive database technology to support the tools we create for our custom notations for high-level design. Figure 1 shows the architecture of our prototype environment. The environment features visual editors for formulating both software designs (as labelled rectangles and edges) and ISF interconnection styles.

Central to the architecture is the Coral [13] deductive database system. Visual software designs are translated into Datalog facts and interconnection styles are translated into Datalog rules. Using Coral, designers can verify the well-formedness of particular designs. The verification process will detect stylistic violations. Examples of such violations are: "a subsystem that contains itself" and "a subsystem that includes its parent subsystem in its interface". Designers may also perform queries on a particular design. An example of such a query is: "show all subsystems that are accessible from subsystem S".

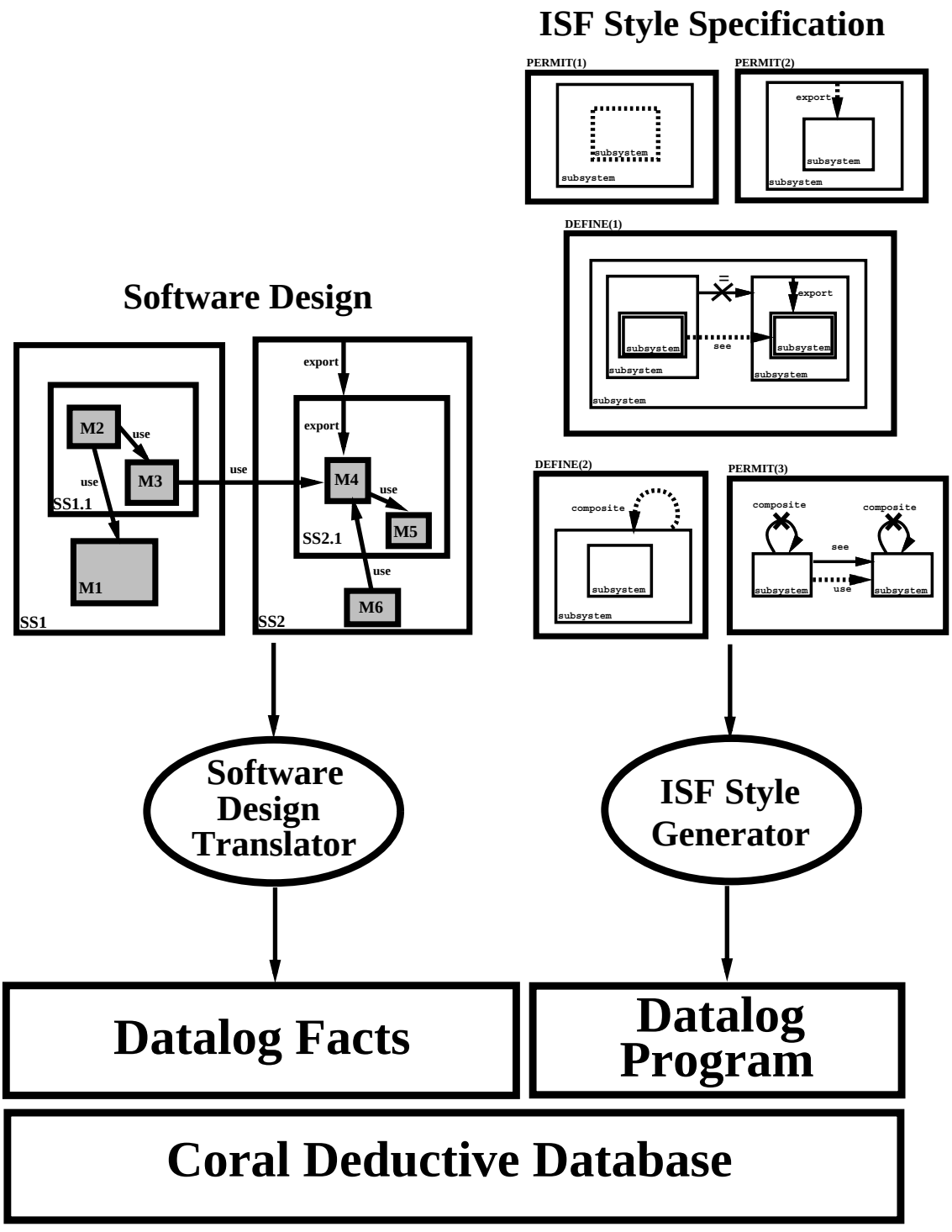


Figure 1. Architecture of Prototype

Visual Formalisms

Visual formalisms are used to model software systems in a formal way. For example, Entity-Relationship (E/R) diagrams [1] and Statecharts [8] are used for modeling the data and dynamic behavior of systems, respectively.

We have developed a visual formalism called ISF that is used for specifying interconnection styles. We chose to develop a visual formalism, as opposed to a textual one, because we wanted to express interconnection styles using symbols that are most commonly associated with software design specifications (*i.e.,* annotated boxes and arrows.)

One part of an ISF specification is used to describe what type of dependencies are permitted among certain types of components. For example, we may want to express that the *export* dependency is only permitted between subsystems that have a parent-child relationship. This aspect of ISF could have been supported by E/R diagrams, which model typed entities and relations.

There are several constraints, however, that cannot be expressed in E/R but are useful when specifying interconnection styles. For example, there is no way to define new relations based on patterns of existing relations using E/R, such as the transitive closure of the *export* dependency. For this reason, ISF supports additional capabilities for defining relations. Such capabilities are common in other visual formalisms such as GraphLog [3].

Software Architecture

"Software architecture is concerned with the principled study of large-grained software components, including their properties, relationships, and patterns of combination." [16]

Architectural Description Languages (ADLs), and their earlier manifestations Module Interconnection Languages (MILs) [12], provide support for specifying software systems in terms of their components and interconnections. Different languages define interconnections in different ways. For example, in MILs [5, 17, 11, 12] connections are mappings from services required by one component to services provided by another component. In ADLs [14, 4] connections define the protocols used to integrate a set of components.

Other work in software architecture goes beyond modeling the designs of specific systems. Garlan and Shaw [7] model architectural patterns (*e.g.,* client-server) for coarse-grained architectures. Gamma et al. [6] model recurring patterns of fine-grained architectures that occur in object-oriented programming (*e.g.,* Model-View Controller). The hope of both groups is that these patterns will codify existing good practices and provide a vocabulary for software architects.

Our work is not tied to any particular notation for describing software designs. It assumes that the notation being used can model designs as typed components and dependencies. Using ISF, designers have the choice of (a) configuring their design language to match an existing design notation or (b) creating a new special-purpose design notation.

Regarding patterns, our work does not attempt to codify common patterns of software structure. Rather, it attempts to codify interconnection styles using ISF. Documenting patterns of interconnection styles can be used to explain the meaning of a variety of high-level components (*i.e.,* subsystems) and dependencies (*i.e.,* export). Such patterns could be useful to the language designers of future MILs and ADLs.

We continue with examples of software designs that follow two distinct interconnection styles. After each example, we use ISF to specify the style employed by the design in that example.

3 Example of a Software Design Following the Export Style

The Export Style described in this paper is a generalization of the export style found in many programming languages (*e.g.,* Eiffel [10]). Specifically, the Export Style facilitates the specification of subsystem interfaces. Subsystem interfaces allow designers to control the interactions between components by using scoping rules. Controlling these interactions is one important way in which the overall complexity of software development and maintenance can be reduced.

Subsystem interfaces are defined using the *export* relation between two software components. In the Export Style, a subsystem may only export modules or subsystems it directly contains. One of the assumptions of this style is that the contents of subsystems are not, by default, visible to external subsystems. Export relations are, therefore, used for exposing otherwise hidden subsystems and modules.

The top of Figure 2 illustrates an example of a software design that follows the Export Style. The bottom of the same figure shows the design expressed as a set of Datalog facts. The software design consists of subsystems (white boxes) and modules (dark boxes). For simplicity, we consider modules to be special kinds of subsystems that do not contain other subsystems. We consider modules to be "atomic" components, even though they contain more finely-grained components such as variables and procedures, which are not modeled explicitly in our example design. Hence, in the Datalog translation (Figure 2) modules are specified as subsystems. Our design features two kinds of directed edges. The thick edges represent export relations, while the thin edges represent use relations between modules. We say that a module M2 uses a module M1 if procedures in M2 call procedures or reference variables and data types in M1.

In the Export Style, modules require permission from their encapsulating subsystems before they can be used by modules in other subsystems. For example, module M6 in subsystem SS2 can use module M4 in subsystem SS2.1 because M4 is exported by its parent subsystem. However, M6 cannot use module M5 because the latter is a hidden (not exported) module.

Similarly, module M3 can access module M4 because the latter is transitively exported by subsystem SS2. By exporting M4, subsystem SS2.1 makes it accessible to modules in the scope of subsystem SS2. In order for M4 to be accessible to modules outside of SS2, the latter must export SS2.1, thus making the exported contents of SS2.1 accessible to the contents of SS1 (and hence to M3).

Note that, in the Export Style, sibling modules (same parent subsystem) are allowed to use each other without any export permissions. For example, module M4 can use its sibling module M5 because they have a common parent subsystem, namely SS2.1. Similarly, modules that are nested in one or more levels of subsystems can access any modules their ancestors can access without requiring any permissions. For example, module M2 can use module M1. Both of the aforementioned scoping rules are generalizations of the block scoping rules found in most modern programming languages.

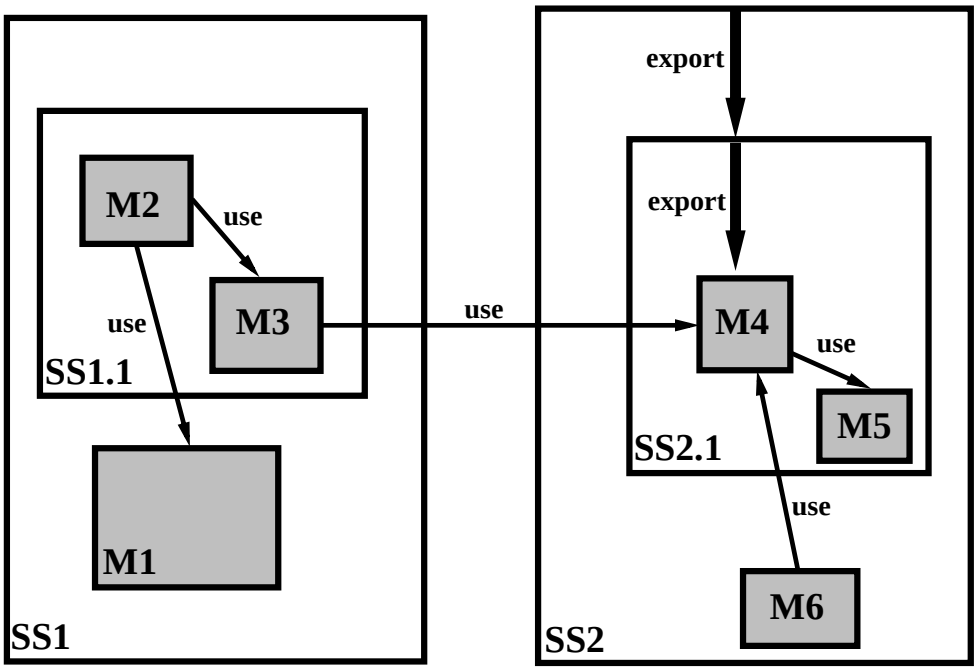
We next present the ISF specification for software designs that follow the Export Style. The semantics of this style is given in both prose and Datalog.

4 The ISF Specification of the Export Style

ISF is a visual formalism that enables designers to specify interconnection styles. Informally, an ISF specification consists of a finite set of logic rules. Each rule is defined by a finite set of entities and relations. Entities are used to represent software components (*e.g.,* modules, subsystems), while relations are used to represent software interconnections (*e.g.,* use, export) and the nesting of software components.

Each ISF rule is represented as a rectangle with a label on its top-left corner. These rectangles

Software Design Diagram



Datalog Facts

% Entity Types
subsystem(’SS1’).
subsystem(’SS1.1’).
subsystem(’M1’).
subsystem(’M2’).
subsystem(’M3’).

% Containment Relations
contain(’SS1’, ’M1’).
contain(’SS1’, ’SS1.1’).
contain(’SS1.1’, ’M2’).
contain(’SS1.1’, ’M3’).

% Module Depenedencies
use(’M2’, ’M1’).
use(’M2’, ’M3’).

% Module Depenedencies
use(’M3’, ’M4’).

% Entity Types
subsystem(’SS2’).
subsystem(’SS2.1’).
subsystem(’M4’).
subsystem(’M5’).
subsystem(’M6’).

% Containment Relations
contain(’SS2’, ’SS2.1’).
contain(’SS2’, ’M6’).
contain(’SS2.1’, ’M4’).
contain(’SS2.1’, ’M5’).

% Export Relations
export(’SS2’, ’SS2.1’).
export(’SS2.1’, ’M4’).

% Module Depenedencies
use(’M4’, ’M5’).
use(’M6’, ’M4’).

Figure 2. Export Style Design

contain other rectangles (entities) and edges (relations). The containment relation is depicted as the nesting of entity rectangles. Each ISF rule contains a single dashed edge or dashed rectangle. The semantics of the dashed edge or dashed rectangle depends on the kind of ISF rule.

There are two kinds of ISF rules: *Permission* rules, labelled PERMIT, determine the syntactically legal (well-formed) relations between the components of a software design. *Definition* rules, labelled DEFINE, are used to define new design relations based on patterns of entities and relations.

Figure 3 shows five visual rules that comprise the ISF specification for the Export Style. Three of these rules are permission rules and two are definition rules. The semantics of these rules in prose and Datalog are as follows:

- **PERMIT(1):**
  - **Informal:** Subsystems are permitted to contain other subsystems. The dashed rectangle represents the permitted containment relation. Note, each permission relation has exactly one permission edge or rectangle associated with it. This dashed edge or dashed rectangle models relations that are permitted to occur under the conditions prescribed by the other relations (solid edges) and entities (solid rectangles) of the rule.
  - **Formal:** wf_contain(PSS, SS) ← subsystem(PSS), subsystem(SS).
- **PERMIT(2):**
  - **Informal:** Subsystems may export the subsystems they directly contain. In this rule, the permission relation is depicted as a dashed edge – remember, only the containment relation is not depicted by an edge.
  - **Formal:** wf_export(PSS, SS) ← subsystem(PSS), subsystem(SS), contain(PSS,SS).
- **PERMIT(3):**
  - **Informal:** Modules (subsystems that are not composite) can use other modules that are in their scope. In other words, a module can use a module that it can see. This rule depends on two relations (*i.e.,* composite, see) that are defined using the two definition rules described next. The crossed relations are negated relations. So, the rule states that subsystems that are not composite (*i.e.,* modules) can use each other if they can see each other.
  - **Formal:** wf_use(SS1,SS2) ← subsystem(SS1), subsystem(SS2), not(composite(SS1)), not(composite(SS2)), see(SS1,SS2).
- **DEFINE(1):**
  - **Informal:** Before we explain this rule we give the reader some background on the ISF notation. Rectangles with double frames represent entities that are transitively contained within an entity. Hence the contained rectangle may be the child, grand child, great-great grand child, and so on, of the container rectangle. Moreover, because the transitivity is reflexive, the contained rectangle may be the container rectangle itself. Similarly, edges with a double arrow head depict reflexive transitive relations. Below is a description of the semantics of the rule presented in two parts:
    1. Reflexive Case: In this case, we assume that the transitive relations (containment and export) are reflexive. Hence, we define the see relation between proper sibling subsystems. This part of the rule enables sibling subsystems to see each other. Note,

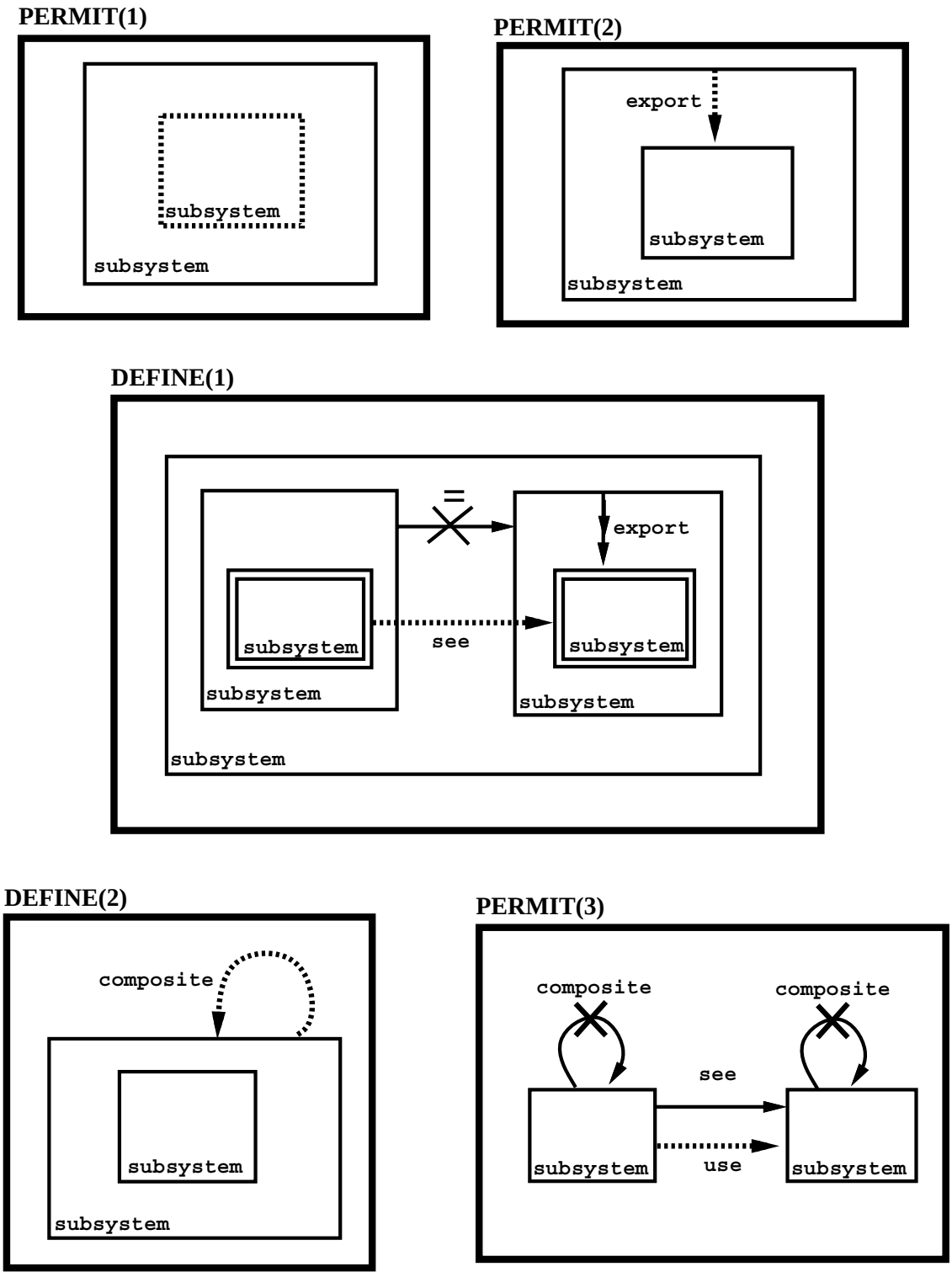


Figure 3. The Export Style

that this rule defines a relation. Unlike permission rules, which specify when design relations are permitted to occur, definition rules actually define new relations.

2. Transitive Case: Here we define the see relation between a subsystem (and its descendants) and the exported descendants of its siblings.

– **Formal:** see(SS1,SS2) ← subsystem(SS1), subsystem(SS2),
subsystem(PSS1), subsystem(PSS2),
not(PSS1=PSS2), subsystem(SS),
contain(SS,PSS1), contain(SS,PSS2),
rtc_contain(PSS1,SS1), rtc_contain(PSS2,SS2),
rtc_export(PSS2,SS2).

where rtc_contain is the reflexive transitive closure of relation contain, which is defined in Datalog as follows:
tc_contain(X,Y) ← contain(X,Y).
tc_contain(X,Y) ← contain(X,Z), tc_contain(Z,Y).
rtc_contain(X,Y) ← X = Y.
rtc_contain(X,Y) ← tc_contain(X,Y).
The definition of rtc_export, which is the reflexive transitive closure of relation export, is similar to that of rtc_contain.

- **DEFINE(2):**

  – **Informal:** This rule defines the composite relation. A subsystem is composite if it contains another subsystem. We need this relation in order to distinguish subsystems from modules.

  – **Formal:** composite(PSS,PSS) ← subsystem(PSS), subsystem(SS), contain(PSS,SS).

- **Well-Formedness Constraint:**

  – **Informal:** The well-formedness constraint is used to guarantee that a design does not have any ill-formed relations. It is an implicit constraint, as it does not actually appear in the ISF specification.

  A design is well-formed, with respect to the Export Style, if all contain, export, and use relations in the design are well-formed.

  – **Formal:**
  ill_formed() ← contain(X,Y), not(wf_contain(X,Y)).
  ill_formed() ← export(X,Y), not(wf_export(X,Y)).
  ill_formed() ← use(X,Y), not(wf_use(X,Y)).
  well_formed_design() ← not(ill_formed()).

Next, we give an example of another style, called the Tube Style.

5 Example of a Software Design Following the Tube Style

The basic idea of a *tube* relation is as follows: If a subsystem SS1 is connected to a subsystem SS2 by a tube relation, the contents of SS1 can access the contents of SS2. Subsystems that are not connected by a tube cannot access each other.

In the Tube Style [9], subsystem interfaces are not specified explicitly via some relation (*e.g.,* export). In this style, subsystems actually have two interfaces. One for those subsystems that are connected to them by a tube and one for those that are not. In the former case, the subsystem interface is the set of all of its contained subsystems. In the latter case, the subsystem interface is the empty set.

The top of Figure 4 illustrates an example of a software design that follows the Tube Style. This is the same example as was used in Figure 2, except that the export and use relations have been removed and tube relations have been added. Note that the use relations between modules have been replaced by tube relations. The bottom of the Figure 4 shows the design expressed as a set of Datalog facts.

The tube between subsystem SS1.1 and module M1 serves as a permission that allows module M2 to access M1. Note that the direction of the tube relation is important. For example, M1 cannot access M2 because there is no tube relation from M1 to SS1.1.

In order for M3 to be able to access M4 there must be a tube between the two modules. In order for any of the contents of SS1.1 to access any of the contents of SS2.1, a tube between these subsystem is required. Hence the tube between SS1.1 and SS2.1. Similarly, in order for any of the contents of SS1 to access any of the contents of SS2, a tube between these subsystem is required. Hence the tube between SS1 and SS2.

As was the case in the Export Style, in the Tube Style, sibling modules are allowed to access each other without requiring special permissions. Note that the tube between M6 and SS2.1 enables M6 to access both M4 and M5. In the Tube Style its is not possible to hide M5 without hiding M4 and vice versa. Such finely-grained hiding is possible with the Export Style, however.

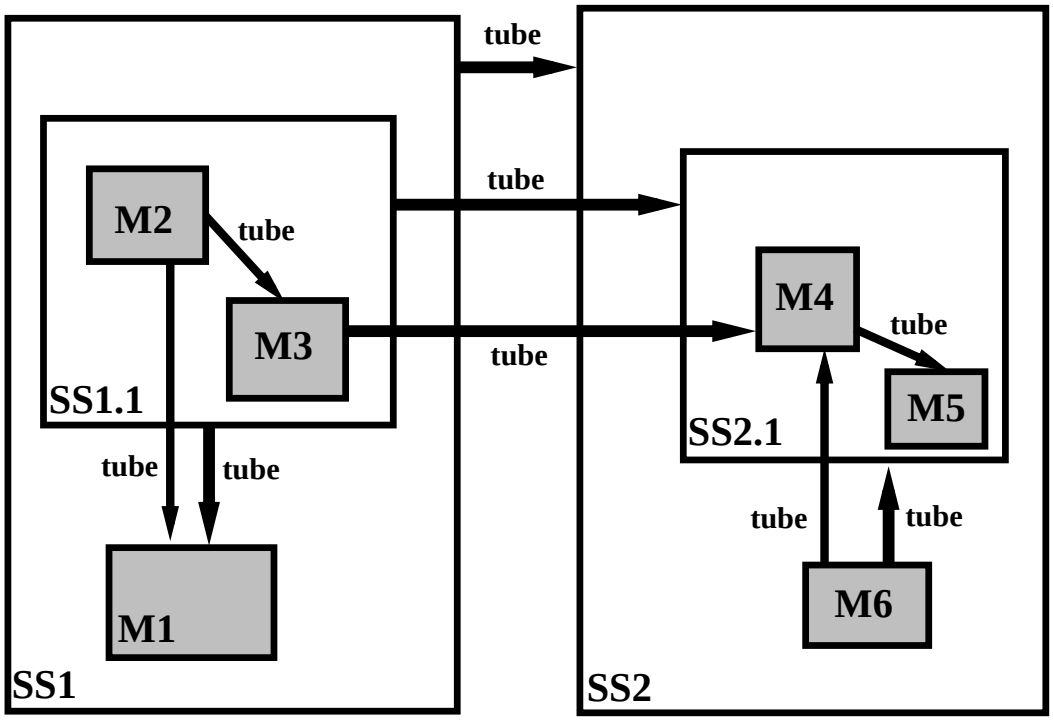
We next present the ISF specification for the Tube Style.

6 The ISF Specification of the Tube Style

Figure 5 is the ISF specification of the Tube Style. The meaning, given informally, of the rules that comprise the specification is given below.

- **PERMIT(1):** Subsystems are permitted to contain other subsystems.
- **PERMIT(2):** Subsystems that are proper siblings may be connected via a tube. Two subsystems are proper siblings if they are distinct and contained within the same parent subsystem.
- **PERMIT(3):** A subsystem SS1 that connects to another subsystem SS2 via a tube may also connect to any subsystem contained in SS2. This permission rule, along with the next permission rule, allows tubes to be specified between subsystems on different levels of the subsystem containment hierarchy.
- **PERMIT(4):** A subsystem may connect to any subsystem that its parent is connected to.
- **PERMIT(5):** Subsystems can connect to each other via a tube if the parents of these subsystems are connected via a tube.

Software Design Diagram



Datalog Facts

% Entity Types
subsystem(’SS1’).
subsystem(’SS1.1’).
subsystem(’M1’).
subsystem(’M2’).
subsystem(’M3’).

% Containment Relations
contain(’SS1’, ’M1’).
contain(’SS1’, ’SS1.1’).
contain(’SS1.1’, ’M2’).
contain(’SS1.1’, ’M3’).

% Tube Relations
tube(’M2’, ’M1’).
tube(’M2’, ’M3’).
tube(’SS1.1’, ’M1’).

% Tube Relations
tube(’SS1’, ’SS2’).
tube(’SS1.1’, ’SS2.1’).
tube(’M3’, ’M4’).

% Entity Types
subsystem(’SS2’).
subsystem(’SS2.1’).
subsystem(’M4’).
subsystem(’M5’).
subsystem(’M6’).

% Containment Relations
contain(’SS2’, ’SS2.1’).
contain(’SS2’, ’M6’).
contain(’SS2.1’, ’M4’).
contain(’SS2.1’, ’M5’).

% Tube Relations
tube(’M4’, ’M5’).
tube(’M6’, ’M4’).
tube(’M6’, ’SS2.1’).

Figure 4. Tube Style Design

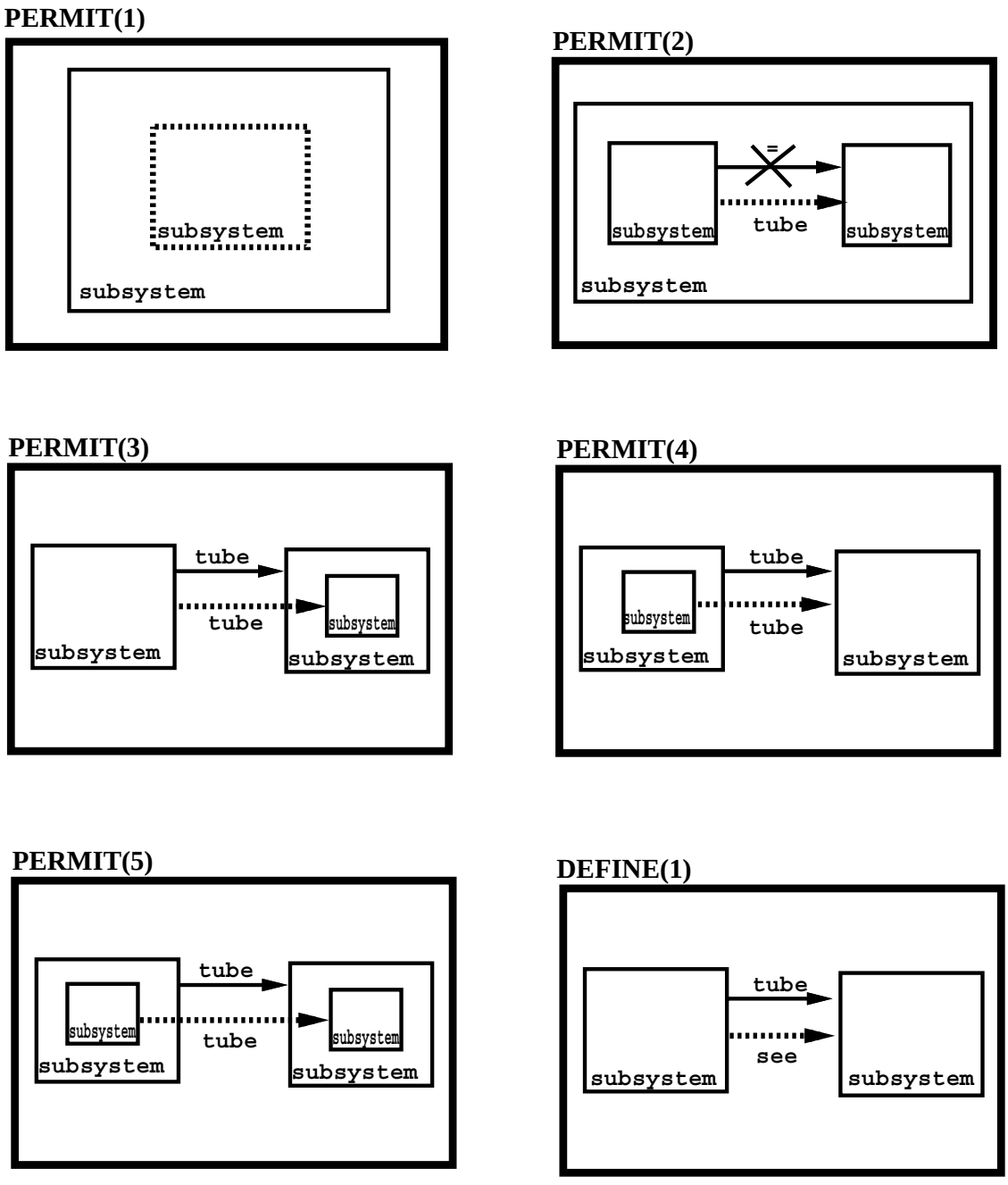


Figure 5. The Tube Style

- **DEFINE(1):** Subsystems can see each other if they are connected by a tube.
- **Well-Formedness Constraint:** A design is well-formed with respect to the Tube Style if all tube and contain relations in the design are well-formed.

Having defined the semantics of two ISF styles, we next describe the formal definition of ISF. The formal definition is important as it shows how Datalog programs for checking the well-formedness of any style of software design can be generated directly from its ISF specification.

7. Formal Definition of ISF

The syntax of ISF is described by presenting the visual symbols of the notation. The visual symbols represent ISF entities, relations, and scoping rules.

The semantics of ISF is described informally using prose and formally using Datalog. The prose description is intended for those interested in a general understanding of ISF. The Datalog description is intended for those interested in a more in-depth understanding of ISF and how ISF rules are translated into Datalog programs. These programs can be processed by a deductive database system, such as Coral, to verify the well-formedness of software designs.

The semantics of the ISF notation is presented in a bottom-up fashion. We start by defining the semantics of ISF entities and relations. We then define the semantics of ISF rules in terms of the semantics of entities and relations.

7.1. ISF Entities

ISF entities are depicted as labelled rectangles. These rectangles should not be confused with ISF permission and definition rule rectangles, which are described later on. Each ISF entity represents a set of typed design components. There are three types of ISF entities, as shown in Figure 6: simple (solid frame), reflexiveTransitive (double frame), and permission (dashed frame) entities. The last two entities must partake in a containment relation, described later. The set of all ISF entities is defined as the following Cartesian product:

$$Entities = \text{id:ID} \times \text{label:Label} \times \text{kind:}\{\text{simple, reflexiveTransitive, permission}\}$$

Note that we specify the Cartesian product together with a naming convention. We use an identifier followed by the colon (:) symbol for each product operand. This allows identifiers to be used in order to access the values for a specific member of the Cartesian product by using the dot (.) symbol. For example, if $e \in Entities$, then $e.label$ is the label of the entity.

Identifiers (ID) and labels (Label) are strings of letters, digits, dashes, and underscores. These strings must, however, begin with a letter.

Let $Ent \subseteq Entities$ be the set of entities in an ISF specification. Each entity in Ent has a unique identifier:

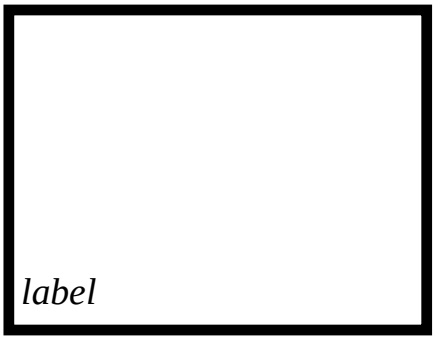
$$\forall\ e_1, e_2 \in Ent, (e_1.\text{id} = e_2.\text{id}) \Leftrightarrow (e_1 = e_2)$$

Since entity identifiers are unique, we can define a function $ent : ID \rightarrow Ent$, that returns the entity corresponding to an identifier:

$$(ent(\text{id}) = e) \Leftrightarrow (e.\text{id} = \text{id})$$

The semantics of an ISF entity $ent \in Ent$, with a label $ent.label$ and a unique identifier $ent.id$, is:

- **Informal:** The set of all entities in a software design of type $ent.label$.



(simple)

(reflexiveTransitive)

(permission)

Figure 6. ISF Entities

- **Formal:** *ent.label*(*ent.id*), where *ent.id* is a free variable whose identifier is unique to that entity.

The semantics of an entity labelled subsystem (see Figure 3) is:

- **Informal:** The set of all subsystems in a software design.
- **Formal:** subsystem(SS), where SS is a free variable.

The entity label, subsystem, represents a relation; its unique identifier, SS, is used as the identifier of the free variable. The unique identifiers of entities are not shown in ISF rules.

The next section describes ISF relations.

7.2. ISF Relations

ISF relations are used to model design relations such as contain, import, export, inherit, and so on. Each relation has a label, a unique identifier for the source ISF entity *(sid)* and a unique identifier for the destination ISF entity *(did)*. (Keep in mind that we will refer to *sid* and *did* throughout this section.) There are two categories of ISF relations. The first category includes containment relations, which are depicted as nested ISF entities. The second category includes edge relations, which are depicted as labelled directed edges between two ISF entities. In either case, the two entities of a relation may be identical if the relation is reflexive.

The set of ISF relations is defined as the following Cartesian product:

$$\begin{aligned} \mathit{Relations} = {} & \text{label:EdgeLabel} \times \text{sid:}\mathit{ID} \times \text{did:}\mathit{ID} \times \\ & \text{kind:}\{\textsf{contain}, \textsf{simple}, \textsf{negatedSimple}, \textsf{reflexiveTransitive}, \\ & \qquad \textsf{negatedReflexiveTransitive}, \textsf{permissionDefinition}\} \end{aligned}$$

where *EdgeLabel* is defined as a Label or an equals character "=".

We continue with descriptions of the containment and edge relations.

Containment Relations

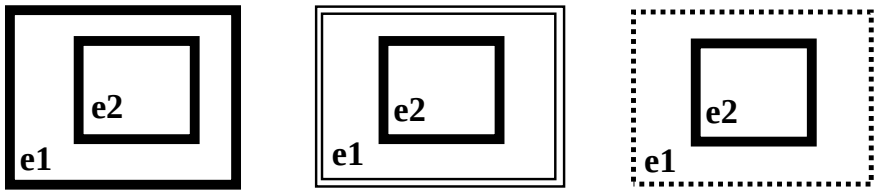
Following the convention of many visual design notations, such as Harel's Statecharts [8], nested entities are used to denote the containment relation.

Let $Ent \subseteq Entities$ be the set of entities of an ISF specification. The semantics of a containment relation between entity $ent_1 \in Ent$, with label e_1 and unique identifier id_1, and entity $ent_2 \in Ent$, with label e_2 and unique identifier id_2, depends on the kind of the contained entity ent_2. Since there are three kinds of entities, there are three semantics that can be given to a containment relation (see Figure 7). The containment of permission entities (dashed) is a special relation whose semantics is given later on when ISF permission rules are presented. The semantics of the other two containment relations are:

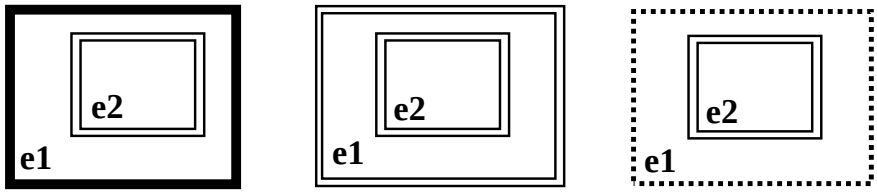
1. **Entity ent_1 contains a simple entity ent_2:**
   - **Informal:** All pairs of entities, where the type of the first entity is e_1, the type of the second entity is e_2, and the first entity contains the second entity.
   - **Formal:** contain(id_1,id_2), e_1(id_1), e_2(id_2)

   The second permission rule of Figure 3 shows two simple entities both labelled subsystem, where one entity is nested inside the other entity. The meaning of this example is:

1. Entity contains simple entity



2. Entity contains transitive entity



3. Entity contains permission entity

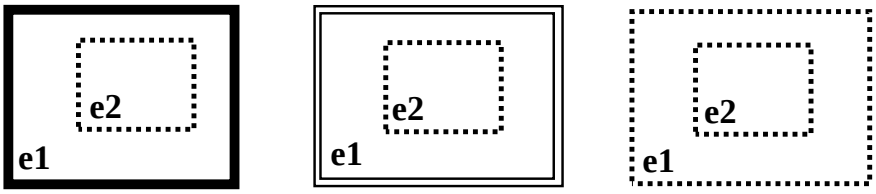


Figure 7. ISF Containment Relations

- **Informal:** All pairs of subsystems such that one subsystem contains the other subsystem.
- **Formal:** contain(PSS,SS), subsystem(PSS), subsystem(SS)

2. **Entity ent_1 contains reflexive transitive entity ent_2:**

   - **Informal:** All entities of type e_1, if e_2 is the same type as e_1 (reflexive case), and all pairs of entities, where the type of the first entity is e_1, the type of the second entity is e_2, and the first entity directly or indirectly contains the second entity (transitive case).
   - **Formal:** rtc_contain(id_1,id_2), e_1(id_1), e_2(id_2)
     where rtc_contain is the reflexive transitive closure of relation contain.

   The first definition rule of Figure 3 shows a reflexive transitive entity nested in a simple entity. Both entities are labelled subsystem. The meaning of this example is:

   - **Informal:** All subsystems (reflexive case), and all pairs of subsystems such that one subsystem directly or indirectly contains the other subsystem (transitive case).
   - **Formal:** rtc_contain(PSS,SS), subsystem(PSS), subsystem(SS)

This concludes the formal definition of containment relations. Next, we describe edge relations, which comprise the other category of ISF relations.

Edge Relations

We mentioned that the containment relation is depicted as two nested entities. All other ISF relations, such as those in Figure 8, are depicted as labelled directed edges between two entities. The entities in a relation may be of any kind (simple, reflexiveTransitive, permission).

Let $Ent \subseteq Entities$ be the set of entities and $Rel \subseteq Relations$ be the set of relations of an ISF specification. The semantics of an ISF relation $rel \in Rel$, with label r, between an entity $ent_1 \in Ent$, with a label e_1 and a unique identifier id_1, and an entity $ent_2 \in Ent$, with a label e_2 and a unique identifier id_2, depends on the kind of the relation rel. There are five kinds of edge relations. The permissionDefinition edges (dashed) are special relations whose semantics is given later on when ISF rules are presented. The semantics of the other four kinds of edge relations are:

1. **Simple relation rel between entity ent_1 and entity ent_2:**

   - **Informal:** All pairs of entities, where the type of the first entity is e_1, the type of the second entity is e_2, and the first entity is related by a relation of type r to the second entity.
   - **Formal:** r(id_1,id_2), e_1(id_1), e_2(id_2)

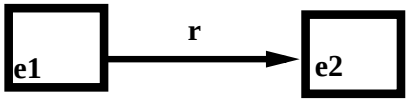
   Example (1) in Figure 9 shows a simple relation, labelled use, between two simple entities, both labelled subsystem. The meaning of this example is:

   - **Informal:** All pairs of subsystems such that one subsystem directly uses the other subsystem.
   - **Formal:** use(SS1,SS2), subsystem(SS1), subsystem(SS2)

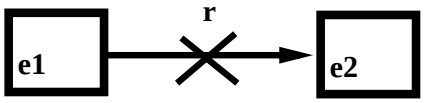
2. **Negated simple relation rel between entity ent_1 and entity ent_2:**

   - **Informal:** All pairs of entities, where the type of the first entity is e_1, the type of the second entity is e_2, and the first entity is not related by a relation of type r to the second entity.

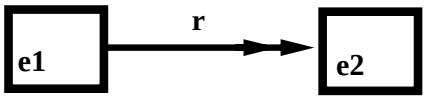
1. Simple Relation



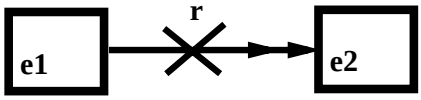
2. Negated Simple Relation



3. Reflexive Transitive Relation



4. Negated Reflexive Transitive Relation



5, 6. Permission or Definition Relation

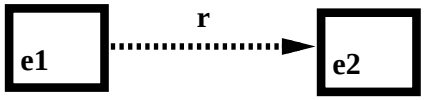


Figure 8. ISF Edge Relations

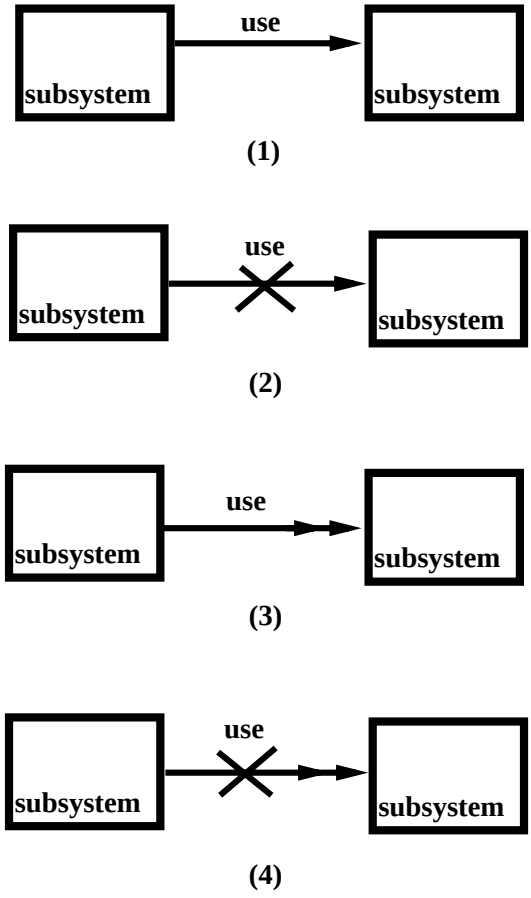


Figure 9. Examples of ISF Edge Relations

- **Formal:** not(r(id_1,id_2)), e_1(id_1), e_2(id_2)

Example (2) in Figure 9 shows a negated simple relation, labelled use, between two simple entities, both labelled subsystem. The meaning of this example is:

- **Informal:** All pairs of subsystems such that one subsystem does not directly use the other subsystem.
- **Formal:** not(use(SS1,SS2)), subsystem(SS1), subsystem(SS2)

3. **Reflexive transitive relation** *rel* **between entity** ent_1 **and entity** ent_2**:**

- **Informal:** All entities of type e_1, if e_2 is the same type as e_1 (reflexive case), and all pairs of entities, where the type of the first entity is e_1, the type of the second entity is e_2, and the first entity is directly or indirectly related by a relation of type r to the second entity (transitive case).
- **Formal:** rtc_r(id_1,id_2), e_1(id_1), e_2(id_2).
  where rtc_r is the reflexive transitive closure of relation r.

Example (3) in Figure 9 shows a reflexive transitive relation, labelled use, between two simple entities, both labelled subsystem. The meaning of this example is:

- **Informal:** All subsystems (reflexive case), and all pairs of subsystems such that one subsystem directly or indirectly uses the other subsystem (transitive case).
- **Formal:** rtc_use(SS1,SS2), subsystem(SS1), subsystem(SS2).
  where rtc_use is the reflexive transitive closure of relation use.

4. **Negated reflexive transitive relation** *rel* **between entity** ent_1 **and entity** ent_2**:**

- **Informal:** All pairs of entities, where the type of the first entity is e_1, the type of the second entity is e_2, and the first entity is not directly or indirectly related by a relation of type r to the second entity. Note that the negated reflexive transitive relation is the same as the negated transitive relation.
- **Formal:** not(rtc_r(id_1,id_2)),e_1(id_1),e_2(id_2)
  where rtc_r is the reflexive transitive closure of relation r.

Example (4) in Figure 9 shows a negated reflexive transitive relation, labelled use, between two simple entities, both labelled subsystem. The meaning of this example is:

- **Informal:** All pairs of subsystems such that one subsystem does not directly or indirectly use the other subsystem.
- **Formal:** not(rtc_use(SS1,SS2)), subsystem(SS1), subsystem(SS2).

Having defined ISF entities and relations, we next define ISF rules, which consist of such entities and relations.

7.3. ISF Rules

An ISF specification consists of a finite set of rules. Each rule is depicted as a rectangle containing entities and relations. We encountered examples of the rectangles representing the two kinds of rules in Figure 3. An ISF specification S is formally defined as follows:

$$S = (Ent,\ Rel, DEFINE,\ PERMIT)$$

Where $Ent \subseteq Entities$ is the set of entities, $Rel \subseteq Relations$ is the set of relations, $DEFINE$ is the set of definition rules, and $PERMIT$ is the set of permission rules.

We proceed with the formal definition of each kind of ISF rule.

Definition Rules

An ISF *definition rule* consists of a DEFINE rectangle that is populated by relations between simple and reflexiveTransitive entities (*i.e.,* no permission entities are allowed in definition rules). For each definition rule there is a single dashed edge, called a permissionDefinition edge, whose label cannot be the equal symbol "=" (*i.e.,* you are not be able to re-define the equality relation). The permissionDefinition edge defines a new relation based on the pattern of typed entities and relations prescribed by the contents of the definition rule rectangle.

The set of definition rules in an ISF specification consists of a set of tuples (E, R, d) that is described as follows:

$$\begin{aligned} DEFINE = \{&(E, R, d) \mid \\ &(E \subseteq \{\text{e} \mid (\text{e} \in Ent) \wedge ((\text{e.kind}=\textsf{simple}) \vee (\text{e.kind}=\textsf{reflexiveTransitive}))\}) \wedge \\ &(R \subseteq \{\text{r} \mid (\text{r} \in Rel) \wedge (\text{r.kind} \neq \textsf{permissionDefinition})\}) \wedge \\ &(d \in \{\text{r} \mid (\text{r} \in Rel) \wedge (\text{r.kind}=\textsf{permissionDefinition}) \wedge (\text{r.label} \neq \text{"="})\}) \\ \} \end{aligned}$$

Each tuple represents a distinct definition rule of an ISF specification. For each definition rule in an ISF specification, E is the set of all entities that belong to the definition rule; R is the set of all (non dashed) relations that belong to the the definition rule; and p is the defined relation (dashed and one per definition rule) of the definition rule.

Each dashed edge represents a relation being defined. The dashed edge has a source and destination entity associated with it. Let srcLabel = ent(d.sid).label and destLabel = ent(d.did).label be the labels of the source and destination entities of a definition rule. The semantics of this rule in Datalog is:

$$\textsf{d.label(d.sid, d.did)} \leftarrow \textsf{srcLabel(d.sid)},\ \textsf{destLabel(d.did)},\ \mu_1, \mu_2, \ldots, \mu_{|R|}.$$

Where μ_i is the meaning (in Datalog) of each ISF relation $r_i \in R$ in the definition rule (as discussed previously in Sections 7.1 and 7.2).

For an example of the semantics of a definition rule see DEFINE(1) in Section 4.

Permission Rules

An ISF *permission rule* consists of a PERMIT rectangle that is populated by relations between entities. For each permission rule there is either a single dashed edge, a permissionDefinition edge, or a single contain relation involving a dashed permission entity. In either case, the permission rule defines a pattern of typed entities and relations that determines whether a dashed relation is permitted to exist.

There should be at least one permission rule for every relation in the design language. For example, if the only relations in the design language are use and export, there should be at least one permission rule for the use relation and one permission rule for the export relation in the ISF specification. To avoid inconsistencies in ISF specifications, definition rules are constrained so that they cannot define relations that are defined using permission rules.

The set of permission rules in an ISF specification consists of a set of tuples (E, R, p) that is described as follows:

$$
\begin{aligned}
PERMIT = \{(E, R, p) \mid\ & \\
& (E \subseteq Ent) \wedge \\
& (R \subseteq \{\mathrm{r} \mid (\mathrm{r}{\in}Rel) \wedge ((\mathrm{r.kind} \neq \textsf{permissionDefinition}) \vee \\
& \qquad\qquad (\mathrm{r.kind}{=}\textsf{contain} \Rightarrow ent(\mathrm{r.did}).\mathrm{kind}{\neq}\textsf{permission}))\}) \wedge \\
& (p \in \{\mathrm{r} \mid (\mathrm{r}{\in}Rel) \wedge ((\mathrm{r.kind} = \textsf{permissionDefinition}) \vee \\
& \qquad\qquad (\mathrm{r.kind}{=}\textsf{contain} \Rightarrow ent(\mathrm{r.did}).\mathrm{kind}{=}\textsf{permission}))\}) \\
\}
\end{aligned}
$$

Each tuple represents a distinct permission rule of the same ISF specification. For each permission rule in an ISF specification, E is the set of all entities that belong to the permission rule; R is the set of all (non dashed) relations that belong to the permission rule; and p is the permission relation (dashed and one per permission rule) of the permission rule.

Each dashed edge or dashed rectangle represents a permitted relation. This arrow has a source and destination entity associated with it. Let srcLabel = ent(p.sid).label and destLabel = ent(p.did).label be the labels of the source and destination entities of a permission rule. The semantics of this rule in Datalog is:

$$\textsf{wf\_p.label(p.sid, p.did)} \leftarrow \textsf{srcLabel(p.sid), destLabel(p.did)}, \mu_1, \mu_2, \ldots, \mu_{|R|}.$$

Where μ_i is the meaning (in Datalog) of each ISF relation $r_i \in R$ in the permission rule (as discussed previously in Sections 7.1 and 7.2).

For an example of a permission rule see, PERMIT(1), in Section 4.

The well-formedness constraint for an ISF specification states that a software design is well-formed, with respect to an ISF specification, if all of the relations of the design are well-formed.

For all permision relations p_i in an ISF specification, let $rel_i = \mathrm{p}_i$.label. The semantics of the well-formedness constraint in Datalog is:

$\textsf{illFormed()} \leftarrow \textsf{rel}_i\textsf{(X,Y)}, \textsf{not(wf\_rel}_i\textsf{(X,Y))}.$
$\textsf{well\_formed\_design()} \leftarrow \textsf{not(illFormed())}.$

We conclude with a summary of the research contributions of this work.

8 Conclusions

In this paper we introduced the ISF visual formalism. We showed how ISF can be used to specify two interconnection styles. By describing the formal definition of ISF, we showed how supporting tools can be automatically generated from ISF specifications.

To summarize, our work makes two significant research contributions:

1. The development of a visual formalism for specifying interconnection styles. These styles can be used to develop custom notations for software design. We believe that no single set of entities, relations, and rules is sufficient for all kinds of software systems and, hence, that a formalism, such as ISF, is beneficial.

2. A formal description of how ISF specifications can be used to generate Datalog code, which can be executed on a deductive database system. This code is used to support well-formedness checking and querying capabilities for a variety of interconnection styles.

References

[1] P. Chen. The Entity–Relationship Model—Toward a Unified View of Data. *ACM Transactions on Database Systems*, pages 9–36, March 1976.

[2] W. F. Clocksin and C. S. Mellish. *Programming in Prolog*. Springer Verlag (3nd ed.), 1987.

[3] M. P. Consens and A. O. Mendelzon. GraphLog: A Visual Formalism for Real Life Recursion. In *Proceedings of the 9th ACM SIGACT–SIGMOD Symposium on Principles of Database Systems*, pages 404–416, 1990.

[4] C. Dellarocas. A Coordination Perspective on Software System Design. In *Proceedings of the 9th International Conference on Software Engineering and Knowledge Engineering*, pages 318–325, June 1997.

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

[6] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series, Reading, Massachusetts, 1995.

[7] D. Garlan and M. Shaw. *In Advances in Software Engineering and Knowledge Engineering*, volume 1. World Scientific Publishing Company, 1993.

[8] D. Harel. On Visual Formalisms. *Communications of the ACM*, 31(5):514–530, May 1988.

[9] S. Mancoridis and R. C. Holt. Recovering the Structure of Software Systems Using Tube Graph Interconnection Clustering. In *Proceedings of the 1996 International Conference on Software Maintenance*, November 1996.

[10] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall International, Englewood Cliffs, New Jersey, 1988.

[11] H. Ossher. A Case Study in Structure Specification: A Grid Description of Scribe. *IEEE Transactions on Software Engineering*, 15(11), November 1989.

[12] R. Prieto-Diaz and J. M. Neighbors. Module Interconnection Languages. *The Journal of Systems and Software*, 6:307–334, 1986.

[13] R. Ramakrishnan, D. Srivastava, and S. Sudarshan. CORAL: Control, Relations, and Logic. In *Proceedings of the International Conference on Very Large Data Bases*, pages 238–250, 1992.

[14] M. Shaw, R. DeLine, D. V. Klien, T. L. Ross, D. M. Young, and G. Zalesnik. Abstractions for Software Architectures and Tools to Support Them. *IEEE Transactions on Software Engineering*, 21, April 1995.

[15] J. D. Ullman. *Principles of Database and Knowledge-Base Systems (Volume 1)*. Computer Science Press, New York, New York, 1988.

[16] A. L. Wolf. Welcome to ISAW-2. In *Proceedings of the Second International Software Architecture Workshop*, October 1996.

[17] A. L. Wolf, L. A. Clarke, and J. C. Wileden. A Model of Visibility Control. *IEEE Transactions on Software Engineering*, 14(4):512–520, April 1988.