Towards a Formal Framework for Object Oriented Architecture [i,ii]

Amnon H. Eden
Department of Computer Science
Concordia University
Montreal, Canada
eden@acm.org

Yoram Hirshfeld
School of Mathematics
Tel-Aviv University
Tel Aviv, Israel
yoram@math.tau.ac.il

Abstract: Architectural representations of software systems are designed to provide a concise roadmap to the elements in the system's composition and the interplay thereof. Supporting Formalisms strive to deliver a succinct expression for architectural specifications. Independently, progress was made in the understanding of O-O architecture through the introduction of patterns of design and architecture. However, there is no agreement which formalism should be used to specify software architecture and architectural patterns, nor there is an agreement which are the elementary units thereof.

This article presents an underlying framework for the formal specification of O-O architecture: We observe building blocks in O-O architectures; reformulate informal concepts in patterns of O-O design and architecture; and define a universal, logic language for the representation of this knowledge. Finally, we use our conceptual toolkit to compare and evaluate proposed formalisms.

Keywords: Software architecture, object oriented programming, design patterns, tool support

1. Introduction

Architectural specifications provide software with "*a unifying or coherent form or structure*" [1]. Used both prescriptively (i.e., placing constraints on expected implementations) and descriptively (i.e., as a form of documentation), architectural specifications determine the gross organisation of software systems and aid in their design and comprehension.

Formal manifestations of architectural constraints are a natural and effective method to establish unambiguous and verifiable representation of design decisions. Various formalisms were proposed for the specification of software architectures, each of which derives from a marked formal language. For example, Allen and Garlan extend CSP [2] with *Wright* [3]; Dean and Cordy [4] use typed, directed multigraphs; and Abowd, Allen and Garlan [5] chose Z [6] as the underlying theory. Other formalisms rely on Statecharts [7] and Petri Nets [8].

O-O software systems, like software fashioned after any other paradigm, must have well-defined architectural representation that models their structure and behaviour. It is generally accepted [9] that clean and simple specifications are a prerequisite for the comprehensibility and maintainability of software systems. Clearly, coherent architectural specifications warrant the recognition of constructs such elementary building blocks and repeating abstractions, such as *inheritance hierarchies* and *function families* (we expand below on these constructs) in the case of O-O systems.

Unlike software constructed after other paradigms, however, object-oriented (O-O) programs comprise of constructs and relations that are idiosyncratic to mechanisms that are inherent to the programming paradigm, such as *inheritance* and *dynamic binding* [10]. Consequently, O-O systems deviate considerably in their models of structure and behaviour from other kinds of systems.

Nonetheless, research in software architecture and in *Architecture Description Languages* (ADLs) [11] largely ignored the O-O idiosyncrasies. In particular, only few works offered a sufficiently expressive yet compact set of elementary building blocks, or could capture the microarchitectures observed in

---

[i] Available online from
`http://www.cs.concordia.ca/~faculty/eden/bibliography.html#tffooa`

[ii] Submitted: *Formal Methods Europe 2001*, 12-16 March 2001, Humboldt-Universitaet zu Berlin, Germany

design and *architecture* patterns. As a result of this oversight, any attempt to capture the architecture of O-O systems using contemporary architectural specification languages overlook key regularities in their organisation.

We conclude that elements and regularities in the structure of O-O architectures are yet to be defined. More specifically, we expect the definition of a small set of primitives ("building blocks") that will be proved sufficient to express the architecture of O-O software. We also expect the observation of higher level abstractions, as well as composite regularities ("motifs") in O-O architecture. Finally, we seek a formalism that will account for these primitives, as well as the composite abstractions that were observed. Ideally, this formalism will provide a succinct expression for most common specifications and will be supported by a simple reasoning mechanism for proving various relations between expressions in this formalism.

1.1 Patterns

Better than in any other form of documentation, the *patterns* literature, and in particular *design patterns* papers, capture essential abstractions in O-O software architecture. Each pattern addresses a separate abstraction of deign or architecture, documented in a fixed format, and designated by a name which carries its intent. These abstractions are used for communicating on common problems and solutions, and incorporate specifications of useful and common abstractions in O-O programs.

Our study of the abstractions manifested by patterns of O-O design and architecture has lead us to believe that the patterns literature provides the best insight into the most common regularities in O-O architecture. However, despite being widely hailed for capturing effective abstractions, we observe that the patterns literature has yet to mature as a structured engineering domain. More specifically, we observed two problems in this context:

1. ***Ambiguity***. The informal, imprecise, and often vague description of implementation solutions puzzles pattern users, raises questions about similarity and differences between patterns, to the extent that even pattern veterans squabble over the "true meaning" of vague specifications (e.g., [12, 13]).

   The open questions that appear to occupy the patterns community most are the following two:

   - *Validation*: Whether a particular implementation *conforms* to the specification of a given pattern (redefined in Definition V);
   - *Refinement* (also *specialisation*): Whether one pattern refines (is a special case of) another (redefined in Definition IX).

2. ***Unstructured knowledge***. With the growth in the number of *patterns* published, the accumulation of information is evolving into an unorganised, unstructured mass, without any effective means of ordering it. The solution to this problem, also known as the *indexing problem*, is some mechanism for hierarchical classification and navigation among the numerous patterns.

Clearly, these two problems result from the lack of explicit and accurate representations. Similarly to formalisms addressing O-O architecture, we expect formal specifications of patterns of O-O deign and architecture, to be used to produce unambiguous specifications, to enable reasoning about the relationships between patterns (e.g., *refinement*), to offer well defined interpretation for activities performed on patterns (e.g., *recognition*), and to allow structuring the rapidly growing body of patterns by a manageable index.

We conclude that the contribution of a prospective formalism for O-O architecture can be tested with respect to the specification of design patterns. As demonstrated in section 4, several proponents of various prospective formalisms have illustrated their contribution by giving the specification of design patterns, most commonly of those listed in the GoF catalogue [14]. We use the widely influential patterns literature as a common reference for our observations, and in comparing different formalizations.

A Note on Terminology. Our use of the term *pattern* diverges from the way it is being used within the patterns community. Inspired by the work of Christopher Alexander [15, 16], a "pattern" is recognised as a prescription for solving a category of problems in specific manner and intended for the wide dissemination of specialised knowledge. Nonetheless, many before us have used this term with relation to the architectural abstraction manifested in the solution part of the *pattern* paper (also known as *microarchitecture*, *lattice* [17], and *leitmotif* [25]), in particular in the case of *design patterns*. Since we concern ourselves with software architectures, the term *pattern* is used only with respect to the architectural abstraction manifested in the solution, designated hereby as the *leitmotif* of patterns of O-O design and architecture.

1.2 Intent

Having observed the difficulties in the patterns literature and the expected benefits of an appropriate formalism, this article introduces the following contributions: We –

- Observe the building blocks of O-O architecture of two types:
  1. A small set of rudimentary elements or *primitives* that is sufficient to express a reasonably large set of architectures; and
  2. The regularities ("motifs"), or composite abstractions, that are most common in the composition of O-O architecture;
- Provide a formal framework that serves as a frame of reference for the discussion in the essential concepts O-O architecture;
- Provide precise definitions for intuitive terms used with reference to O-O patterns;
- Demonstrate how pattern specifications in various formalisms translate to our framework;
- Demonstrate the utility of our approach through formally establishing informally observed relations between patterns;
- Provide criteria for assessing the properties of a prospective pattern specification language (e.g., *expressiveness* and *completeness*).

A Methodological Note. We employ mathematics as means for precise specification, reasoning, and for making predictions. Yet, as in other scientific disciplines where the subject of the statements made (verbal descriptions of patterns in our case) is informal, there is no rigorous way to prove the correctness of some of the statements we make. These claims can only be treated as approximations for some "natural phenomena" [18]. As in standard scientific practice [18], our aim is to provide evidence which support our claims and that are widely applicable, as well as test the implications of the proposed formalisms using the evidence supplied.

2. The Building Blocks of O-O Architecture

In this section we observe the abstractions we consider as the fundamental motifs in O-O architecture. Examples are predominantly taken from patterns of design and architecture described in the *Design Patterns* catalogue [14], also referred to as the Gang of Four (GoF) catalogue.

Rudiment A: ***Participants***. The elementary agents in the implementation of O-O software, or the *participants* partaking in the pattern's leitmotif, are *classes, methods*, and *objects*.

Traditionally, Methods and data members (also *features* and *attributes*) are represented as class elements. We chose, however, to model *classes* as atomic primitives, while class "members" are treated as separate entities that are related to their containing class through the *Defined-In* and *Reference* relations. We expand on this design decision in section 3.2.

Rudiment B: ***Collaborations***. We identify a small set of "ground" ("primitive") *relations* that are sufficient to account for the types of most of the collaborations among participants in the GoF patterns. These relations are listed in Table 1. Their sufficiency is illustrated in more detail in [19] and, to a lesser degree, in [20].

From this point on, a *ground relation* is a collaboration between an n-tuple of participants of the appropriate types and number.

In the following specifications, $\mathbb{F}$ stands for the domain of *methods* (also *function members* or *routines*), $\mathbb{C}$ for the domain of *classes*. Where convenient, the specification of some of these relations takes the form of a function.

Abstract : $\mathbb{F} \cup \mathbb{C}$
Indicates whether a class or a method are abstract (in Eiffel: *deferred*, in C++: *pure virtual function*)

Creation : $\mathbb{F} \times \mathbb{C}$
Indicates that the body of the method contains an expression that results in the creation of a new instance of the class. Such expressions include, for example:

```
new id_numbers;              (Java)
list<int> id_numbers[2];     (C++)
if (string("A") == s)        (C++)
!!id_numbers.make(2);        (Eiffel)
```

Defined-in : $\mathbb{F} \times \mathbb{C}$
Indicates that a method is defined in a certain class

Forwarding : $\mathbb{F} \times \mathbb{F}$
Forwarding(*f*,*g*) means that *f* invokes *g*, where the actual arguments in the invocation expression are the formal arguments of *f*. For example (using C++):

```
void TCPConnection::ActiveOpen(int time) { _state->ActiveOpen(time); }
```

Invocation : $\mathbb{F} \times \mathbb{F}$
Indicates that within the body of the first method there is an explicit invocation expression of the second method (also: “may call”). *Note that we ignore the control flow of the first method*

Inheritance : $\mathbb{C} \times \mathbb{C}$
Indicates that the first class inherits from the second

Reference [*-to-many*]: $\mathbb{C} \times \mathbb{C}$
Indicates that one class defines a member whose type is a reference to a single (multiple) instance(s) of the second class

Return-Type : $\mathbb{F} \times \mathbb{C}$
Indicates that the execution of the method results in an object with the specified class as its (static) type

Same-Signature : $\mathbb{F} \times \mathbb{F}$
Signifies that the two functions have the same name and formal arguments. This condition may vary with the programming language according to the criteria for overriding (as required in Definition I.)

Table 1: Ground relations in our model

Rudiment C: ***Uniform Sets***. Arrays consisting of either classes or methods, whose elements uniformly conform to a specific role, are omnipresent in the description of patterns. To demonstrate the uniformity of these sets, here are a few examples: The set of *factory-methods* of the *ABSTRACT FACTORY*; the set of *product* classes in the same pattern; the set of *operation-implementation* defined in the *BRIDGE*; the set of *concrete-strategy* classes that inherit from the abstract *strategy* class of the *STRATEGY* pattern; the set of *decorator* classes of the *DECORATOR* pattern; and so forth.

From this point on, by a [*ground*] *set* we refer to a *uniform set* of participants.

Rudiment D: ***Higher Order Sets***. Sets of sets are also abundant in patterns. For instance, the set of sets of *product* classes, and the set of sets of *factory methods* in the *ABSTRACT FACTORY*; the set of sets of *visit*(*element-i*) methods of the *VISITOR* pattern; and so forth.

The last two examples are also characterised as *clans* (Rudiment E).

Rudiment E: ***Clans and Tribes***. Method sets of special interest are ones that enable dynamic binding. This implies that all the methods in that set share the same *signature*, and that each method is defined in a different class (most often these classes are ordered in a certain inheritance hierarchy). Formally:

> Definition I: *Clan*
>
> A set of methods F is *a clan in the set of classes* C iff all $f \in F$ share the same signature ([i]), and that each such f is defined in a different class in C.

Clans are ubiquitous and occur at least once in each pattern. For example, the set of *visit*(*element*$_i$) methods of the *VISITOR* pattern; the set of *create*(*product*$_i$) of the *ABSTRACT FACTORY*; the set of the *update* methods of the *OBSERVER* pattern; the set of *ConcreteAlgorithms* of the *STRATEGY* pattern; and so forth.

Having mentioned sets of higher order, we note a set of *clans* of special interest: A set of clans T such that every clan $F \in T$ share the same set of classes C. Formally:

> Definition II: *Tribe*
>
> A set of sets of functions $\{F_i\}$ is *a tribe in a set of classes C* iff every set $F \in \{F_i\}$ is a *clan* in C.

Although *tribe* has not been observed as a first-class abstraction, almost every pattern and OOP program incorporate one or more *tribes*. For example, the set of *create*(*product-i*) clans of the *ABSTRACT FACTORY* jointly makes a tribe in the *Factory* classes, and so does the set of *visit*(*element-i*) clans in the *Visitors* set of the *VISITOR* pattern. In the *BRIDGE* pattern, for example, for every possible implementation, a clan of *Operation-Imp* method is redefined throughout the *Implementors* hierarchy, and jointly makes a *tribe* in *Implementors*.

Rudiment F: ***Bijections***. Bijective mappings of *relations* characterise collaborations between *uniform sets* of various orders. For instance, the description of the *FACTORY METHOD* [14] states that "*each function of Factory-Methods creates an object of a class in the Products hierarchy*" [*ibid.*, p. 108] which implies that the *Creation* relation between the sets *Factory-Methods* and *Products* is a bijection. In the *consequences* section of the same pattern it is stated that its implementation "*connects parallel class hierarchies*" [*ibid.*, p. 108]. Clearly, this implies a bijective relation between the sets of classes *ConcreteCreators* and *ConcreteProducts*. Similarly, every clan of *visit*(*element-i*) methods in the *Visit* tribe is expected to "visit" a class *element-i* (i.e., operate on instances thereof) of the *Elements* hierarchy; thus "visiting" is a bijective relation between the *Elements* set of classes and the *Visit* tribe. Other bijections exist in every pattern of the GoF [14] catalogue, as demonstrated in [20].

Note also that a *F is a clan in C* can be defined as using a bijective *Defined-In* relation between F and C.

Rudiment G: ***Inheritance Hierarchies***. Inheritance hierarchies are abundant within O-O software systems. A particular kind of hierarchy, however, prevails in the specifications of patterns: A single inheritance hierarchy, consisting of an abstract "root" class and a uniform set of concrete classes, each of which inherits (possibly indirectly) from the root (see Formula 4). Observe also the occurrences of *sets of hierarchies*, e.g., the set of *product* hierarchies in the *ABSTRACT FACTORY*.

Every pattern in the GoF [14] contains one or more instances of *hierarchy* exactly as it was defined here. For example: The set of *concrete-observers* which inherit from the abstract *observer* (*OBSERVER* pattern); the set of *concrete-products*, all of which inherit from the abstract *product* class (*FACTORY METHOD* pattern); and the set of *concrete-command* classes which inherit from the abstract *Command* class (*COMMAND* pattern). From this point on, by the term *hierarchy* we refer only to the abstraction demonstrated here.

---

[i] A weaker requirement involving *conformance* would have been sufficient in most circumstances. For the sake of simplicity, however, we use a stronger requirement to avoid a discussion in the differences between programming languages.

3. Setting the Scene

In this section, we form a foundation for a formal discourse in static O-O architecture and patterns. Using this context, we define terms used with respect to patterns.

3.1 Static Vs. Dynamic Models

We focus our discussion in the static model of software systems for various reasons. The static components and their correlations are the ingredients of the system's structure, and form the key to the comprehension of its behaviour. Moreover, static properties are far easier to specify, prove or refute, and most important, to comprehend. if the primary reason for the software crisis was (and still is) lack of *abstraction*, as implied by the OOP literature, then there is more than enough room to improve the structural picture of software systems.

In particular, O-O programs are distinguished by supporting *inheritance*, an entirely static abstraction mechanism. This is true not only for statically, strongly typed languages, but also for dynamically typed OOP languages. The topology of an O-O system is predominantly characterised through the graph of its inheritance relations. We conclude that OOP architecture is characterised best in its static form. In particular, the organisation of *classes*, *methods*, and their relations (Rudiment B), is the "stage" on which the dynamic manifestation of the program takes place, where classes materialise as objects, and methods take the form of messages.

Finally, by studying leitmotifs of the predominant patterns that are categorised as "behavioural", one may realise that much of their characterisation can be expressed through structural, static relations. The examples provided in section 4 illustrate this observation.

3.2 The Conceptual Framework

Programmers are aware of the conceptual model that accompanies every O-O program: A universe inhabited by classes, operations, attributes, and relations among them. Some of these relations are explicit in the programming language, such as *inheritance*; others, such as *invocation* (Rudiment B), might have more implicit incarnations (e.g., an invocation to constructor within the declaration expression.)

Static Models. Regardless of the implementation language, static and dynamic conceptual models exist throughout the phases in the system's lifecycle, and are at the heart of its architectural reference model.

Design patterns deal with structural and behavioural features of the universe. The "rudiments" we observed lead to the conclusion that, predominantly, design patterns characterise *static* attributes of this universe. Even patterns with dynamic flavour ("behavioural" patterns) actually describe a configuration in the static model.

There is a lot to be gained by making this conceptual "stage" into an explicit (logic) *structure*. Many vague claims, such as "this pattern is just a special case of another", become straightforward facts in this structure.

We propose to render this structure explicit exactly like any other mathematical structure, such as algebraic band, boolean algebra, or geometry of points and lines. The crux of our contribution is, to start with, in recognising the *participants*, their essential *collaborations*, and the suitable way of manifesting them. This practice is not entirely unlike distinguishing the participants in geometric discourse – *points* and *lines*, and the primary relations – *point x is in line y*. Finally, we consider the "staging". In the geometric context, for instance, we ask whether the relation *line x is parallel to line y* is primary, or whether it is deduced from other relations. Similarly we ask whether *lines* are atomic entities, or perhaps should be represented as sets of *points*, namely, as more elaborated constructs.

The framework we propose incorporates *classes* and *methods* as its atomic entities, similarly to *points* and *lines*, and the relations *class d inherits from class b*, *method f is defined in class c*, and *method f creates instances of class c*, as the ground relations amongst them. In conclusion, each program ([i]) is represented by a *model*, as demonstrated in Example 1.

---

[i] For the purpose of this article, a program in a programming language $\mathscr{L}$ is any piece of text that is considered well formed by the rules of $\mathscr{L}$.

```
class Decorator {
   void Draw() {}
}

class BorderDecorator extends Decorator {
   void Draw() {
      Decorator d[2]; ...
   }
   int  BorderWidth;
}
```

Atoms (“*participants*”):

`Decorator`, `BorderDecorator`, `int` of type *class*

`Draw1` and `Draw2` of type *method*

Relations (“*collaborations*”):

Defined-in(`Decorator::Draw`,`Decorator`)

Defined-in(`BorderDecorato::Draw`,`BorderDecorator`)

Inherit(`BorderDecorator`,`Decorator`)

Reference (`BorderDecorator`,`int`)

Create(`BorderDecorator::Draw`,`Decorator`)

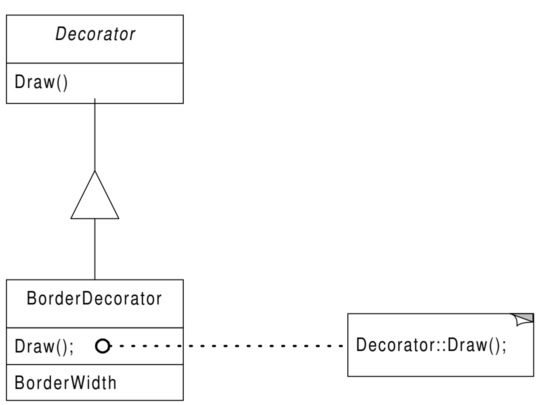
Example 1: Program vs. its Model

By *object language* we refer to the OOP language that was used to write “programs” in their original sense. Thus, the object language of the source code in Example 1 is C++.

Various object notations (e.g., UML [29] , OMT [21], Booch [22], OPEN [23]) represent class methods and attributes (also *features*, *data members*, *instance variables*) as elements of the respective class. However, To accomplish a cleaner and more uniform representation of architectural elements, we represent both *methods* and *classes* as first class primitives. The association between a method f and the class c in which it is defined is represented using a relation: *Defined-In*(f,c). Similarly, if class c contains one attribute of type a, it is represented by the relation *Reference*(c,a). By this we achieve a simpler picture (an essential step towards the abstraction level of *architecture*), where a program is represented as a *model* which consists of primitive entities and ground relations, just as logic *structure*.

To illustrate how our representation scheme compares with object notations, consider the Structure diagram of the *DECORATOR* pattern [14] quoted in Extract 1. Note, in particular, that information regarding the relation between methods cannot be conveyed in first class terms; instead it is specified using informal “notes”. In contrast, consider the formal representation of the same information as given in Figure 1. Note that what is represented as *attributes* in OMT (as well as UML and all other object notations) we represent by means of relations between classed.

Note that although the OMT notation is used in this particular book, class diagrams in this notation differ from those made in other object notations merely by the choice of geometrical shapes that represent classes.



Extract 1: OMT diagram of a segment of the DECORATOR *[14] pattern*

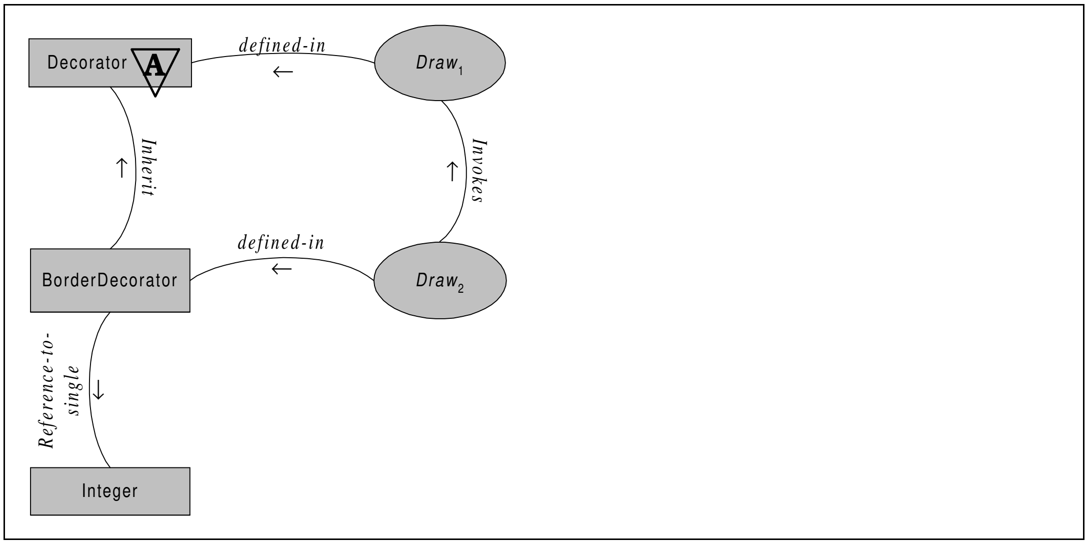


Figure 1: Reformulating Extract 1 in terms of entities and relations

Having illustrated the relation between *programs*, *class diagrams*, and the conceptual framework we propose, we move to define the concept of a "model of a given program" in formal terms:

> Definition III : *Model of a Program*
>
> Given an *object language* $\mathscr{L}$, a set of *participant types* $\mathbb{T}=\{\mathbb{F},\mathbb{C}\}$, a set of relations $\mathfrak{R}=\{$*Inherit*, *Abstract*, …$\}$, and a function $\mathcal{M}$ that maps programs in $\mathscr{L}$ to $\mathbb{T}$ and $\mathfrak{R}$, we define $\mathcal{M}(p)$, or *the model of a program p according to* $\mathcal{M}$, as the pair $\langle participants_{\mathcal{M}}(p), collaborations_{\mathcal{M}}(p)\rangle$, where $participants_{\mathcal{M}}(p)$ is a collection of atomic entities, each representing a construct in p and is of a type of $\mathbb{T}$, and $collaborations_{\mathcal{M}}(p)$ are the relations in $\mathfrak{R}$ that are defined amongst $participants_{\mathcal{M}}(p)$.

Definition III gives rise to a well defined universe of *models*, which we designate as $\mathbb{P}$. Note that since models are abstractions of programs, each model represents an infinite number of programs.

A set of relations and of participant types that is different than those mentioned in Definition III may be considered for similar purposes.

Each *program* in our discussion is handled through its representation as the respective model. Thus we may discard programs in their original sense from our discussion and use the term *program* with reference to its abstract representation as a model rather than in its original sense. Because we need not refer directly to programs in their original representation, we do not see a difficulty in using this alias.

We use higher order logic formulae, specifying static constructs, uniform sets, and relations as defined above as a *canonical form*, in contrast with notations which incorporate specialised terms and shorthand symbols. In the following discussion we will demonstrate how expressions in other specification languages, including many behavioural specifications, map to our canonical form.

3.3 Specification Languages

Mathematical logic provides numerous formalisms for deliberating algebraic constructs such as the one proposed in Definition III, from first order predicate calculus to higher order languages, which afford the representation of higher order sets, functions, and relations. Different publications present different views on the suitable formalism for the discussion in patterns (e.g., higher order monadic logic [24] in LePUS [25], and constraint diagrams [26]). Some of these languages are discussed in section 4.

Specification languages differ in the following aspects:

1. Their underlying formalism;
2. Their constructs represented, including both primitives and the composite abstractions
3. The properties that are featured as *dynamic*, i.e., as properties of runtime entities, vs. the properties that are formulated by static entities.

Seeking a common denominator for these variations, we concluded that:

1. The examples each publication quotes can be expressed in higher order logic;
2. The atomic and composite constructs need account for the building blocks listed in section 2;
3. Many descriptions of properties that appear to be "dynamic," such as *behavioural patterns* [14], can in fact be formulated through expressions over static entities. For example, "invokes" (section 2) is a static relation between methods, as it designates the fact that the body of one method contains a statement whose execution will result in the invocation of another method, (AKA *may-invoke*). Such relations distinguish static specifications, where only in dynamic specifications one may specify constraints such as: *if event e occurs than method f must be invoked*.

To illustrate statements 1 and 3 above, we analyse in section 4 examples from various languages and transform them into expressions in higher order logic over static entities and relations.

Instance of a Pattern. Definition III gave rise to a well defined universe of *models*, and thus to many languages which offer *formulae* specifying properties of models. A formula would naturally contain *variables*, possibly *typed variables*, as well as *set variables*, such that certain models will satisfy it and others will not.

If leitmotifs are expressed as formulae in a language that arises from this picture, we can use this uniform framework to formally rephrase concepts in the practice of software patterns as well defined terms, such as *instance of a pattern*. Definition IV establishes the distinction between the pattern (leitmotif), the elements that partake in its implementation (its *instance*), and the program that contains the pattern's instance. This distinction should resolve the common confusion between the pattern's *leitmotif* and a specific *instance* thereof:

> Definition IV: *Instance (also witness, implementation) of a Pattern*
>
> Let φ designate a formula which stands for the pattern π, $\mathfrak{M}$ designate a model. Let $a_1,\ldots a_n$ designate an n-tuple of ground entities in $\mathfrak{M}$ that satisfy φ.
>
> The model $\mathfrak{M}'$ in $\mathfrak{M}$ is said to be *an instance of* π iff it consists of:
>
> – $a_1,\ldots a_n$
>
> – The relations that are defined among $a_1,\ldots a_n$ in $\mathfrak{M}$ and that are described in φ

Example 2 below illustrates this definition.

4. Sample Specifications

In this section we illustrate the usability of our framework as a foundation for the specification of O-O software architecture. By means of examples that represent patterns of O-O design and architecture, taken from various formalisms, we compare existing architectural specification languages, and demonstrate how these expressions are mapped to our framework.

Various formalisms give rise to different logics. The example in Formula 1 is in a language that is similar to first order predicate calculus:

$$Creates(f, c)$$ **Formula 1**

Example 2 illustrates an instance of the trivial pattern manifested by Formula 1.

> Example 2: *Instance of Formula 1*
>
> Model $\mathfrak{M}$ of Example 1 is *an instance of* the pattern represented in Formula 1. To show this, consider the assignment of `BorderDecorator::Draw` to f and of `Decorator` to c. If we apply this assignment we get:
>
> *Creates*(`BorderDecorator::Draw,Decorator`)
>
> which is true in $\mathfrak{M}$.

In the following example, let *root* range of entities of type *class* and let *Leaves* range over sets of entities of type *class*:

$$\forall\, leaf \in Leaves : Inherit^{+}(leaf, root)$$ **Formula 2**

where $\mathcal{R}^{+}$ designates the transitive closure of the relation $\mathcal{R}$. Formula 2 indicates that all the classes in *Leaves* inherit, possibly indirectly, from the class for which *root* stands for. To render explicit the information on type and order of the free variables, one may choose to qualify the free variables in Formula 2 as demonstrated in Formula 3:

$$\exists\, root \in \mathbb{C}, Leaves \in 2^{\mathbb{C}} : (Formula\ 2)$$ **Formula 3**

where $2^{\mathbb{C}}$ stands for the domain of uniform sets of ground classes.

Different languages define various abbreviations ("syntactic sugars") according to the abstractions in their focus. Consider for example the definition given for the shorthand notation for *inheritance hierarchy* (Rudiment G) in LePUS [25]:

$$\exists!\, root \in h : Abstract(root) \wedge \forall\, leaf \in h : \ (leaf \neq root \rightarrow Inheritance^{+}(leaf, root))$$ **Formula 4**

where *h* stands for a set of classes.

Different languages represent variables in different ways, and allow for different kinds of relations to be defined. Consider, for example, the specification of the *Mediator* class in the *MEDIATOR* pattern [14] in DisCo [27], a specification language based on *temporal logic of actions* [28]:

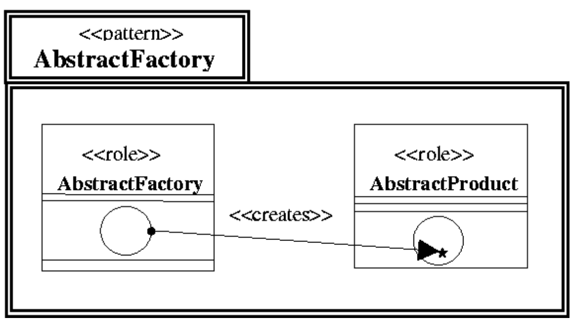
class $Mediator = \{\, \{\, Boxes \,\} : Mbox \,\}$ **Extract 2** (DisCo)

Converting this specification into our canonical, static form, we get:

$$Reference\text{-}to\text{-}Many(Mediator, MBox)$$ **Formula 5**

Where *Mediator* and *Mbox* are variables that range over the domain of entities of type *class*.

Compare Extract 2 with the "role model" diagram ([i]) of the *ABSTRACT FACTORY* pattern, given in the "constraint diagrams" visual specification language [26] (Extract 3). The circle inside the *AbstractProduct* "role" in Extract 2 represents the set of instances of classes that inherit from *AbstractProduct*. The <<creates>> relation indicates that for each such object there exists a method which is defined in some class that inherits from *AbstractFactory*, that generates each instance thereof.



Extract 3: "Role Diagram" [26] of the ABSTRACT FACTORY *pattern*

If this specification appears to be dynamic, Formula 6 demonstrates how the same information can be conveyed in our static, canonical form:

$$\begin{aligned}&Defined\text{-}in(f, ConcreteFactory) \wedge \\ &Creates(f, ConcreteProduct) \wedge \\ &Inherits^{+}(ConcreteFactory, AbstractFactory) \wedge \\ &Inherits^{+}(ConcreteProduct, AbstractProduct)\end{aligned}$$ **Formula 6**

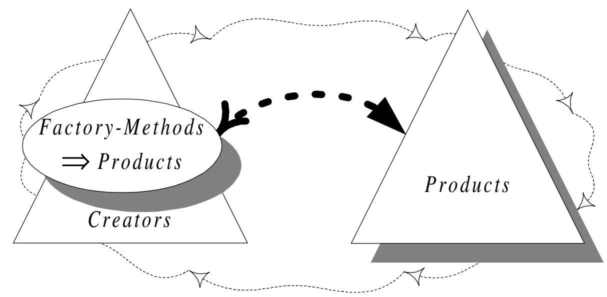
Where *f* ranges over the domain of methods, and the remaining variables range over the domain of classes. The free variables *ConcreteProduct* and *ConcreteFactory* were introduced to account for the entities which partake in the *creates* relation.

---

[i] The "role model" diagram of Extract 2 is only a partial specification of the *ABSTRACT FACTORY* that is further refined by a number of other diagrams.

DisCo [26] is defined as extension to UML [29], an object notation whose diagrams depict O-O programs. In contrast, LePUS [25] is defined as a language for the specification of O-O software architecture. LePUS offers a visual equivalent to its formulae, which are a compact subset of higher order monadic logic.

Consider for example the specification of the *ABSTRACT FACTORY* in LePUS. Extract 4 gives the visual version, while its textual equivalent is given in Formula 7, where the variable *Factory-Methods* varies over sets of sets of classes (3rd order sets), *Creators* stands for a *hierarchy* (whose definition appears is given in Formula 4), and *Products* stands for a set of hierarchies.



Extract 4: LePUS diagram of the *ABSTRACT FACTORY*

Formula 7

$$\begin{array}{l} \mathit{tribe}(\mathit{Factory\text{-}Methods}, \mathit{Creators}) \wedge \\ \mathit{Production}^{\leftrightarrow}(\mathit{Factory\text{-}Methods}, \mathit{Products}) \wedge \\ \mathit{Return\text{-}Type}^{\leftrightarrow}(\mathit{Factory\text{-}Methods}, \mathit{Products}) \wedge \\ \mathit{Commute}\ (\mathit{factory\text{-}method}, \mathit{Products}) \end{array}$$

Note that LePUS incorporates syntactic sugar for bijections ([i]), so that $\mathcal{R}^{\leftrightarrow}(X, Y)$ indicates that $\mathcal{R}$ is a bijective relation between the sets *X*, *Y*.

4.1 Related Work

In addition to the results analysed above, the following research directions were reported in literature:

Design patterns' formalisms. Formal Languages for the specification of design patterns are described in several publications. Helm, Holland and Gangopadhyay [30] defined "Contracts", an extension to 1st order logic with representations for function calls, assignments, and an ordering relation between them. The "behavioural compositions" described do not address structural relations but only run time ("behavioural") characterisations.

Tool Support. Florijn, Meijers, and van Winsen [31] propose a tool that supports the application of design patterns. They use the "Fragments Model" for the representation of patterns, a graph with labelled arcs, whose nodes stand for the *participants* in the pattern's leitmotif, and its arcs describe the roles of the connecting nodes. The *patterns' wizard* [32] supports the *application* of patterns, represented as metaprogramming algorithms which describe the sequence of steps in the application of the respective pattern. Other tools are described by Pal [33], by Budinsky, Finnie, Vlissides, and Yu [34], and by Quintessoft Inc. [35], all of which are reviewed in [19].

Ground Relations. Other works have observed "elementary" relations between entities that can be used to document O-O architecture. Keller et. al [36] list *Call Actions* and *Create Actions* (corresponding to *Invocation* and *Creation* relations of Table 1 respectively) among the information their reverse engineering environment maintains for the representation of O-O programs, as well as information on the relations between classes (e.g., *Inheritance*, *Reference*). Chiba [37] describes a metaprogramming environment for manipulating C++ programs whose features are classified under *Function Invocation* and *Object Creation*, among other features.

---

[i] The *Commute* relation indicates that the two bijections from *Factory-Methods* to *Products* overlap.

5. Applications

In this section we use the framework laid in previous sections to formally redefine concepts of the patterns' jargon and to establish relations between abstractions.

5.1 Activities Related to Patterns

We observe a number of activities related to design patterns carried out during the process of developing software. We expect to benefit from a formal definition of a particular activity if it is to be supported by a CASE tool, possibly automating each of these activities in part or as a whole.

Validation test whether a particular piece of code is indeed an instance of a given pattern, a question that occupies many pattern users. By representing a program as a model, and a pattern as a formula, *validation* question can be resolved conclusively in a straightforward way:

> Definition V: *Validation*
>
> A program p is said to *validate* the pattern represented by the formula φ iff the model of p is an instance of φ.

The term *application* refers to the process that manifests what is widely perceived as the original purpose of patterns, that is, to provide software designers with a recipe for implementing a solution to a category of design problems.

To explain the *application* process, we start with a definition of a different one: A *code generation* process does not modify any of the existing parts of a program. Instead, it only extends it with new modules that are dedicated entirely to the implementation of a new instance of a selected pattern. A tool that supports such process is described in [34].

In reality, however, the constructs that partake in the implementation of a pattern most often have more than one role in the program, and therefore are likely to exist. Thus, a more frequent process is one that *modifies* existing constructs rather then just add new ones. In such case we say that the pattern was *applied in the context of* the modified constructs.

We assume that the pattern at hand is represented as a formula. It is convenient to assume that the formula is open, where the free variables represent the *participants* (Rudiment A) of the pattern. Thus, the *application* process can be characterised as follows

1. The programmer selects a certain pattern as appropriate during the evolution of a program.
2. Next, the programmer selects the *context* of the application. This means that, most often, constructs that already exist in the program will become participants in the newly applied pattern. In practice, the programmer specifies an assignment from the free variables to existing constructs in the target program. Variables that are not assigned stand for new constructs that are to be generated.
3. The pattern is applied in the specified context. As a result, the original program is modified and/or extended to accommodate to the specification of the respective pattern.

The mathematical representation of the association of existing constructs with free variables uses the concept of *assignment*. Example 3 illustrates the expected results of an application process of Formula 5 in the context of different assignments.

Definition VI focuses on the result of the application process:

> Definition VI: *Application* (also *implementation*, *instantiation*) *of a Pattern*
>
> Given formula φ which stands for the pattern π, model $\mathfrak{M}$, and assignment $\mathfrak{A}$ from the free variables in φ to $\mathfrak{M}$, we say that *an application of* π *to* $\mathfrak{M}$ *in the context of* $\mathfrak{A}$ is a model $\mathfrak{M}'$ that "enriches" $\mathfrak{M}$ and in which φ is satisfied by $\mathfrak{A}$.
>
> An "enrichment" involves assigning the yet unassigned variables in φ with ground entities that are newly added to $\mathfrak{M}$, as well as adding ground relations among the entities in the assignment.

We refer to a process that results in an application of a pattern in $\mathfrak{M}$ as an *application process*.

Example 3: *Application processes*

Consider three different assignments into the free variables of Formula 5 as follows: Assignment $\mathfrak{A}_1$ assigns both free variables with concrete entities as follows:

- $\mathfrak{A}_1$(*Mediator*)=`FontDialogDirector`, $\mathfrak{A}_1$(*MBox*)=`Widget`
- partial assignment $\mathfrak{A}_2$ gives $\mathfrak{A}_2$(*MBox*)=`Widget`
- and assignment $\mathfrak{A}_3$ is empty

The application of Formula 5 in the context of a model $\mathcal{M}$ containing the atoms {`FontDialogDirector`,`Widget`} of type *Class* and the three different assignments above results in the following:

- The application of Formula 5 in the context of assignment $\mathfrak{A}_1$ results in adding to $\mathcal{M}$ the relation *Reference-to-Many*(`FontDialogDirector`,`Widget`).
- The application of Formula 5 in the context of assignment $\mathfrak{A}_2$ results in adding to $\mathcal{M}$ an atom `NewWidget` of type *class* and the relation *Reference-to-Many*(`FontDialogDirector`,`NewWidget`).
- The application of Formula 5 in the context of assignment $\mathfrak{A}_3$ results in adding to $\mathcal{M}$ two atoms New`Widget1`, New`Widget2` of type *class* and the relation *Reference-to-Many*(`NewWidget1`,`NewWidget2`).

Recognition is the process of searching for instances of a certain pattern in an existing, often large program. It can be defined as follows:

Definition VII : *Recognition*

Given a program *p* and a specification φ of a pattern π, a *recognition* process gives all the instances of π in *p*.

Given the formal specification of a pattern, the *recognition* process can be fully automated. Such a tool may be useful with large software systems, possibly even "legacy systems"([i]).

"Pattern mining" [38] designated the process of discovering patterns by searching for regularities ("mining") in large programs. It is the least understood of the activities mentioned in this section, and its definition highly depends on the understanding of what is, and what is not, a pattern. Nonetheless, the process of discovering patterns may benefit from a tool that searches for "candidates" for the title "pattern" by seeking regularities in a set of programs. Whether the outcome of such process is worthy for the title "pattern" is outside the scope of this article.

Definition VIII : *Discovery*

Given a specification language $\mathscr{L}$, a set of programs $\{p_i\}$, and a set of "known patterns" $\{\varphi_i\}$, a *discovery* process is a search for a pair $<\varphi,\{p_i\}>$, where

- $\varphi \notin \{\varphi_i\}$ (i.e., φ is not a "known" pattern)
- $\{p'_i\}$ is a set of instances of φ, where each p'_i is a subprogram of p_i

5.2 Relations Among Patterns

This section discusses relations among patterns and provides formalisation thereof using the definitions defined previously.

Two articles [39; 40], describe one pattern as a "refinement" of another. Cargil [41] describes what he calls "categories of patterns," where a set of patterns "refine" the specifications given by others. Kim and Benner [42] describe the *push* and *pull models*, which are "refinements of the OBSERVER pat-

[i] Naturally, the feasibility of such a tool depends on its ability to efficiently generate only "reasonable" assignments.

tern" []. Similarly, Rohnert [43] describes "specializations of the *PROXY* pattern," such as *CACHE PROXY* and *PROTECTION PROXY*.

By means of the above definitions we formalise the intuition behind *refinement*, or *specialisation* as follows:

> Definition IX: *Refinement* (also *specialisation*)
>
> We say that pattern π *refines* pattern ρ iff for every assignment $\mathfrak{A}$ that satisfies ρ in some model $\mathfrak{M}$, $\mathfrak{A}$ also satisfies π in $\mathfrak{M}$.

In the absence of formal specification, it is hard to resolve conclusively whether one pattern manifests a special case of another. This difficulty is demonstrated in a classic example reported by Vlissides [40], which tells of a debate that took place between the authors of the GoF catalogue over the question whether one pattern is a refinement of the other. Eden, Gil, Hirshfeld and Yehudai give a definition similar to Definition IX and use it to resolve the above mentioned debate [44].

Projection designates *refinements* of particular interest, where a participant *x* of the one pattern is replaced with a *uniform set* (Rudiment C) of *participants* of the same type. Formally:

> Definition X: *Projection and Abstraction*
>
> A *projection of X in $\Phi(X)$*, where X is a free variable in X of participant type $2^{\mathcal{T}}$, is a formula $\phi(x)$ where x is a free variable in ϕ of type $\mathcal{T}$ and replaces X in Φ. That is:
>
> $$\phi(x) \stackrel{\Delta}{=} \Phi\left[{}^{x}\!/\!_{X}\right]$$
>
> If this relation holds, we say that Φ is an *abstraction* of x in ϕ.

This definition extends in a straightforward manner to any number of free variables and to variables of higher orders.

A similar definition of *projection* is given in [25]. That article also illustrates how the *FACTORY METHOD* pattern results from a two steps of projection of the *Factory-Methods* and *Products* variables of the *ABSTRACT FACTORY* pattern, as specified in Formula 7. Example 4 illustrates this relation.

> Example 4: *Projecting the ABSTRACT-FACTORY*
>
> To demonstrate *FACTORY-METHOD* is a projection of the *ABSTRACT-FACTORY* we express the latter in terms of the former.
>
> To remind the reader of the specification of the *ABSTRACT-FACTORY*, the variables in Formula 7 were defined as follows:
>
> (i) *Factory-Methods* $\in 2^{2^{\mathbb{F}}}$ (ranges over sets of sets of functions)
>
> (ii) *Products* $\in 2^{\mathbb{H}}$ (ranges over sets of hierarchies).
>
> For the specification of the *FACTORY-METHOD* we *project* the two variables into new ones:
>
> (i) *factory-methods* $\in 2^{\mathbb{F}}$ (ranges over sets of functions)
>
> (ii) *products* $\in \mathbb{H}$ (ranges over hierarchies, or sets of classes)
>
> Consequently, we can define *FACTORY-METHOD* in terms of *ABSTRACT-FACTORY* as follows:
>
> $$\textit{FACTORY-METHOD} \stackrel{\Delta}{=} \textit{ABSTRACT-FACTORY}\left[{}^{factory-methods}\!/\!_{Factory-Methods}, {}^{products}\!/\!_{Products}\right]$$

6. Comparative Criteria

Looking to compare the relative merits of proposed specification languages, we provide criteria for the comparison of pattern specification languages. In this section we draw conclusions from the framework we have presented. Thanks to the formal foundation we laid, some of these criteria, namely: *concision*, *completeness*, and *expressiveness*, can be now measured precisely. Later we discuss *compactness* which, despite its significance, could not be well defined at this point.

Clearly, allowing for shorter expressions is an important property of a specification language. The following definition allows us to compare the relative concision of specifications:

> Definition XI: *Concision*
>
> Given two specification languages Λ and Γ, a numeric constant c, and a metric function L: which measures the length of expressions in both languages and returns a natural number. Let $\|\pi\|_\Lambda$ designate the specification of π in Λ, and let Π designate a fixed set of patterns.
>
> Λ is *more concise than* Γ *with respect to* π iff $L(\|\pi\|_\Lambda) < L(\|\pi\|_\Gamma)$.
>
> Λ is *c-concise with respect to* Π iff for every π in Π, $L(\|\pi\|_\Lambda) < c$.

Naturally, different formalisms give rise to different subsets of $\mathbb{P}$, and thus may or may not account for design patterns of interest. Since this capacity is fundamental to the issue of specification, we offer *descriptiveness* and *completeness* as means to measure it.

Given a fixed set of patterns Π we define *completeness* as follows:

> Definition XII: *Completeness*
>
> We say that $\mathscr{L}$ is *complete* with respect to Π iff every pattern in Π can be described by $\mathscr{L}$.

Nevertheless, the question whether a certain formula φ describes π is difficult to answer conclusively exactly because the contemporary means of specification are ambiguous. This problem may be alleviated by providing a different measure, *expressiveness*, which focuses on the observations made in section 2:

> Definition XIII: *Expressiveness*
>
> A specification language $\mathscr{L}$ is ***expressive*** if it incorporates the rudiments listed in section 2, namely:
>
> - *participants* (Rudiment A)
> - *ground relations* (Rudiment B, which can be defined between *participants*)
> - *uniform sets* (Rudiment C) and *higher order sets* (Rudiment D) of *participants*
> - Relations between higher order sets, in particular bijections (Rudiment E)
> - Function families (Rudiment E), in particular *clans* (Definition I) and *tribes* (Definition II)
> - *hierarchies* (Rudiment G)

Note that a uniform definition of *class* and *method* is not a prerequisite to Definition XIII. Although different formalisms offer different interpretations to these concepts, what counts is the means the formalism provides to engage sets of participants in the required correlations, and the microarchitectures that can be described by these means. Note also that in sections 3 and 4 we treat *participants* as well defined abstractions.

The understanding of software patterns shall greatly benefit from their specification in relatively short formulae. "Small" formulae should allow a clearer and more understandable picture of these systems. Similarly, for visual specification languages, a small number of elements on each diagram allows clearer and more comprehensible diagrams.

Compactness. A trivial candidate to satisfy expressiveness is a language that is a superset of all the concepts of section 2, such as higher order logic ([i]). Observe that a language can be "too expressive" if it incorporates many more expressions and constructs than it is necessary for our purpose.

---

[i] A detailed proof that these constructs can be expressed in higher order logic appears in [Eden, Gil, Hirshfeld & Yehudai 99a].

Following is a summary for the reasons for a specification language to be as small as possible:

1. ***Abstraction***. Leitmotifs (of design patterns) are abstractions that manifest the essentials of systems' architecture. When essentials are considered (rather than details), we expect that only a small number of expressions should be sufficient to accommodate to our needs. The more variation among the possible expressions is allowed, the more likely are the specifications to become cluttered and elaborated, which defies the purpose of the abstraction process.

2. ***Reasoning***. A smaller set of possible expressions facilitates the reasoning process on the properties of the abstractions that are given rise by it and leaves room for more proofs on their properties.

3. ***Discovery***. The semi-automation of the process of *discovery* (Definition VIII) might become feasible if the set of possible patterns is limited by the specification language, and a tool that engages in the search process is more likely to give meaningful results.

Examples for subsets of $\mathbb{P}$ that bear little relevance to static architecture are:

- The set of programs that contain exactly *k* inheritance relations (for some constant *k*)
- The set of programs that can be written in the Java™ programming language
- The set of programs the do not terminate

We conclude that languages may be "too expressive" for our purpose and therefore look to define *compactness*. Thus, *compactness* has the intuitive meaning of a language that does not allow, or allows a minimal number, of "unreasonable" leitmotifs.

While we have already acknowledged the difficulty in defining the domain of a prospective specification language, more specifically, which subsets of $\mathbb{P}$ are of interest, it is even more difficult to characterise the subsets are not. However, since the idea of "unreasonable" leitmotifs has yet to be defined, we could not provide at this point a precise measure for compactness.

7. Conclusions and Summary

We have established a formal framework for the discussion in O-O software architecture by observing primitive and composite abstractions in its specification and by offering a logic model for representing and deliberating these abstractions. We have demonstrated how examples in various pattern specification languages map to our framework. We have used our framework to define informal concepts in formal terms. Finally, we offered means for comparative assessment of notations offered as formalism for the representation of patterns of design and architecture. This last contribution should assist in particular in light of the growing popularity of patterns and acknowledging the need for their precise specification.

7.1 Future Directions

Dynamic Specifications. Despite the achievement in specification through static properties, our frame of reference does not allow the expression of behavioural conditions. It would be of interest to integrate our results with dynamic specifications. Formalisms that may apply are Temporal Logic of Actions [45], as in [46], and Abstract State Machines [47].

Refining and Completing the Criteria. Additional constraints can be suggested to refine and improve the criteria we suggested. Compactness, for instance, may be defined as a relation between languages.

Applying the Criteria. With the few examples provided by each formalism, it is almost impossible to compare the criteria proposed to these specification languages. An interesting result can be achieved by comparing the *concision* of the specifications of elaborated patterns between two formalisms or more.

Acknowledgements

Special thanks to Dr. Björn Victor of the Department of Computer Systems, Uppsala University, for his insights and detailed comments. Many thanks also to Peter Grogono for his feedback. The authors also wish to thank also to Marie Gerard for her support.

References

1 D. E. Perry, A. L. Wolf (1992). “Foundation for the Study of Software Architecture”. ACM SIGSOFT *Software Engineering Notes*, 17 (4), pp. 40-52.

2 C. A. R. Hoare (1985). *Communicating Sequential Processes.* Prentice-Hall, 1985.

3 R. Allen, D. Garlan (1997). “A Formal Basis For Architectural Connection”. *ACM Transactions on Software Engineering and Methodology*, July 1997.

4 T. R., Dean, J. R. Cordy (1995). “A Syntactic Theory of Software Architecture”. *IEEE Transactions on Software Engineering*, vol. 21, no. 4, April 1995.

5 G. Abowd, R. Allen, D. Garlan (1993). “Using Style to Understand Descriptions of Software Architecture”. Proceedings of the ACM SIGSOFT symposium on the foundations of software engineering, December 1993.

6 J. M. Spivey (1989). *The Z Notation: A Reference Manual*. Prentice-Hall.

7 D. Harel (1987). “Statecharts: A Visual Formalism for Complex Systems.” *Science of Computer Programming*.

8 C. A. Petri (1962). “Communications with Automata”. Technical report RADC-TR-65-377, Applied Data Research, Princeton, N. J.

9 G. Odenthal, K. Quibeldey-Cirkel (1997). “Using Patterns for Design and Documentation”. *Proceedings of the European Conference of Object Oriented Programming* 1997. Lecture Notes in Computer Science. Berlin: Springer.

10 I. Craig (1999). *The Interpretation of Object-Oriented Programming Languages*. Berlin: Springer-Verlag.

11 N. Medvidovic, R. N. Taylor (1997). "A Framework for Classifying and Comparing Architecture Description Languages". In *Proceedings of the Sixth European Software Engineering Conference*, together with *Fifth ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pp. 60-76, Zurich, Switzerland, September 22-25, 1997.

12 `pattern-discussion`. Mailing list:
`http://hillside.net/patterns/Lists.html#patterns-discussion`

13 `gang-of-four-patterns`. Mailing list:
`http://hillside.net/patterns/Lists.html#gang-of-4`

14 E. Gamma, R. Helm, R. Johnson, J. Vlissides (1994). *Design Patterns: Elements of Reusable Object Oriented Software*. Addison-Wesley.

15 C. Alexander, S. Ishikawa, M. Silverstein, M. Jacobson, I. Fixdahl-King, S. Angel (1977). *A Pattern Language*. Oxford University Press, New York.

16 C. Alexander (1979). *The Timeless Way of Building.* Oxford University Press, New York.

17 A. H. Eden (1998d). “Giving ‘The Quality’ a Name”. Guest column, *Journal of Object Oriented Programming*, June 98. SIGS Publications.
`http://www.math.tau.ac.il/~eden/bibliography.html#joop98`

18 K. Popper (1969). *Conjectures and Refutations.* London: Rutledge.

19 A. H. Eden, Y. Hirshfeld, A. Yehudai (1999a). “Towards a Mathematical Foundation for Design Patterns”. Technical report 1999-004, Department of Information Technology, Uppsala University.
`http://www.math.tau.ac.il/~eden/bibliography.html#ecoop99`

20 A. H. Eden, Y. Hirshfeld, A. Yehudai (1998a). “LePUS - A Declarative Pattern Specification Language”. Technical report 326/98, department of computer science, Tel Aviv University.

21 J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen (1991). *Object Oriented Modeling and Design*. Prentice Hall.

22 G. Booch (1994). *Object Oriented Analysis and Design With Applications*. 2nd edition. Benjamin/Cummings.

23 D. G. Firesmith, I. Graham, B. Henderson-Sellers (1998). *Open Modeling Language (Olm) Reference Manual*. Cambridge University Press.

24 J. Barwise (1976, ed.) *Handbook of Mathematical Logic*, pp. 4-56.

25 A. H. Eden, J. Gil, Y. Hirshfeld, A. Yehudai (1999b). "Motifs in Object Oriented Architecture". Submitted: *IEEE Transactions on Software Engineering.* `http://www.math.tau.ac.il/~eden/bibliography.html#miooa`

26 A. Lauder, S. Kent (1998). "Precise Visual Specification of Design Patterns". *ECOOP 98*.

27 T. Mikkonen (1998). "Formalizing Design Patterns". *Proceedings of ICSE 98,* IEEE Computer Society Press, pp. 115-124.

28 L. Lamport (1994). "The Temporal Logic of Actions." *ACM Transactions on Programming Languages and Systems*, vol. 16, no. 3, May 1994, pp. 872-923.

29 G. Booch, I. Jacobson , J. Rumbaugh (1999). The Unified Modeling Language Reference Manual. Addison-Wesley.

30 R. Helm, I. M. Holland, D. Gangopadhyay (1990). "Contracts: Specifying Compositions in Object Oriented Systems". Proceedings of OOPSLA 1990, SIGPLAN Notices, vol. 25 no. 10.

31 G. Florijn, M. Meijers, P. van Winsen (1997). "Tool Support in Design Patterns". In: Askit M., S. Matsuoka (1997, eds.) *Proceedings of the 11th European conference on Object Oriented Programming – ECOOP 97*. Lecture Notes in Computer Science no. 1241. Berlin: Springer-Verlag.

32 A. H. Eden, J. Gil, A. Yehudai (1997c). "Precise Specification and Automatic Application of Design Patterns". *The 12th IEEE International Automated Software Engineering Conference - ASE 1997*. `http://www.math.tau.ac.il/~eden/bibliography.html#ase`

33 P. P. Pal (1995). "Law-Governed Support for Realizing Design Patterns". *TOOLS USA* 17.

34 F. J. Budinsky, M. A. Finnie, J. M. Vlissides, P. S. Yu (1996). "Automatic code generation from design patterns". *Object Technology*, Vol. 35, No. 2.

35 Quintessoft Engineering, Inc. (1997). C++ Code Navigator 1.1. `http://www.quintessoft.com`

36 R. K. Keller, R. Schauer, S. Robitaille, P. Page. "Pattern-based Reverse Engineering of Design Components." In: *Proceedings of the Twenty-First International Conference on Software Engineering*, pages 226-235, Los Angeles, CA, May 1999. IEEE.

37 S. Chiba (1995). "A Metaobject Protocol for C++". In: *Proceedings of OOPSLA'95*.

38 J. Coplien (1995). "Pattern Mining". *C++ Report*, Oct. 1995. SIGS Publications.

39 E. Agerbo, A. Cornils. (1998). "How to Preserve the Benefits of Design Patterns". OOPSLA 98.

40 J. Vlissides (1997a). "Multicast". *C++ Report*, Sep. 97. SIGS Publications.

41 T. Cargill (1996). "Localized Ownership: Managing Dynamic Objects in C++". In: J. M. Vlissides, J. O. Coplien, N. L. Kerth (1996, eds.) *Pattern Languages in Program Design 2*. Addison-Wesley.

42 J. J. Kim, K. M. Benner (1996). "Implementation Patterns for the Observer Pattern". In: J. Vlissides, J. O. Coplien, N. L. Kerth (1996, eds.) *Pattern Languages in Program Design 2*. Addison-Wesley.

43 H. Rohnert (1996). "The Proxy Design Pattern Revisited". In: J. Vlissides, J. O. Coplien, N. L. Kerth (1996, eds.) *Pattern Languages in Program Design 2*. Addison-Wesley.

44 A. H. Eden, Y. Hirshfeld, A. Yehudai (1998c). "Multicast - Observer ≠ Typed Message". *C++ Report*, October 1998, SIGS Publications. `http://www.math.tau.ac.il/~eden/bibliography.html#cppr98`

45 L. Lamport (1994). "The Temporal Logic of Actions." *ACM Transactions on Programming Languages and Systems*, vol. 16, no. 3, May 1994, pp. 872-923.

46 T. Mikkonen (1998). "Formalizing Design Patterns". *Proceedings of ICSE 98,* IEEE Computer Society Press, pp. 115-124.

47 Y. Gurevich (1994). "Evolving Algebras." In IFIP 13th World Computer Congress 1994, Volume I: Technology and Foundations, eds. B. Pehrson and I. Simon, North-Holland, Amsterdam, 423-427.