Chapter 8

Charaterising Object-Based Frameworks in First-Order Predicate Logic

Shui-Ming Ho and Kung-Kiu Lau

School of Computer Science, University of Manchester
Oxford Road, Manchester M13 9PL, U.K.

In the component-based approach Catalysis, a framework is a reusable artefact that can be adapted and composed into larger systems. The *signed contract* between components specifies how the required properties of one component are satisfied by the provided properties of another. We examine this concept in the context of framework-based development. Although Catalysis advocates rigorous development, frameworks lack a comprehensive formal foundation. We consider a simplified view of frameworks and their transformation into first-order logic. Theorem proving may be used to check the consistency of framework specifications and we identify ways in which these specifications may be simplified beforehand to reduce the burden of proof.

Contents

8.1. Introduction . . . 222
8.2. Catalysis Frameworks . . . 223
8.3. A Cursory Overview of FML . . . 225
  8.3.1. Datatypes . . . 225
  8.3.2. Classes and Associations . . . 225
  8.3.3. Behavioural Elements . . . 227
  8.3.4. Constraints . . . 227
  8.3.5. Importing Mechanisms . . . 228
8.4. The Specification of Structure . . . 229
  8.4.1. Notation . . . 229
  8.4.2. Object Identity and State . . . 230
  8.4.3. State Space Specifications . . . 233
  8.4.4. Object Diagrams . . . 234
8.5. Structuring and Modularity . . . 235
  8.5.1. State Models . . . 237
  8.5.2. Class Extension and Composition . . . 237
8.6. Behavioural Modelling and Specification . . . 239
  8.6.1. State History Functions . . . 239
  8.6.2. Events . . . 239
  8.6.3. Constraints in FML . . . 240
8.7. Framework Consistency . . . 242
  8.7.1. Contract Composition in Catalysis . . . 242
  8.7.2. Contracts in FML . . . 243
  8.7.3. Consistency Checking . . . 243

8.8. Frameworks in Component Modelling . . . 245
8.8.1. Signed Contracts and Composition . . . 246
8.9. Related Work . . . 249
8.10. Summary . . . 250
Bibliography . . . 251

8.1. Introduction

Code is not the only reusable artefact obtained during system development. Specifications and designs also have the potential for reuse. In particular, there has been a growing interest in the identification and application of recurring patterns of interaction. The relationship between *design patterns* and *component*, for example, has previously been studied by Johnson [1] and Larsen [2]. Patterns may be applied during the development process and they may be realised by one or more software components.

In the component-based development approach *Catalysis* [3], designs patterns are at the heart of *frameworks*. Framework-based development shifts the focus or reuse from single classes to groups of classes. Underlying this approach are the concepts of *abstraction* and *refinement*: generic frameworks may be specialised and adapted to specific problem domains. The use of these frameworks may be subject to constraints and ensuring these constraints are satisfied is integral to the Catalysis approach.

Given its emergence as the *de facto* standard for object-oriented modelling, the Unified Modelling Language [4] (UML) is used to model frameworks in Catalysis. The Object Constraint Language [5] (OCL) can be used to specify their semantics. Different modelling approaches introduce their own modelling concepts and Catalysis is no exception. Although the use of UML and OCL is widespread, they are not necessarily applicable to these approaches. Since the publication of D'Souza and Wills' text on Catalysis (Ref. 3), both UML and OCL have been revised, culminating in the definitions of UML 2.0 and OCL 2.0. Both languages can express more but because of the specifics of Catalysis' modelling approach, they are still inadequate for framework modelling. Prior to OCL 2.0, much work has been devoted to increasing OCL's expressiveness. Of importance has been the need to provide greater support for business modelling and the need to express different kinds of business rules, e.g., those classified by Eriksson and Penker [6]. Catalysis has relied on its own versions of UML and OCL, at the same time prescribing a different semantics for OCL. Different notations can be used in different situations, making it difficult to fix a standard notation for framework modelling.

In this chapter, for the purpose of framework modelling, we will depart entirely from UML/OCL. Instead, we make use of a textual language, Framework Modelling Language (FML), for defining frameworks from scratch. The intention is that FML can be used to define the *core* structural and behavioural properties of frameworks. However, these are informal descriptions and we will examine how frameworks in FML might be formalised. Many existing approaches to formalising UML and OCL are based on the transformation of UML models and OCL constraints into

an existing formalism. Given the differences in UML/OCL and Catalysis' version of these languages, the applicability of these approaches to frameworks is limited. In this chapter, we outline how frameworks defined in FML may be translated into first-order logic. The resulting specification may be submitted for theorem proving. Where appropriate, this specification may be simplified beforehand. These simplifications are explored in Sec. 8.7..

8.2. Catalysis Frameworks

That different aspects of a system can be modelled independently, then assembled together, is not new: *role modelling* in OORAM [7], *aspect-oriented development*, and framework modelling in Catalysis are variations of this concept.

Traditionally, frameworks are defined as groups of interacting objects. Figures 8.1. and 8.2. show two such frameworks. These domain specific frameworks describe the *roles* a person plays in different contexts: that of a driver and an employee. Figure 8.1. shows the structural relationship between drivers and their cars. Collaborative behaviours can be expressed as joint actions, as illustrated in Fig. 8.2.: employees and companies collaborate in the action employ. Joint actions may be thought of as system level behaviours, much like use cases in object-oriented analysis. They may be decomposed into much smaller actions (*local actions*, or messages) which occur at the object level. The decomposition of these actions may be represented in a similar manner to use cases in UML. The discussion of frameworks in this chapter, however, will not be concerned with such decompositions.

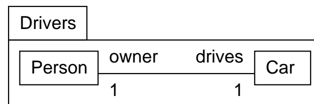

Fig. 8.1. People playing the role of drivers.

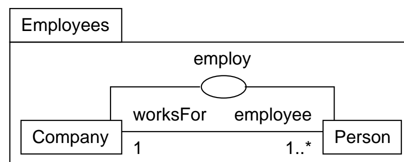

Fig. 8.2. Companies and employees collaborating in the action employ.

The above frameworks can be augmented with constraints, which can be expressed in either OCL or in any other notation, e.g., natural language.

Model Synthesis/Composition. The models of Figs. 8.1. and 8.2. may be synthesised, or composed, to form a new framework Drivers+Employees. Figure 8.3. shows the *unfolded* view of the framework. Whether this synthesis is possible in UML is dependent upon how well defined the package extension mechanism of UML is. For a discussion of UML's package extension mechanisms, the reader is referred to Cook *et al.*'s discussion [8]. In Catalysis, the derivation of Drivers+Employees is expressed either using a package dependency diagram (Fig. 8.4.a) or as a pattern application (Fig. 8.4.b).

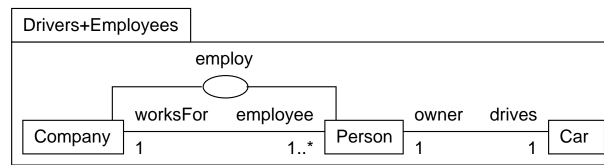

Fig. 8.3. The synthesis of the Driver and Employee frameworks.

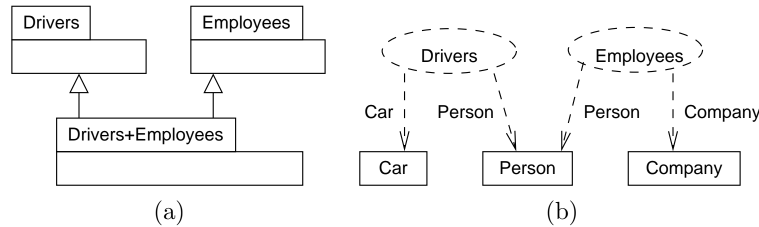

Fig. 8.4. The derivation of Drivers+Employees can be expressed in two ways: (a) using package dependency diagrams; or (b) using pattern application diagrams.

External Interactions. Usually, objects within a framework collectively maintain some invariant, which must be observed by interactions *within* the framework and also by interactions between objects in different frameworks. This latter situation arises when model composition occurs and the behavioural models of each component framework must be unified. The synthesis of frameworks may result in invalid models either because constraints conflict or existing ones are too weak, resulting in unexpected behavioural models.

An *effect invariant* is a constraint on actions both within a framework (*internal actions*) and those defined elsewhere (*external actions*). They are usually expressed as *trigger rules*, similar to the behavioural contracts of Helm *et al* [9]. For example, in the *Observer* pattern an effect invariant could specify the sequence of actions that take place whenever the state of a subject changes: the trigger—a change in the subject's state—results in the subject sending a notification to its observers, in turn, causing them to update themselves. OCL 2.0 goes some way toward expressing

these kinds of rules. However, these rules can only occur within the context of a single class although, in general, rules may be defined on the whole system.

From the informal to the formal

The remainder of this chapter is devoted to the problem of deriving a formal specification, *spec*(F) of an FML framework F. Ideally, we would like to fix a *standard* notation for defining frameworks and their properties. Catalysis makes use of its own extensions to UML/OCL. The notations may be intuitive but they lack a proper definition within the (informal) semantic framework of UML/OCL. The extensions either require subtle changes to the existing semantics of OCL, or complicate the language unnecessarily. For this reason, FML is used.

8.3. A Cursory Overview of FML

In the majority of UML models there are two kinds of *model element*: structural elements and behavioural elements. In FML, structural elements correspond to classes, attributes, and associations; behavioural elements correspond to events that occur within the framework.

8.3.1. *Datatypes*

Common to many object-oriented (or object-based) languages is the notion of a set of basic data types (e.g., Integer, String, etc.) from which more complex data structures can be built. UML is no exception and basic data types are defined within a package called DataTypes. In FML, we will assume the existence of a corresponding framework, DataTypes, which contains the ADT definitions of primitive types. Parameterised collection types (bags, sets, and sequences) also exist in UML/OCL. In the sequel we make use of one such collection: Set(Data), sets of Data elements. In addition to primitive types it is also possible to define additional datatypes within a framework. This is illustrated later.

8.3.2. *Classes and Associations*

The FML declaration

```
framework F
  class C {}
```

defines a class C in framework F. The class has no *features* (or *fields*), which is denoted by the empty parentheses. The term *feature* is used to refer to a property of a class as attributes and associations are not conceptually distinguishable as they are in UML.

The features of a class may be represented by either functions or predicates. In the declaration

```
class C { f: -> T;  p: (T) }
```

the class C has two features, denoted by the function f and the predicate p. In general, f and p may either be attributes of class C or associations in which instances of C participate. If T is a class, then f can be interpreted as the name of an association that is navigable from C to T, through which a single instance of T is returned. The feature p may be interpreted as the name of an association which, when navigated, results in any number of T-instances. This loosely corresponds to the situation in UML where an association p is marked with the variable multiplicity marker * at the association end at T. Given the above, the correspondence between features in FML and attributes and associations in UML can be described as follows.

Let F be a framework expressed in Catalysis' extended UML notation and let F consist of the n classes read from the diagram: $C_1,\ldots,C_n$. The corresponding framework in FML will be denoted by F and the classes of F will be denoted by $C_1,\ldots,C_n$.

Attributes. Suppose the class C_1 is defined as follows:

$$C_1 \texttt{ \{ a: -> T; b:(}t_1\texttt{,...,}t_k\texttt{, T) \}}$$

(1) The feature a corresponds to the UML attribute declaration a: T in C_1.
(2) The feature b corresponds to the (Catalysis) *parameterised* attribute b with parameter types $t_1,\ldots,t_k$ and result type T.

Binary Associations. Suppose R is a binary association joining the classes C_2 and C_3 (where both classes have no attributes or other associations) and that each association end is labelled with the role names r_2 and r_3 respectively.

(1) The declaration C_2 { r_3: -> C_3 } corresponds to the situation where the association end named r_3 has the multiplicity constraint 1.
(2) The declaration C_3 { r_2: (C_2) } corresponds to the situation where the association end named r_2 has any of the following multiplicity constraints: 0..*, 1..*, or 0..1.

In the above, it is assumed that navigation is possible from one class to another if and only if there is a role name at the target end of the association. This gives rise to a third condition.

(3) In the absence of role names, the direction of navigation may be explicitly marked as being from C_2 to C_3 in which case we either have the declaration

$$C_2 \texttt{ \{ R: -> } C_3 \texttt{ \}} \quad \text{or} \quad C_3 \texttt{ \{ R: (} C_3 \texttt{) \}} \;,$$

depending on the multiplicity at C_3 as outlined above.

Item (2) above represents a departure from UML/OCL in that it deviates from the usual notion of navigation through an association. An alternative is to allow collection types, i.e., r_2: -> Set(C_2). Other collections may be used. For example, if r_2 had the UML stereotype *ordered*, then we might use the declaration

$\mathtt{r_2}$`: -> Sequence(`$\mathtt{C_2}$`)` instead. The use of collection types allows us to express a greater range of multiplicity constraints on association ends.

***n*-ary Associations.** These associations represent a general relationship between n participant classes. Unlike binary associations, however, an n-ary association is not navigable. If R is a ternary association between C_1, C_2, and C_3, then each of those classes has a feature named R, e.g., a feature `R: (`C_2`,`C_3`)` in class C_1.

It should be noted that the above covers only a subset of UML's repertoire of associations. The frameworks considered in this chapter are *object-based* frameworks, not *object-oriented*. Although inheritance may be used for more interesting designs, it may also introduce its own problems, particularly where behavioural overriding occurs [10]. Classes in FML are considered as defining a *traits* rather than taxonomic structures.

Additionally, UML has the relationships of composition and aggregation. How aggregation should be applied has been the subject of much discussion. Both these relationships, however, may be represented as normal associations coupled with suitable constraints to represent either a composition or aggregation relationship.

8.3.3. *Behavioural Elements*

Whereas Catalysis talks about *actions*, in FML observable behaviours are described using *timed events*. An event is instantaneous and corresponds to the occurrence of some action. Alternatively, it may represent some signal that is raised whenever some property holds at a given point in time.

A timed event in `F`, declared as $\mathtt{m(C_1,\ldots,C_m,t_1,\ldots,t_n,Time)}$, where $C_1,\ldots,C_m$ are the participant classes of the action and $t_1,\ldots,t_n$ are additional parameter types of the action, can be interpreted in two ways:

(1) there is a joint action $\mathsf{m}(t_1,\ldots,t_n)$ in F in which instances of $C_1,\ldots,C_m$ participate; or
(2) there is a local action m initiated by an instance of C_1, which involves objects of the other specified classes.

Actions, whether local or joint, are considered as *globally* observable behaviours in FML, occurring within an explicit time context, as dictated by the `Time` parameter.

8.3.4. *Constraints*

Conceptually, objects have state histories and in FML it is possible to refer to an object as it is at specific points in time. We denote the state of an object `x` at time `t` by the term `$(x,t)`. As usual, the dot notation is used to represent feature access. Thus, if `a` is a function feature of `x`, then the term `$(x,t).a` denotes the value of `a` for object `x` at time `t`.

Constraints on objects are introduced as *facts* of the framework. Facts are expressed as formulae in first-order logic using FML's notation. As is the case with events, constraints are always global properties unlike OCL, where constraints are

always local properties of classes.

As an example, the multiplicity on the role named employee in Fig. 8.2. may be expressed as the situation where the set employee must not be empty:

```
fact oneOrMoreEmployees {
  all c: Company, t: Time:: !empty($(c,t).employee)
} .
```

This assumes that association ends with variable multiplicity are represented as functions returning collections. In this case, employee is assumed to be a function with target type Set(Person) as opposed to a predicate employee: (Person).

Preconditions and postconditions may be attached to events. The employ declaration may be attached with a pre- and postcondition as follows:

```
event employ(c: Company, p: Person, t: Time) {
  pre:   !mem(p, $(c,t).employee)
  post:  $(c,next(t)).employee = add(p,$(c,t).employee)
} .
```

The term $(c,next(t)) denotes the state of the object c at time t+1 and the post-condition states that employ results in a person p being added to the set employee of c at time t+1.

8.3.5. *Importing Mechanisms*

The import mechanism of FML provides the basis for framework composition and extension. An importing framework may extend existing frameworks in a number of ways. It may define new features for classes, new events, and new constraints. In addition, the importing framework may rename elements from imported frameworks.

A statement "import F[A\B, A.f\g]" denotes the importing of the framework F subject to two conditions: the class (or type) A is renamed to B and the feature f of A is renamed to g. Renaming allows us to force classes from different frameworks to be considered as partial definitions of the same class.

Example: Suppose that two FML frameworks Drivers and Employees have been defined. Then the derivation of Drivers+Employees is accomplished by

```
framework Drivers+Employees
  import Drivers
  import Employees .
```

This is the *structured definition* of the framework. The body of the framework module consists of two import statements, analogous to a textual inclusion of the bodies of the Drivers and Employees frameworks. The consequence is that there are two Person definitions: one with a feature drives and the other a feature worksFor.

Suppose there is a procedure *flatten*, which takes the structured definition of F and produces the unfolded view of F. The purpose of *flatten* is to resolve multiple

definitions of the same class and to produce a single definition of that class. The *unfolded* view, or *flat definition*, of `Drivers+Employees` is shown in Fig. 8.5.. The *flatten* procedure takes the definitions of `Person` from `Drivers` and `Employees` and amalgamates them into the single definition shown.

```
framework Drivers+Employees
  class Person { drives: -> Car,  worksFor: -> Company
  }
  class Car { owner: -> Person
  }
  class Company { employee: -> Set(Person)
  }
  fact oneOrMoreEmployees {
    all c: Company, t: Time:: !empty($(c,t).employee)
  }

  event employ(c: Company, p: Person, t: Time) {
    pre:  !mem(p,$(c,t).employee)
    post: $(c,next(t)).employee = add(p,$(c,t).employee)
  }
```

Fig. 8.5. The unfolded `Drivers+Employees` framework in FML.

In the following sections we will look at how, given the FML definition of a framework in FML, we can derive its specification in many-sorted first-order logic. For structured definitions, we can define corresponding *structured specifications*. Using the *flatten* procedure, we can obtain flat definitions and, correspondingly, obtain *flat specifications*. The next section explores the specification of the static structure of frameworks. In particular, the presentation will be concerned with the definition of state spaces for objects. Classes are given a semantics by defining how all possible states (or *local object configurations*) can be generated. Thereafter, the specification of constraints and the dynamic aspects of frameworks is discussed.

8.4. The Specification of Structure

For the time being, we concentrate on class specifications as opposed to framework specifications. The idea presented here is that from the FML definition of a class `C`, we can construct a state space specification *vspec*(`C`). The specification *vspec*(`C`) enumerates all the possible *state values* that instances of `C` may take, each value representing a unique assignment of values to the features of `C`.

8.4.1. *Notation*

Let `F` be a framework and let $\mathtt{C} = \{\mathtt{C}_1, \ldots, \mathtt{C}_n\}$ be the set of classes that participate in `F`. The specification of `F` is denoted by *spec*(`F`) and the specification of class $\mathtt{C}_\mathtt{i}$

will be denoted by *spec*($\texttt{C}_\texttt{i}$).

For each class $\texttt{C}_\texttt{i}$, let there be two lists, *funs*($\texttt{C}_\texttt{i}$) and *preds*($\texttt{C}_\texttt{i}$). The former is the *lexicographically ordered* list of the function features of $\texttt{C}_\texttt{i}$ and the latter is a similarly ordered list of predicate features of $\texttt{C}_\texttt{i}$. In the `Drivers+Employees` framework this would mean

$$\textit{funs}(\texttt{Person}) = [\ \texttt{drives: -> Car}, \texttt{worksFor: -> Company}\ ]$$

and

$$\textit{preds}(\texttt{Person}) = [\ ]\ .$$

8.4.2. *Object Identity and State*

At this point it should be noted that a class is defined by a pair of specifications: the specification of its object identities and the specification of its state space. As such, a class is considered purely in a structural sense: it is the framework that gives the class a *context*, specifying state constraints for objects and providing the link between object identities and state values.

8.4.2.1. *Object Identity*

Formally, each class $\texttt{C}_\texttt{i}$, where $\texttt{1} \leq \texttt{i} \leq \texttt{n}$, has associated with it a corresponding sort C_i. We call the sorts $C_1, \ldots, C_n$ the *identifier* (or *reference*) *sorts* for $\texttt{C}_\texttt{1}, \ldots, \texttt{C}_\texttt{n}$. Intuitively, the identifier sorts serve two purposes: firstly, the data elements of the sort act as names (or object identifiers) for the individual instances of $\texttt{C}_\texttt{1}, \ldots, \texttt{C}_\texttt{n}$; secondly, they act as object reference types in associations. A feature $\texttt{a: -> C}_\texttt{j}$ of class $\texttt{C}_\texttt{i}$ therefore represents an association from $\texttt{C}_\texttt{i}$ to $\texttt{C}_\texttt{j}$.

For each class $\texttt{C}_\texttt{i}$, we can associate with it an abstract datatype of object identifiers, i.e., an abstract datatype in which there is one constant $c_i0 : [] \rightarrow C_i$ and a function $next : [C_i] \rightarrow C_i$ for generating successive object names. There are therefore an infinite number of object names for each class.

8.4.2.2. *Object States*

Associated with each class $\texttt{C}_\texttt{i}$ is a sort C_iState. The sort C_iState is called the *object value sort* of class $\texttt{C}_\texttt{i}$ and its data elements the *object values* or *state values* of $\texttt{C}_\texttt{i}$. The set of data elements of C_iState represents the state space of $\texttt{C}_\texttt{i}$. Each state value $x : C_iState$ denotes an object configuration corresponding to a specific assignment of values to the function features of an object and a specific state over which properties (predicate features) hold.

8.4.2.3. *Features*

The state of an object is observable through its associated features. The state of a $\texttt{C}_\texttt{i}$-object at any point in time is given by a single state value $x : C_iState$. For each function `a: -> T` in *funs*($\texttt{C}_\texttt{i}$) the specification contains a corresponding function

symbol

$$a : [C_i State] \to T \ .$$

Similarly, for each predicate $\mathtt{p} : (\mathtt{t}_0, \ldots, \mathtt{t}_n)$ in *preds*($\mathtt{C_i}$) there is a corresponding predicate

$$p : [C_i State, t_0, \ldots, t_n] \ .$$

8.4.2.4. *Generating Object States*

Here, we consider the specifications of object states for the classes $\mathtt{C_i}$. We denote these specifications by *vspec*($\mathtt{C_i}$). The specifications *vspec*($\mathtt{C_i}$) describe how object states, $x : C_i State$, are generated.

Undefined States. For each class $\mathtt{C_i}$ we require that among its object values is one which represents the undefined state for any $\mathtt{C_i}$-instance. This may be represented by the constant

$$null : \ [] \to C_i State \ .$$

In FML this undefined value is denoted as `null`.

Defined States. To enumerate all *defined* (non-*null*) states of $\mathtt{C_i}$ we require appropriate state generation functions. These functions may be derived according to the lists *funs*($\mathtt{C_i}$) and *preds*($\mathtt{C_i}$). If a class has function features then there must be a constructor which assigns to each of these features a value. If *funs*($\mathtt{C_i}$) = [$\mathtt{a_1}$: -> $\mathtt{t_1}$, ..., $\mathtt{a_k}$: -> $\mathtt{t_k}$], then a constructor

$$C_i^* : [t_1, \ldots, t_k] \to C_i State$$

is introduced. A ground term $C_i^*(x_1, \ldots, x_k)$ denotes a state in which the features $a_1, \ldots, a_k$ are set to the values $x_1, \ldots, x_k$ respectively. We call the states generated by C_i^* constructors *initial states* of $\mathtt{C_i}$.

Previously, it was mentioned that one way of dealing with collections is to take a predicative approach whereby $x : T$ is an element in a collection p at state $c : C_i State$ if $p(c, x)$ holds. In this case, the intended object state denoted by the term $C_i^*(x_1, \ldots, x_k)$ is one where each of the collections $\mathtt{p}$ in *preds*($\mathtt{C_i}$) is empty.

Note that if *funs*($\mathtt{C_i}$) = [] then the nullary function $C_i^* : [] \to C_i State$ is introduced. Therefore, the simplest class $\mathtt{C_j}$, which has no features, can be in either one of two states: *null* or C_j^*—in an undefined state or *some* defined state.

If we are dealing with a predicative approach to collections or if there are parameterised features then we need constructors for generating the different states over which each predicate holds. For each predicate $\mathtt{p} : [\mathtt{t}_1, \ldots, \mathtt{t}_k]$ in *preds*($\mathtt{C_i}$) we introduce a constructor

$$assertp : [C_i State, t_1, \ldots, t_k] \to C_i State \ .$$

Intuitively a term $assertp : [s, x_1, \ldots, x_k]$ denotes a state $s' : C_i State$ for which the formula $p(s', x_1, \ldots, x_k)$ holds.

8.4.2.5. *Constructor and Feature Axioms*

We can take advantage of the conventions used to generate constructors (i.e., the lexicographical ordering of arguments in C_i^* constructors) to derive appropriate axioms for both the constructors and features of each class. For functions, equations describe how they can be evaluated over each state; for predicates, we can describe over which states a property can be computed to hold; and for constructors, we can identify some standard properties and generate axioms appropriate properties accordingly.

Evaluating Features. Let *funs*($\mathtt{C_i}$)= [$\mathtt{f_1}$`: -> `$\mathtt{t_1}$`, ..., `$\mathtt{f_k}$`: -> `$\mathtt{t_k}$] and let $\mathtt{f_j}$ ($\mathtt{1} \leq \mathtt{j} \leq \mathtt{k}$) be a function in *funs*($\mathtt{C_i}$). An axiom

$$(\forall x_1 : t_1, \ldots, x_k : t_k)\ (f_j(C_i^*(x_1, \ldots, x_k)) = x_j)$$

describes the evaluation of the function f_j over the initial states of $\mathtt{C_i}$. Additional axioms are required to specify how the function f is evaluated over more complex states, i.e., those states reached through the application of the *assert* constructors. For each predicate `p` in *preds*($\mathtt{C_i}$), an axiom

$$(\forall x : C_i State, x_1 : t_1, \ldots, x_l : t_l)\ (f(assertp(x, x_1, \ldots, x_l)) = f(x))$$

is generated. Similarly, for each predicate p, an axiom

$$p(assertp(x, x_0, \ldots, x_k), y_0, \ldots, y_k) \Leftrightarrow (x_0 = y_0 \wedge \ldots \wedge x_k = y_k) \vee p(x)$$

is generated.

Assert Constructors Properties. For the *assert* constructors we can specify a number of properties possessed by them. As discussed earlier, a predicate approach to representing collections can be used, in which case we would like the *assert* constructors to have the same properties as the member addition constructors of sets. The *assert* constructors therefore have the properties of being idempotent and commutative, i.e., for each predicate p we have the equations

$$assertp(assertp(x, \vec{y}), \vec{y}) = assertp(x, \vec{y})$$

and

$$assertp(assertp(x, \vec{x}), \vec{y}) = assertp(assertp(x, \vec{y}), \vec{x})\ .$$

The situation becomes more complex when dealing with multiple predicate features as different terms involving different *assert* constructors may refer to the same state. For instance, a class *Person* with features

$$name : [] \rightarrow String, \quad parents : [String], \quad \text{and } children : [String],$$

has associated with it the constructor

$$Person^* : [String] \rightarrow PersonState$$

and the *assert* constructors

$$assertParents : [String] \rightarrow PersonState$$

and

$$assertChildren : [String] \rightarrow PersonState \ .$$

Let $x : PersonState$ be a state in which the following holds:

$$name(x) = \text{Noel}, \ \ parents(x, \text{Mary}), \ \ children(x, \text{Ann}) \ .$$

The state x can be reached from the initial state $Person^*(\text{Noel})$ either by applying *assertParents* first, i.e.,

$$assertChildren(assertParents(Person^*(\text{Noel}), \text{Mary}), \text{Ann}) \ ,$$

or by applying *assertChildren* first, i.e.,

$$assertParents(assertChildren(Person^*(\text{Noel}), \text{Ann}), \text{Mary}) \ .$$

Interpreting the Undefined State. There are two ways of viewing this. The first approach is to take the view that if an object is in an undefined state then any query on its features results in an undefined value. Thus, a function applied to the object value *null* would itself result in an undefined value $\perp$. Each datatype is extended to include an undefined value, resulting in the underlying logical formalism being a three-valued logic.

Alternatively, a function applied to the value *null* results in some unspecified value. Thus, the term $f(null)$ may denote any value in the range of f. In this case, the features queried over *null* may be *observationally* equivalent to those of any other C_i-state although the two states may be distinct. The idea can be extended to predicates. Here, we make the assumption that any property trivially holds for the *null* state. That is, for each predicate p we include the axiom

$$(\forall x_0 : t_0, \ldots, x_k : t_k) \ (p(null, x_0, \ldots, x_k)) \ .$$

A motivation for taking this stance is that any state that can be constructed from the undefined state must itself be the undefined state. In general, such states are constructed using the *assertp* constructors, in which case we have the axiom

$$(\forall x_0 : t_0, \ldots, x_k : t_k) \ (assertp(null, x_0, \ldots, x_k) = null) \ .$$

8.4.3. *State Space Specifications*

We can collect all the elements discussed in Sec. 8.4.2. and use them to form *object value specifications* $vspec(\mathtt{C_i})$, for each class $\mathtt{C_i}$. These specifications provide the focal point for studying class extension and composition in frameworks.

Example 8.1. Let `A` be the framework defined below.

```
framework A
  class C { }
```

Thus C is the simplest possible definable class discussed earlier. Given the above FML model, we can generate the following object value specification $vspec(\mathtt{C_A})$:

$$\begin{aligned} &vspec(\mathtt{C_A}) = \\ &\quad sorts: \quad CState \\ &\quad funs: \quad null, C^* : [] \rightarrow CState\ . \end{aligned}$$

Given that $funs(\mathtt{C}) = []$ and $preds(\mathtt{C}) = []$, according to the discussion in Sec. 8.4.2., $vspec(\mathtt{C_A})$ has no axioms. Specifications are interpreted with initial semantics. Thus, the above specification is one in which there are two distinct states.

8.4.4. *Object Diagrams*

Up to this point we have only considered state spaces. We have yet to consider the binding of object states to object names. In order to bind a $\mathtt{C_i}$-object to one of its possible states we need an assignment function

$$\$: [C_i] \rightarrow C_i State\ ,$$

which tells us which state values are associated with each object in the framework.

An instance diagram for framework A can be formalised using models of $spec(\mathtt{A})$. Note that in our *current* understanding of framework specifications $spec(\mathtt{A})$ consists of the C-object identifier ADT specification, the object value specification $vspec(\mathtt{C})$, and the valuation function \$.

An instance diagram for A can be described by one possible model for $spec(\mathtt{A})$. For example, let $\mathcal{A}$ be a model for $vspec(\mathtt{C_A})$ extended with the valuation function $\$: [C] \rightarrow CState$ such that

$$\begin{aligned} C^{\mathcal{A}} &\stackrel{def}{=} \{c_1, c_2, c_3, \ldots\} \\ CState^{\mathcal{A}} &\stackrel{def}{=} \{0, 1\} \\ null^{\mathcal{A}} &\stackrel{def}{=} 0 \\ C^{*\mathcal{A}} &\stackrel{def}{=} 1 \\ \$^{\mathcal{A}} &\stackrel{def}{=} \{(c_1, 1), (c_2, 1), (c_3, 0), (c_4, 0), \ldots\}\ . \end{aligned}$$

This interpretation corresponds to an instance diagram where there are two objects in existence, $c_1 : C$ and $c_2 : C$, each of which is in the state denoted by the constant C^*. All other objects $c_3, c_4, \ldots$, which we take to be mapped to the value 0, are interpreted as being *inactive* or do not exist.

This treatment of instance diagrams is similar to Bordeau and Cheng's [12] work on giving a formal semantics for OMT object diagrams and instance diagrams. In their work, object diagrams are formalised as algebraic specifications and instance diagrams as algebras satisfying these specifications. In their case, an instance diagram corresponds to an algebra with the addition of a special element err_{C_i}, which denotes the *error object* of class C_i, and a special state $undef_{C_i}$, which is the *un-*

defined state of C_i. An axiom

$$\$(err_{C_i}) = undef_{C_i} \ ,$$

fixes the interpretation of the state of the error object. In our specifications, we do not have such objects.

Although our treatment on formalising static aspects is similar to their work, concerning the introduction of object state sorts, it must be noted that we use the term *state* differently to Bordeau and Cheng. There, the term refers to the simplest possible observation of an object, i.e., observations of objects described in state charts: "object-states are the simplest kind of attribute, as they provide a simple summary of the condition of an object." Attributes are therefore different kinds of observations, each one of which is given a name. Thus, for each attribute α of `Ci` there is a valuation function

$$\alpha : [C_i] \rightarrow T \ ,$$

the provision being that if an object $x : C_i$ contains a link to an object y and if x is an error object then y must also be an error object, i.e.,

$$\alpha(err_{C_i}) = err_T \ .$$

8.5. Structuring and Modularity

Many of the structuring mechanisms in algebraic specification, (e.g., renaming, extension/enrichment, hiding, and union) form the basis for Catalysis' notion of package extension. In this section we will consider extension/composition in-the-small, i.e., composition/extension of classes. When defining frameworks in FML, we take the composition of classes to mean the union of their individual definitions. Whether a class is extended or composed, the state space of a class becomes larger. This is reflected in the object value specifications of the extended/composite class.

Example 8.2. Recall framework `A` from Ex. 8.1. Let `B` be the framework defined below by extending `C` with the features `a` and `r` and with the type `T`.

```
framework B
import A

extend class C { a: -> T,  r: (T) }

type T { t1: -> T, t2: -> T }
```

The specification *vspec*($\mathtt{C_B}$), shown in Fig. 8.6., can be derived from *vspec*($\mathtt{C_A}$) by introducing new sorts, functions, and predicates, i.e., *vspec*($\mathtt{C_B}$) is obtained by extending *vspec*($\mathtt{C_A}$) with the function

$$a : [CState] \rightarrow T$$

and the predicate

$$r : [CState, T] \ .$$

The state space of $\mathtt{C_B}$ consists of values which encapsulate the new data introduced by `a` and `r`. States encoding this data can be generated by the new constructors

$$C^* : [T] \rightarrow CState \qquad \text{and} \qquad assertr : [CState, T] \rightarrow CState \ .$$

$$
\begin{array}{rl}
vspec(\mathtt{C_B}) = & \!\!\!\!\!\!\textbf{extend } vspec(\mathtt{C_A}) \textbf{ with} \\
\text{sorts} : & T, CState \\
\text{funs} : & t_1, t_2 : [] \rightarrow T \\
 & C^* : [T] \rightarrow CState \\
 & assertr : [CState, T] \rightarrow CState \\
 & a : [CState] \rightarrow T \\
\text{preds} : & r : [CState, T] \\
\text{axioms} : & (\exists! x : T)\ (C^* = C^*(x)) \\
 & (\forall x : T)\ (a(C^*(x)) = x) \\
 & (\forall x : CState, y : T)\ (a(assertr(x, y)) = a(x)) \\
 & (\forall x : T)\ (assertr(null, x) = null) \\
 & (\forall x : CState, y : T)\ (assertr(assertr(x, y), y) = assertr(x, y)) \\
 & (\forall x : CState,\ y, z : T) \\
 & \quad (assertr(assertr(x, y), z) = assertr(assertr(x, z), y)) \\
 & (\forall x : T)\ (r(null, x)) \\
 & (\forall x : CState,\ y, z : T)\ (r(assertr(x, y), z) \Leftrightarrow (y = z \vee r(x, z)))
\end{array}
$$

Fig. 8.6. Extending the specification $vspec(\mathtt{C_A})$.

We can enumerate all the desired states of $\mathtt{C_B}$. The intended state space can be described as the union of the initial states set

$$\{\ null, (t_1, \{\}), (t_2, \{\})\ \}$$

and the set of states

$$\{\ (t_1, \{t_1\}), (t_1, \{t_2\}), (t_1, \{t_1, t_2\}), (t_2, \{t_1\}), (t_2, \{t_2\}), (t_2, \{t_1, t_2\})\ \}$$

generated from the initial states via the *assertr* constructor. The intention is that the undefined state *null* in $vspec(\mathtt{C_A})$ corresponds to the undefined state in $vspec(\mathtt{C_B})$

and that the state denoted by the constant C^* corresponds to any one of the initial states of $\mathtt{C_B}$ listed above. This condition is made explicit by the axiom

$$(\exists! x : T)\ (C^* = C^*(x))\ .$$

8.5.1. *State Models*

We can make the following observations about the relation between the state space models of class `C` in frameworks `A` and `B`. Let $\mathcal{A}$ and $\mathcal{B}$ be models of *vspec*($\mathtt{C_A}$) and *vspec*($\mathtt{C_B}$) respectively, then,

(1) $CState^{\mathcal{A}} \subseteq CState^{\mathcal{B}}$;
(2) $null^{\mathcal{A}} = null^{\mathcal{B}}$; and
(3) $C^{\mathcal{A}} = C^{\mathcal{B}}$.

In fact, $CState^{\mathcal{A}} \subset CState^{\mathcal{B}}$ since, in framework `B` the state space *CState* is extended with new elements. In the general case, we make the following observations.

Let `C` be a class in framework `M` and let `C` be extended in a framework `M'`. Thus, *vspec*($\mathtt{C_M}$) and *vspec*($\mathtt{C_{M'}}$) are object value specifications for `C` in `M` and `M'` respectively with models $\mathcal{M}$ and $\mathcal{M}'$. Then,

(1) $s^{\mathcal{M}} \subseteq s^{\mathcal{M}'}$ for each sort s in *vspec*($\mathtt{C_M}$).
(2) $c_s^{\mathcal{M}} = c_s^{\mathcal{M}'}$ for each constant c of sort s.
(3) $f^{\mathcal{M}}(x_1, \ldots, x_m) = f^{\mathcal{M}'}(x_1, \ldots, x_m)$
for all functions $f : [s_1, \ldots, s_m] \to s$ in *vspec*($\mathtt{C_M}$),
where $x_i : s_i^{\mathcal{M}}$ for $1 \leq i \leq m$.
$f^{\mathcal{M}} = f^{\mathcal{M}'}|(s_i^{\mathcal{M}})^m$,
i.e., each m-placed function $f^{\mathcal{M}}$ of $\mathcal{M}$ is the restriction to $dom(f^{\mathcal{M}})$ of the corresponding function $f^{\mathcal{M}'}$ of $\mathcal{M}'$.
(4) $p^{\mathcal{M}}(x_1, \ldots, x_n)$ iff $p^{\mathcal{M}'}(x_1, \ldots, x_n)$ for all predicates $p : [s_1, \ldots, s_n]$ in *vspec*($\mathtt{C_M}$),
where $x_i : s_i^{\mathcal{M}}$ for $1 \leq i \leq n$.

That is, $\mathcal{M}$ is a submodel of $\mathcal{M}'$.

The term *extend* is somewhat of a misnomer in our specifications: strictly speaking, *vspec*($\mathtt{C_B}$) does not extend *vspec*($\mathtt{C_A}$). We take extension to mean that the initial semantics properties of *no junk* and *no confusion* are preserved whenever new sorts and function and predicate symbols are added. A more appropriate term is *enlargement* [13]. Enlargement is similar to the *extending* mode of import used in OBJ3 [14], which preserves the no confusion property across module imports.

8.5.2. *Class Extension and Composition*

Extension. If `A` and `B` are two classes such that *vspec*(`A`) $= \langle \Sigma_A, \Phi_A \rangle$ and *vspec*(`B`), then "`B extends A` $\{\ldots\}$" corresponds to

$$vspec(\mathtt{B}) = \langle \Sigma_A \cup \Sigma_\delta, \Phi_A \cup \Phi_\delta \rangle\ .$$

As described in the example Σ_δ and Φ_δ emerge as a result of taking into consideration the new class definition `B`. New constructors are introduced to enlarge the state

space of A and function/predicate symbols are introduced for each newly declared feature symbols of B. The set Φ_δ consists of the axioms that can be generated according to the new definition of B—the axioms extend the interpretation of features over the newly introduced states. In addition, Φ_δ includes the equality axioms for constraining how A^*-generated states are mapped to equivalent B^*-generated states.

Composition. The case for class composition is similar. If $\mathtt{C_1}$ and $\mathtt{C_2}$ are classes such that $vspec(\mathtt{C_1}) = \langle \Sigma_{C_1}, \Phi_{C_1} \rangle$ and $vspec(\mathtt{C_2}) = \langle \Sigma_{C_2}, \Phi_{C_2} \rangle$ and $\oplus$ is the implied class composition operator, then the composition

$$\mathtt{C} = \mathtt{C_1} \oplus C_2 = \mathtt{C_2} \oplus \mathtt{C_1}$$

means

$$vspec(\mathtt{C}) = \langle \Sigma_{C_1} \cup \Sigma_{C_2} \cup \Sigma_\delta, \Phi_{C_1} \cup \Phi_{C_2} \cup \Phi_\delta \rangle \ ,$$

the union of $vspec(\mathtt{C_1})$ and $vspec(\mathtt{C_2})$, followed by an extension by $\langle \Sigma_\delta, \Phi_\delta \rangle$. As with extension, Σ_δ and Φ_δ arise from the generation of new state constructors. However, whereas extension always results in the addition of $\langle \Sigma_\delta, \Phi_\delta \rangle$, the composition of $\mathtt{C_1}$ and $\mathtt{C_2}$ may result in

$$vspec(\mathtt{C}) = \langle \Sigma_{C_1} \cup \Sigma_{C_2}, \Phi_{C_1} \cup \Phi_{C_2} \rangle \ .$$

This situation can arise when $\mathtt{C_1} = \mathtt{C_2}$. Note that in the case of $\mathtt{C_1} \oplus \mathtt{C_2} = \mathtt{C_2}$ ($\mathtt{C_1} \neq \mathtt{C_2}$) additional axioms Φ_δ are still required to equate C_1^*-generated states to a subset of C_2^*-generated states.

Framework Extension and Composition. The idiosyncrasies associated with class extension and composition are reflected at the framework level. For framework extension we have

$$spec(\mathtt{G}) = \langle \Sigma_F \cup \Sigma_\Delta, \Phi_F \cup \Phi_\Delta \rangle \ ,$$

where

$$\Sigma_\Delta = \Sigma_\gamma \cup \bigcup_i \Sigma_{\delta_i} \qquad \text{and} \qquad \Phi_\Delta = \Phi_\gamma \cup \bigcup_i \Phi_{\delta_i} \ .$$

The extension of each class $\mathtt{C_i}$ introduces the extensions $\langle \Sigma_{\delta_i}, \Phi_{\delta_i} \rangle$ as described above. The extension of a framework with new event declarations and new constraints is reflected by the introduction of Σ_γ and Φ_γ respectively. Here, Σ_γ consists purely of the new event predicate symbols introduced in G whereas Φ_γ may consist of (additional) rules specifying the effects of already declared events or new constraints. In the case of pure composition we have

$$spec(\mathtt{F}) = \langle \Sigma_{F_1} \cup \Sigma_{F_2} \cup \Sigma_\Delta, \Phi_{F_1} \cup \Phi_{F_2} \cup \Phi_\Delta \rangle \ .$$

This time Σ_Δ and Φ_Δ are defined by

$$\Sigma_\Delta = \bigcup_i \Sigma_{\delta_i} \qquad \text{and} \qquad \Phi_\Delta = \bigcup_i \Phi_{\delta_i} \ .$$

8.6. Behavioural Modelling and Specification

For dynamic properties of frameworks we need to consider temporal versions of the state assignment functions described earlier—*state history functions.* When considering temporal aspects, we assume time is discrete and that there is a global (synchronous) time scale valid across frameworks, i.e., objects do not have local time. The use of time was illustrated in Fig. 8.5. in the definition of the fact `oneOrMoreEmployees` and the pre- and postcondition definition for the event `employ` in `Drivers+Employees`.

8.6.1. *State History Functions*

For each class $\mathtt{C_i}$, the framework specification contains a function

$$\$: [C_i, \mathit{Time}] \rightarrow C_i\mathit{State} \ .$$

The sort *Time* acts as an abstract time axis against which observations on framework state are made. The definition of time is assumed to reside in the aforementioned `DataTypes` framework. Here, *Time* is interpreted as the natural numbers with constant 0, successor function *next*: $[\mathit{Time}] \rightarrow \mathit{Time}$ and ordering relation $<$. The term $\$(x, t)$, where $x : C_i$ and $t : \mathit{Time}$, denotes the state of the $\mathtt{C_i}$-object named x at the tth observable moment of the framework and $\$(x, \mathit{next}(t))$ the next observable state of x as was illustrated previously.

The issue of object existence was raised when we considered instance diagrams in Sec. 8.4.4.. The scheme can be extended to state histories: an object x exists at time t if and only if $\$(x, t) \neq \mathit{null}$. From this we can go on to define the usual notions of existence sets of objects and populations of objects within an OO system. In the framework of FML, these correspond to temporal variations of OCL's allInstances operator.

8.6.2. *Events*

In FML, both the internal state of objects and the history of events are noted. On the one hand, state sequences allow us to talk about how the states of objects may change over time while timed events allow us to describe simple interaction protocols and the effects of actions on the states of objects. The motivation for having both state sequences and externally observable events is that some state transitions are silent—they may occur as a result of some unknown event occurring, i.e., the occurrence of an external action. Consequently, we do not consider the specification of *locality axioms* [15], which restrict what actions can modify an object's state.

The approach adopted in this formalisation of Catalysis frameworks is based on *non-reified* temporal logic, i.e., where events and state valuation functions are explicitly augmented with a time parameter. In Fig. 8.5. the declaration of the event `employ` illustrates the essence of the approach. This differs from *reified* approaches in which explicit event sorts and event predicates are introduced.

8.6.3. *Constraints in FML*

In Sec. 2, we were interested in three kinds of constraints: invariants, expressed as *facts* in FML; pre- and postconditions, which in FML, are attached to event declarations; and external effect invariants. According to the Catalysis definition, external effect invariants may be thought of as conditions that must be checked whenever a framework is imported into others. These are expressed in a similar manner to facts but introduced by the keyword `assert`. The translations of invariants and these assertions into first-order logic is straightforward. As we saw earlier, the invariants of FML are expressed as formulae in first-order logic but using FML's object-based syntax. The translation converts object terms/formulae into non-object-based terms/formulae. Thus, a term `$(x,t).f` is rewritten as the term $f(\$(x,t))$ and the formula `$(x,t).g(y)` is rewritten as $g(\$(x,t),y)$.

8.6.3.1. *Invariants*

From the preceding sections it can be seen that we have a choice when it comes to constraining the state of an object. The first method is to address the state space of a class directly. That is, we can assert that all states of class C satisfy a given property P, i.e.,

$$(\forall x : \mathit{CState})\ P(x)\ ,$$

or, in FML, by the fact

```
fact { all x: CState:: P(x) } .
```

The alternative is to constrain which states may be assigned to objects. In this case the property P is asserted to hold over assigned states:

$$(\forall x : C, t : \mathit{Time})\ P(\$(x,t))\ ,$$

expressed by

```
fact { all x: C, t: Time:: P($(x,t)) } .
```

The former approach requires that the sort *CState* is visible to the modeller. In the latter case, the sort is implicit and hidden. In FML, *CState* is considered a hidden sort—individual states may only be referenced using object terms of the form $\$(x,t)$.

Implicit Properties. It should be noted that there are a number of intrinsic properties applicable to all objects/classes. These properties are defined as axioms of the specification *spec*(`F`). An example of one such property is the persistence of objects: once an object becomes active or is created, it is not destroyed. For each class $\mathtt{C_i}$, the following axiom can be generated to assert this:

$$(\forall x : C_i, t : \mathit{Time})\ (\$(x,t) \neq \mathit{null} \Rightarrow \$(x, \mathit{next}(t)) \neq \mathit{null})\ .$$

This constraint ensures that an event does not cause an object to move from a defined stated to an undefined state.

8.6.3.2. *Events*

Let `m` be a timed event and $\langle \mathtt{m_{pre}}, \mathtt{m_{post}} \rangle$ denote the pre- and postcondition pair attached to `m`. Then, the event axiom

$$(\forall \vec{x})\ (m(\vec{x}) \wedge m_{\text{pre}}(\vec{x}) \Rightarrow m_{\text{post}}(\vec{x})))$$

specifies the intension of `m`.

In any interpretation of a framework, the extension of m gives its event history, describing when `m` occurs and which objects were involved in each occurrence. Underlying the event is the assumption that any object participating in it is one that exists. Thus, if $\vec{x} = (x_1, \ldots, x_n, t)$ and $x_i : C_i$, then the event `m` would have a guard

$$(\forall \vec{x})\ \neg(m(\vec{x}) \wedge \bigwedge_{i=1}^{n} \$(x_i, t) = \mathit{null})\ .$$

For example, the `employ` event would be described by the axiom

$$\begin{aligned}&(\forall c : \mathit{Company}, p : \mathit{Person}, t : \mathit{Time})\\ &\quad (\mathit{employ}(c, p, t) \wedge \neg \mathit{mem}(p, \mathit{employee}(\$(c, t)))\\ &\qquad \Rightarrow \mathit{employee}(\$(c, \mathit{next}(t))) = \mathit{add}(p, \mathit{employee}(\$(c, t))))\end{aligned}$$

subject to the guard

$$\begin{aligned}&(\forall c : \mathit{Company}, p : \mathit{Person}, t : \mathit{Time})\\ &\quad \neg(\mathit{employ}(c, p, t) \wedge \$(c, t) = \mathit{null} \wedge \$(p, t) = \mathit{null})\ .\end{aligned}$$

8.6.3.3. *External Effect Invariants*

One of the main features of Catalysis is the ability to choose any level of abstraction when modelling frameworks. This is reflected in the different ways in which a constraint may be expressed. For example, in the *Observer* pattern, the synchronisation of data between two objects may either be expressed as a static invariant between subject and observer, or using behavioural contracts. These contracts may be expressed in FML as facts.

In contrast, external effect invariants act as additional assertions that must be checked whenever a framework is imported into others. If `F` and `G` are frameworks, where $\mathtt{m_G}$ is an event in `G`, then the presence of an external effect invariant $\mathtt{E_F}$ in `F` would require that

$$I_{F+G} \wedge (m_G \wedge P_G \Rightarrow Q_G) \wedge E_F$$

holds whenever `F` and `G` are composed. In reality, there is little to distinguish effect invariants from invariants. Hence, in the sequel we do not concern ourselves with *external* effect invariants.

8.7. Framework Consistency

Intuitively, a framework specification $spec(\mathtt{F}) = \langle \Sigma_F, \Phi_F \rangle$ is *consistent* if the axioms Φ_F are not in some way contradictory. The danger of basing framework composition on union rather than disjoint union is that specifications of the same class in different frameworks may not be compatible. Catalysis defines two types of composition to address these issues, depending on the application of the framework.

Firstly, a framework may define one of several slices of behaviour exhibited by a system. Classes are partially defined and, likewise, operations may also be partially defined. This corresponds to the idea in UML that complex constraints may be decomposed into smaller ones or that multiple constraints may be combined into single statements. In Catalysis the composition associated with this is known as *joining*. Secondly, the individual functionality of components may need preserving. Taking the *intersection* of classes ensures that the actions of composite classes adhere to mutually exclusive constraints. As we will see both types of composition differ from the standard notion of subcontracting.

8.7.1. *Contract Composition in Catalysis*

Join. The join of n pre- and postcondition pairs $\langle P_i, Q_i \rangle$ $(1 \leq i \leq n)$ for an event m is given by the pair formed by

$$\langle P_1 \wedge \ldots \wedge P_n, Q_1 \wedge \ldots \wedge Q_n \rangle \; .$$

This is not the only way of joining action specifications in Catalysis or in UML tools such as the KeY system [16]. An alternative way is to factor the preconditions of each specification m into their postconditions, i.e., to derive the specification

$$\langle \; \mathbf{true}, \; (P_1 \Rightarrow Q_1) \wedge \ldots \wedge (P_n \Rightarrow Q_n) \; \rangle \; .$$

From this a resultant precondition can be computed to derive

$$\langle \; P_1 \vee \ldots \vee P_n, \; (P_1 \Rightarrow Q_1) \wedge \ldots \wedge (P_n \Rightarrow Q_n) \; \rangle \; .$$

The reader is referred to Hennicker *et al.* [17] for a discussion on the semantics of each kind of contract composition.

Intersection. The difference between joins and intersections lies in the fact that, for intersections, the invariants I_i and operations specifications $\langle P_i, Q_i \rangle$ for each class $\mathtt{C_i}$ are assumed to be mutually inconsistent but nonetheless the composition of contracts should still be allowed. Intersection may result in the *refactoring* of a design. To avoid the problem of inconsistency between the invariants I_i, the invariants must be factored into the method specifications as follows:

$$\left\langle \; \bigvee_i (P_i \wedge Q_i), \; \bigwedge_i (I_i \wedge P_i \Rightarrow Q_i) \right\rangle \; ,$$

i.e., it involves a retraction of axioms.

8.7.2. *Contracts in FML*

It can be seen that the notions of join and intersection differs from the principle of subcontracting in *Design by Contract* [18], where preconditions are weakened and postconditions strengthened:

$$\langle P_1 \vee \ldots \vee P_n, Q_1 \wedge \ldots \wedge Q_n \rangle \ .$$

When class extension occurs there are a couple of ways we can deal with the refinement of an event specification. We could allow an extending class to provide an alternative axiom to describe the intention of the event. Following the subcontracting principle, we could replace the event axioms $m \wedge P_i \Rightarrow Q_i$ by a single subcontracting-compliant axiom

$$m \wedge (P_1 \vee \ldots \vee P_n) \Rightarrow Q_1 \wedge \ldots \wedge Q_n \ .$$

Instead we leave the set of imported axioms untouched, preserving the axiom set

$$\{m \wedge P_1 \Rightarrow Q_1, \ldots, m \wedge P_n \Rightarrow Q_n\}$$

This coincides with joining, i.e., we can show that the join

$$m \wedge (P_1 \vee \ldots \vee P_n) \Rightarrow (m \wedge P_1 \Rightarrow Q_1) \wedge \ldots \wedge (m \wedge P_n \Rightarrow Q_n)$$

is a logical consequence of the above axiom set.

8.7.3. *Consistency Checking*

Consistency of a specification implies that there is some model which satisfies the specification. Tools such as Alcoa [19] typically check constraints in two ways: (i) by exercising invariants or operations, attempting to find satisfying states and transitions respectively; or (ii) by checking that some well-known property of a system is a logical consequence of the object model constraints. Our aim is to check that there are models which satisfy the axioms of a framework, i.e., ensuring that the invariants (axioms) are not so strong that they rule out any satisfying states and ensuring that the resulting states from events are reachable.

One source of difficulty (and complexity) in consistency checking in specifications derived from FML models is the manner in which invariants and operations are specified. A lack of distinction between local object invariants and global invariants makes it difficult to direct the theorem proving process on specific parts of the model. Another source of complexity arises from the use of time in specifications. As we saw earlier, many generated axioms are required to specify intrinsic properties of frameworks, e.g., persistence constraints. For theorem proving purposes, we can seek to reduce the number of axioms we need to deal with by examining which parts of a specification are unnecessary for invariant and operation checking. In the remainder of this section we will identify ways in which the complexity of a specification *spec*(`F`) may be reduced for theorem proving. The resulting specification may be processed by any number of theorem provers, e.g., the tableaux-based 3TAP [20], or the resolution-based OTTER [21]. In the latter case, the additional

step of reducing the many-sorted specification into a single-sorted specification [22] is required.

8.7.3.1. *(Static) Invariants*

The decision to consider the sort *CState* as hidden means that a static invariant in OCL, for example, would be formulated as a temporal invariant in FML. Constraints on a state history function $ indirectly restricts which states in *CState* are applicable. We can view the constraint

$$(\forall x : C, t : \mathit{Time})\ (a(\$(x,t)) > 0)$$

as being a weaker form of the static constraint

$$(\forall x' : \mathit{CState})\ (a(x') > 0)\ .$$

That is, temporal constraints may be rewritten into static constraints. For any valid object x' we should be able to make a query

$$(\exists x' : \mathit{CState})\ (a(x') > 0 \wedge x' \neq \mathit{null})\ ,$$

i.e., we want to see whether we can construct a state for which $a(x') > 0$ and one that happens not to be the undefined state. The condition that the state is undefined is important for checking that actions do result in what we intuitively consider as valid states. These assumptions are made explicit by the introduction of persistence axioms and the event-guard axioms mentioned earlier.

8.7.3.2. *Temporal Invariants and Events*

The situation for handling temporal invariants that contain references to next-states and events are similar. Given a temporal invariant $P \Rightarrow Q$, we wish to ensure that from all states satisfying a property P there exists a state satisfying Q. The persistence constraint is one such example. The statement may be recast: from all defined states, there exists a state which is also defined, resulting in

$$(\forall x' : \mathit{CState})\ (x' \neq \mathit{null} \Rightarrow \delta(x') \neq \mathit{null})\ .$$

As before, a term $\$(x,t)$ is replaced by the state variable x'. In addition, for each x' the term $\$(x, \mathit{next}(t))$ may be replaced by a term $\delta(x')$. The consequence of these rewrites is that the datatype *Time* may be eliminated from the framework specification.

An observation that can be made from the above is that event-guard axioms may also be discarded. The reason is that we wish to consider the effects of operations over *valid* (non-*null*) object states. Consequently, we can go further and discard the constant *null* and its related axioms.

8.7.3.3. *Flat Specifications vs Structured Specifications*

A final consideration is that the flat specification *flatspec*(`F`) of a framework can be used as the basis for consistency checking as opposed to the structured specification

spec(F). The flat specification for F is obtained from applying the *flatten* procedure to F, and then generating a specification from the flattened model.

From the discussion of structuring and modularity at the class level in Sec. 8.5. it can be seen that in structured specifications some function symbols and axioms become redundant whenever classes are extended. These 'legacy' function symbols and axioms arise as a result of enlarging the state space of objects. As new state generators are added existing constructors become redundant: if C_1 is extended by C_2 any C_1-state reachable using the constructor C_1^* is also reachable using the constructor C_2^*. The introduction of the constructor C_2^* also brings about the introduction of new axioms ranging over C_2^*-generated states. These axioms describe the evaluation of features over C_2-states. However, these are a superset of the existing C_1-states. Consequently, the equations introduced for each function over the states of C_1 may also be discarded. Flat specifications have the property that they consist of only symbols and axioms for the class C_2. Axioms are not generated for C_1 since the fact that C_2 is derived from C_1 is discarded when unfolding takes place. That is, *flatspec*(F) is a sub-specification of *spec*(F).

8.8. Frameworks in Component Modelling

In this section, we consider how the concepts of *framework* and *component* are related by means of an example of a production planning system (PPS), adapted from Rausch's article on *design by signed contract* [23] for componentware. The goal of the PPS is to optimise the scheduling of jobs to robots. The operation of robots is constrained in the following ways:

(1) each robot may process only one job at a time; and
(2) no two robots may be assigned the same job.

The first condition is violated if a robot is assigned two jobs whose scheduled times overlap.

The PPS itself is modelled as a component-based system, constructed from two subcomponents: a scheduler and a robot component. This can be expressed using a UML component diagram, as illustrated in Fig. 8.7.. Each component has a provided and required interface. The components are defined such that the provided interface of one satisfies the required interface of the other.

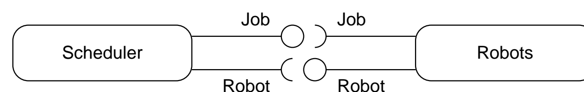

Fig. 8.7. Component diagram showing how the production planning system is assembled from subcomponents.

A framework representing the scheduling component is shown in Fig. 8.8.. However, unlike the Scheduler component in Fig. 8.7., the framework does not distinguish between the provided and required services.

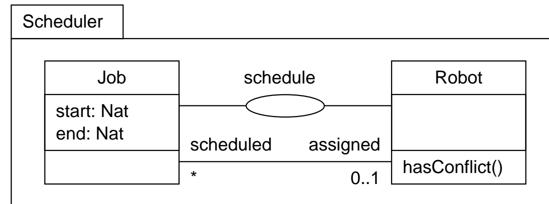

Fig. 8.8. A framework representing a job scheduling component.

The Robots component is similar to Scheduler but describes only the static relationships between jobs and robots. It differs from Scheduler in the definition of robots, in which case it adds a (derivable) attribute duration, and the absence of the schedule action.

Figure 8.9. shows the FML definition for the component Scheduler. In particular, the fact labelled `noConflicts` expresses the condition that a robot may not be assigned two jobs whose scheduled times overlap. The event `schedule`, corresponding to an assignment of a job to a robot, should maintain this invariant. An event `hasConflict` is signalled whenever the invariant is violated. Thus, a valid model for this framework is one in which `hasConflict` does not occur.

The robot handling component, `Robots`, is shown in Fig. 8.10.. The fact `conflictGuard` provides is an alternative way of expressing the invariant `noConflict` in `Scheduler`. It states that the `hasConflict` property should not become true whenever there is a change in a robot's state. This is an effect invariant but not an external one as it applies equally to interactions within `Robots` and to `Scheduler`.

Implicit Properties. In the framework `Scheduler`, an intrinsic property is that every object referenced in the collections `scheduled` and `assigned` are valid objects. That is, we have the axiom

$$(\forall r : Robot, j : Job, t : Time)\ \neg(mem(j, scheduled(\$(r,t)))\ \wedge\ \$(j,t) = null)$$

for the feature `scheduled` and a similar axiom is generated for `assigned`. Like persistence axioms and event-guard axioms these axioms may be discarded before the theorem proving begins.

8.8.1. *Signed Contracts and Composition*

The *signed contract* between two components is a user's specification of the syntactic and behavioural mappings between them. It specifies how the required interface of one component is satisfied by the provided interface of another. The intention of signed contracts is to enable users or developers to check whether all required properties of a component are fulfilled by another component. Signed contracts are similar to specification fitting morphisms for instantiating parameterised algebraic specifications.

```
framework Scheduler
import Sets[Data\Job]
import Sets[Data\Robot]

class Robot { scheduled: -> Set(Job)
}
class Job { assigned: -> Set(Robot), start: -> Nat, end: -> Nat
}
fact IntervalNonNegative {
    all j: Job, t: Time:: $(j,t).start < $(j,t).end
}
fact { all j: Job, r1,r2: Robot, t: Time::
    mem(r1,$(j,t).assigned) & mem(r2,$(j,t).assigned)
      ==> r1=r2
}
fact noConflicts { all r: Robot, j1,j2: Job, t: Time::
    j1!=j2 & mem(j1,$(r,t).scheduled) & mem(j2,$(r,t).scheduled)
  & $(j1,t).start<=$(j2,t).start
  ==> $(j1,t).end<=$(j2,t).start
}
fact { all r: Robot, t: Time:: hasConflict(r,t)
  <==> exists j1,j2: Job:: j1!=j2
     & mem(j1,$(r,t).scheduled) & mem(j2,$(r,t).scheduled)
     & $(j1,t).start<=$(j2,t).end & $(j2,t).start<=$(j1,t).end
}
event hasConflict(r: Robot, t: Time)

event schedule(j1: Job, r: Robot, t: Time) {
  decls: all j2: Job
  pre:   empty($(j1,t).assigned) & j1!=j2 & mem(j2,$(r,t).assigned)
       & (($(j1,t).start<=$(j2,t).end & $(j2,t).start<=$(j1,t).end)
          or
          ($(j2,t).start<=$(j1,t).end & $(j1,t).start<=$(j2,t).end))
  post:  $(j1,next(t)).assigned=add(r,$(j1,t).assigned)
       & $(r,next(t)).scheduled=add(j1,$(r,t).scheduled)
}
```

Fig. 8.9. The `Scheduler` component.

In the following, to differentiate one class from another, elements in `Scheduler` will be subscripted by `S` and those in `Robots` by `R`. We wish to map the class $\texttt{Robot}_{\texttt{R}}$ to $\texttt{Robot}_{\texttt{S}}$ and $\texttt{Job}_{\texttt{S}}$ to $\texttt{Job}_{\texttt{R}}$. In frameworks the syntactic mapping is straightforward. In general, this can be achieved using the renaming mechanism of FML whenever the names of required components differs from that of the provided component. However, in addition to syntactic mappings, the signed contract may specify behavioural mappings between components.

In the previous section, the discussion on consistency checking focused on ensuring that there exist valid object states satisfying the framework axioms. This is

```
framework Robots
import Sets[Data\Job]
import Sets[Data\Robot]

class Job { assigned: -> Set(Robot),
            start: -> Nat, end: -> Nat, duration: -> Nat
}
class Robot { scheduled: -> Set(Job)
}
fact nIntervalNonNegative {
    all j: Job, t: Time:: $(j,t).start < $(j,t).end
}
fact { all j: Job, t: Time::
    $(j,t).start + $(j,t).duration = $(j,t).end
}
fact { all j: Job, r1,r2: Robot, t: Time::
    mem(r1,$(j,t).assigned) & mem(r2,$(j,t).assigned) ==> r1=r2
}
fact { all r: Robot, t: Time:: hasConflict(r,t)
      <==> exists j1,j2: Job:: j1!=j2
         & mem(j1,$(r,t).scheduled) & mem(j2,$(r,t).scheduled)
         & $(j1,t).start<=$(j2,t).end & $(j2,t).start<=$(j1,t).end
}
fact conflictGuard { all r: Robot, t: Time::
    $(r,t)!=$(r,next(t)) ==> !hasConflict(r,next(t))
}
event hasConflict(r: Robot, t: Time)
```

Fig. 8.10. The Robots component in FML.

sufficient for checking the consistency of the operation `schedule` against the invariants in `Scheduler` and the temporal invariant

```
fact { all r: Robot, t: Time::
   $(r,t)!=$(r,next(t)) ==> !hasConflict(r,next(t))
```

in `Robots`. In Rausch's signed contracts, however, a mapping between the constraints in one component and those in another may also be specified. For example, if `Robots` *requires* jobs to have the property `nIntervalNonNegative` and `Scheduler` *provides* jobs with this property (`IntervalNonNegative`), then this behavioural dependency may be expressed using signed contracts. In FML, the precise way in which one property should be matched with another should be made using explicit assertions. Thus, we have

```
assert { IntervalNonNegative ==> nIntervalNonNegative }
```

to represent one mapping. Such assertions are required for event matching. Signed contracts follow the subcontracting principle: iven the pre- and postcondition pair $\langle P_{\rm L}, Q_{\rm L}\rangle$ of a provided operation L and the pre- and postcondition pair $\langle P_{\rm R}, Q_{\rm R}\rangle$ of a

required operation R, the conditions $P_\mathrm{L} \Rightarrow P_\mathrm{R}$ and $Q_\mathrm{R} \Rightarrow Q_\mathrm{L}$ should hold. However, not all events in FML are defined using pre- and postcondition pairs. The event `hasConflict` is one such example. In both frameworks the event is defined using a fact and a suitable condition must be formulated to test whether $\mathtt{hasConflict}_\mathtt{R}$ is a suitable match for $\mathtt{hasConflict}_\mathtt{S}$:

```
assert { all r: Robot, t: Time::
  Robots::hasConflict(r,t) ==> Scheduler::hasConflict(r,t)
} .
```

8.9. Related Work

The formalisation of object-based (or, in the wider context, object-oriented) systems has been studied extensively within different disciplines of computer science. In systems modelling, much effort has been placed on giving a formal semantics for the different kinds of diagrams used in UML. Prominent among these is the work of Richters [24,25], which is concerned with the formalisation of OCL with respect to a subset of UML's class diagrams. This work may be used as a basis for the validation of UML models and OCL constraints. The aforementioned KeY tool [16] may also be used to check the consistency of UML class diagrams but differs in its formalisation approach. Class diagrams and *static* OCL constraints are translated into first-order predicate logic [26]—a similar transformation might defined to translate OCL (pre version 2.0) constraints into FML; OCL pre- and postconditions specifications are translated into dynamic logic [27]. Dynamic logic has been used elsewhere, firstly to examine formal foundations for conceptual modelling [28], and secondly to examine the way in which *dynamic classes* can be used to model the roles of objects [29], supporting the notion that objects may switch between different behavioural roles at different times.

Temporal logic has also been applied extensively. For example, OCL expressions have been translated into the temporal logic Botl [30]. The full expressivity of Botl, however, is not exploited because of the lack of temporal operators in OCL. Increasing the temporal expressiveness of OCL has been a subject of great interest. There have been numerous proposals for the incorporation of time in OCL, e.g., by Hamie *et al.* [31] and Sendall and Strohmeier [32], either by defining temporal operators for OCL or adding explicit notions of discrete or real time for timing constraints. The need for a temporal OCL for modelling business components has been discussed by Conrad and Turowski [33]. Temporal constraints can be represented in UML's state charts but the definitions of UML/OCL limit what kind of temporal properties can be specified, hence the need for a proprietary OCL. Similarly, Ziemann and Gogolla describe their own version of OCL, *TOCL* (*Temporal OCL*) [34], which extends OCL with temporal operators and adapts existing OCL operators to a temporal context and Bradfield *et al.* [35] define a template-based approach for specifying the temporal properties.

Increasing the expressivity of OCL, however, does not address the issue of a lack of integration between the different views—*static*, *behavioural*, *interactive*, and

functional—of an object model. The issue has been addressed partially by the integration of class diagrams and sequence diagrams [36–38]. Different views of a model may introduce different kinds of constraints (e.g., timing constraints in sequence diagrams) without having to increase the expressivity of the constraint language. However, Conrad and Turowski's observations apply even to the interaction view in the integrated approach.

In many of the above approaches, the object-oriented aspects of models are lost in their formalisation. Objects have a shared vocabulary for their attributes/associations and operations: in essence, local object features are projected onto a global context. This differs from class-as-template approaches, where classes are templates from which objects derive their own unique vocabulary ([39] - [41]). The idea has been applied to Catalysis frameworks, exploring the formal semantics for frameworks and interaction [42] and embedding this static semantics into a temporal setting using modular distributed temporal logic (MDTL) [43]. Static semantics for frameworks are limited in the kinds of interactions that may be expressed: interactions occur via framework parameters. MDTL alters the situation and allows the specification of effect invariants while also allowing the specification of synchronous and asynchronous communication between objects from different frameworks.

The formalisation of a class described in this chapter differs from the above approaches in one common aspect: the way in which object state is treated. In this text, there is an explicit specification of a class' state space. In model-based approaches this state space is similar to taking the Cartesian product of the attribute types of a class and augmenting this set with a special undefined element. Object value specifications are similar in style to formalising object state using algebraic specifications [44] where, with the addition of ordered sorts, it is possible to use subsorts as a mechanism for partitioning state spaces.

Much of the work on formalising UML/OCL has concentrated on class diagrams. At the same time, the aforementioned shift toward increasing the expressivity of OCL for dynamic constraints (i.e., action clauses [45,46] in OCL 2.0) has been realised to some degree. Indeed, Catalysis goes further and allows the specification of how actions are invoked and sequenced within constraints. In contrast, FML is less expressive owing to its simplified event model based on a synchronous time framework. Consequently, FML is restricted compared to OCL 2.0 or MDTL when it comes to specifying the interactions between objects.

8.10. Summary

This chapter has been concerned with how model frameworks in Catalysis, which makes use of extensions and adaptations to UML/OCL, may be formalised. There are many approaches to formalising object-based or object-oriented systems, some of which have been discussed. In this chapter first-order logic is used as the formal foundation for frameworks.

From the FML definition of a framework `F`, we have looked at how a specification of `F`, *spec*(`F`), can be derived. Contained within this specification are the

specifications of classes and object states. A semantics for a class `C` is given by its associated object value specification *vspec*(`C`), which enumerates the state space of `C`. This notion of explicitly enumerating object states influences the way in which framework composition and extension is defined.

The composition of two frameworks `F` and `G` is defined if the axioms in the resulting framework `F+G` are consistent, which can be verified by a theorem prover. There are ways in which the specification *spec*(`F+G`) can be reduced prior to theorem proving: symbols and axioms which become redundant in *spec*(`F+G`) may be identified and discarded. Not all redundant formulae are discarded. Typically, one framework may strengthen the constraints of another but nonetheless these weaker constraints are retained in framework specifications.

It has been argued that the specification of software contracts is not enough for componentware. The notion of *design by signed contract* extends Design by Contract with the ability to specify the syntactic and behavioural mappings between the provided services of one component and the required services of another. This notion has been examined in the context of frameworks. Syntactic mappings may be identified during importing and renaming may be used to ensure that provided components are mapped to required components. The signed contract may also identify behavioural mappings between components. These are equivalently represented using assertions in FML.

Acknowledgements

The first author is indebted to the Engineering and Physical Sciences Research Council, without whose support this work would not have been possible.

Bibliography

1. R.E. Johnson. Frameworks = (Components+Patterns). *Communications of the ACM*, 40(10): 39–42, 1997.
2. G. Larsen. Designing Component-Based Frameworks Using Patterns in UML. *Communications of the ACM*, 42(10): 38–45, 1999.
3. D. D'Souza and A. Wills. *Objects, Components, and Frameworks with UML*. Addison-Wesley, 1998.
4. OMG Unified Modeling Language Specification, Version 1.5 (Draft), March 2003.
5. J. Warmer and A. Kleppe. *The Object Constraint Language: Precise Modeling with UML*. Addison-Wesley, 1999.
6. H.-E. Eriksson and M. Penker. *Business Modeling with UML, Business Patterns at Work*. Wiley, 2000.
7. T. Reenskaug *et al. Working with Objects*. Manning/Prentice-Hall, 1995.
8. T. Clark, A. Evans, and S. Kent. A Metamodel for Package Extension with Renaming. In J.-M. Jezequel, H. Hussman, and S. Cook, editors, *Proc. UML 2002*, *LNCS* 2460, pages 305–320, Springer-Verlag, 2002.
9. R. Helm, I. Holland and D. Gangopadhyay. Contracts: Specifying Behavioural Compositions in Object-Oriented Systems. In *Proc. OOPSLA*, 169–180, October 1990.
10. M. Vaziri and D. Jackson. Some Shortcomings of OCL, the Object Constraint Con-

straint Language of UML. Technical report, Massachusetts Institute of Technology, December 1999.
11. D. Jackson. Micromodels of Software: Lightweight Modelling and Analysis with Alloy (Draft). Technical report, Massachusetts Institute of Technology, February 2002.
12. R.H. Bordeau and B.H.C. Cheng. A Formal Semantics for Object Model Diagrams. *IEEE Transactions on Software Engineering*, 21(10): 799–821, October 1995.
13. H. Ehrig, B. Mahr. *Fundamentals of Algebraic Specification 1, Equations and Initial Semantics*. Springer-Verlag, 1985.
14. J.A. Goguen, T. Winkler, J. Meseguer, K. Futatsugi and J.P. Jouannaud. Introducing OBJ. In J.A. Goguen, editor, *Applications of Algebraic Specification using OBJ*, Cambridge, 1993.
15. N. Aguirre and T. Maibaum. A Temporal Logic Approach to Component-based System Specification and Verification. In *Proc. ICSE02*, 2002.
16. W. Ahrendt *et al.* The KeY Tool. Department of Computer Science, Chalmers University and Goteborg University, Technical Report in Computer Science No. 2003–5, February 2003.
17. R. Hennicker, H. Hussmann and M. Bidoit. On the Precise Meaning of OCL Constraints. In *Object Modeling with the OCL*, *LNCS* 2263, pages 69–84, Springer-Verlag 2002.
18. B. Meyer. Design by Contract. Technical Report TR-EI-12/CO, ISE Inc., 1987.
19. D. Jackson, I. Schechter and H. Shlyakhter. Alcoa: The Alloy Constraint Analyzer. In *Proc. 22nd Intl. Conf. on Software Engineering*, pages 730–733, 2000.
20. B. Beckert, R. Hähnle, P. Oel, and M. Sulzmann. The Tableau-based Theorem Prover 3TAP, Version 4.0, In *Proc. 13th Intl. Conf. on Automated Deduction*, *LNCS* 1104, pages 303–307, Springer, 1996.
21. W.W. McCune Otter Reference Manual and Guide, Argonne National Laboratory, January, 1994.
22. H.B. Enderton. *A Mathematical Introduction to Logic*. Academic Press, New York, 1972.
23. A. Rausch. "Design by Contract" + "Componentware" = "Design by Signed Contract" In *Journal of Object Technology, Special issue: Proc. TOOLS USA 2002*, 1(3): 19–36, 2002.
24. M. Richters and M. Gogolla. OCL: Syntax, Semantics, and Tools. In *Object Modeling with the OCL*, *LNCS* 2263, pages 42–68. Springer-Verlag, 2002.
25. M. Richters. *A Precise Approach to Validating UML Models with OCL Constraints*. PhD thesis, Universität Bremen, 2002.
26. B. Beckert, U. Keller, and P.H. Schmitt. Translating the Object Constraint Language into First-order Predicate Logic. In *Proc. VERIFY, Workshop at Federated Logic Conferences*, Copenhagen, Denmark, 2002.
27. T. Baar, B. Beckert, and P.H. Schmitt. Extension of Dynamic Logic for Modelling OCL's @pre Operator. In D. Bjorner, M. Broy, and A.V. Zamulin, editors, *Proc. 4th Intl. Andrei Ershkov Memorial Conf., Perspectives of Systems Informatics*, *LNCS* 2244, pages 47–54, Springer-Verlag 2001.
28. R.J. Wieringa. A Formalisation of Objects using Equational Dynamic Logic. In C. Delobel, M. Kifer and Y. Masunag, editors, *2nd Intl. Congress on Deductive and Object-Oriented Databases* 566, pages 431–452, Springer-Verlag, 1991.
29. R.J. Wieringa, W. de Jonge, and P. Spruit. Using Dynamic Classes and Role Classes to Model Object Migration. In *Theory and Practise of Object Systems*, 1(1): 61–83, 1995.
30. D. Distenfano, J.P. Katoen, and A. Rensink. On a Temporal Logic for Object-based

Systems. In S.F. Smith and C.L. Talcott, editors, *Formal Methods for Open Object-based Distributed Systems IV*, pages 305–326. Kluwer Academic Publishers, September 2000.

31. A. Hamie, R. Mitchell, and J. Howse. Time-based Constraints in the Object Constraint Language. Technical Report CMS-00-01, University of Bristol, 2000.
32. S. Sendall and A. Strohmeier. Specifying Concurrent System Behaviour and Timing Constraints Using OCL and UML. In *Proc. UML 2001*, *LNCS* 2185, pages 391–405, Springer-Verlag 2001.
33. S. Conrad and K. Turowski. Temporal OCL: Meeting Specification Demands for Business Components. In K. Siau and T. Halpin, editors, *Unified Modeling Language: Systems Analysis, Design and Development Issues*, Chapter 10, pages 151–166, Idea Publishing Group, 2001.
34. P. Ziemann and M. Gogolla. An Extension of OCL with Temporal Logic. In *Critical Systems Development with UML—Proc. UML'02 Workshop*, pages 53–62, Technische Universität München, Institut für Informatik, 2002.
35. J. Bradfield, J. Küster-Filipe, and P. Stevens. Enriching OCL using observational mu-calculus. In R.-D. Kutsche, and H. Weber, editors, *Proc. Fundamental Approaches to Software Engineering 2002*, *LNCS* 2306, pages 203–217, Springer-Verlag 2002.
36. J. Yang, Q. Long, Z. Liu and X. Li. A Formal Semantics of UML Sequence Diagrams. In Z. Liu and K, Araky, editors, *Proc. of 1st International Colloquium on Theoretical Aspects of Computing (ICTAC 2004), Lecture Notes in Computer Science 3074*, pages 170-186, Springer, 2005.
37. X. Li, Z. Liu and J. He. A Predicative Semantic Model for Integrating UML Models. In *Proc. Australian Software Engineering Conference*, pages 168-177, IEEE Computer Society, 2004.
38. Z. Liu, J. He, X. Li and J. Liu, Unifying views of UML , *Electronic Notes in Theoretical Computer Science*, Volume 101, pages 95-127, 2004.
39. E. Amir. Object-Oriented First-Order Logic. In *Linkoping University Electronic Articles in Computer and Information Science*, ISSN 1401–9841, 4(1999): 042.
40. H.-D. Ehrich. Object Specification. In E. Astesiano, H.-J. Krewski, and B. Krieg-Bückner, editors, *Algebraic Specification*, Chapter 12, pages 435–465, Springer, 1999.
41. K.-K. Lau and M. Ornaghi. Correct Object-Oriented Systems in Computational Logic. In A. Pettorossi, editor, *Proc. LOPSTR '01*, *LNCS* 2372, pages 168–190, Springer-Verlag, 2002.
42. K.-K. Lau, S. Lui, M. Ornaghi, and A. Wills. Interacting Frameworks in *Catalysis*. In *Proc. 2nd Intl. Conf. on Formal Engineering Methods*, pages 110–119, IEEE Computer Society Press, 1998.
43. J. Küster Filipe, K.-K. Lau, M. Ornaghi, and H. Yatsu. Intra- and Inter-OOD Framework Interactions in Component-based Software Development in Computational Logic. In A. Brogi and P. Hill, editors, *Proc. 2nd Intl. Workshop on Software Development in Computational Logic*, September 1999.
44. J.A. Goguen and R. Diaconescu. Towards an Algebraic Semantics for the Object Paradigm. In *RECENT Trends in Data Type Specification: Workshop on Specification of Abstract Data Types: COMPASS: Selected Papers*, *LNCS* 785, Springer-Verlag, 1994.
45. A. Kleppe and J. Warmer. Extending OCL to Include Actions. In *Proc. UML 2000*, *LNCS* 1939, 440–450, Springer-Verlag, 2000.
46. A. Kleppe and J. Warmer. The Semantics of the OCL Action Clause. In *Object Modeling with the OCL*, *LNCS* 2263, 213–227, Springer-Verlag, 2002.