Action Reification in Object-Oriented Specifications

Extended Abstract

G. Denker, H.-D. Ehrich*

Technische Universität Braunschweig, Informatik, Abt. Datenbanken
Postfach 3329, D–38023 Braunschweig, Germany

e–mail: {denker|ehrich}@idb.cs.tu-bs.de

1 Motivation

The formal step by step development of implementations from specifications is necessary to allow incremental descriptions of large software systems and hence split the software development process in manageable portions. Due to the complex notion of objects as units of structure and behavior the refinement process has to be reconsidered in the object-oriented framework. Apart from refining structure the behavioral part gives rise to refine actions by transactions.

Thus, results about refinement from the theory of abstract data types (e.g., [Hoa72, Nip86, ST88, EGL89, EM90]) as well as from process theory (e.g., [Jif89, vGG90, Vog90, AH93]) have to be taken into consideration. Besides this our interest is the design of information systems, i.e., reactive systems with an underlying database and application programs. Therefore, reification of actions causes synchronization problems when several resulting transactions concurrently access shared data. We aim at establishing a semantical framework in which reification of actions can be defined in the most liberal way. I.e., the sequential composition of transactions which derive from reifying a sequence of abstract actions should be liberalized such that independent concrete actions, i.e., actions which are not accessing the same resources, may be interleaved arbitrarily and do not have to wait for each other. Thus, assuming a notion of independence on actions the intended class of models will incorporate all those sequences which represent correct interleavings of transactions.

We present an approach to integrate transactions in object-oriented specifications and illustrate the main problems of synchronizing them on commonly used resources. Furthermore, we provide a denotational semantics based on event structures in which the sequential composition of transactions can be appropriately liberalized and outline the ideas with some examples.

2 Liberal Sequential Composition

We will illustrate our ideas with an example specified in a TROLL [HSJ+94] similar language. On the one hand we abstracted from some TROLL specific details for simplicity reasons and on the other we introduced some ad hoc notations because of illustration reasons.

The first example specifies a Step object, which is able to change its attribute quantity by performing increasing (inc) or decreasing (dec) actions. The specified axioms (enabled, changing) establish the intended behavior of these actions, i.e., inc events increase quantity by one and dec events, which are only enabled if the quantity attribute is greater than zero, decrease it by one. Moreover, two actions for creating and destroying Step objects are specified.

*Work reported here was partially supported by CEC under ESPRIT-II Basic Research Working Group No. 6112 COMPASS and by CEC under ESPRIT BRA WG 6071 IS-CORE
template Step
  attributes quantity:nat initialized 0 .
  actions
    create birth .
    inc
      changing quantity:=succ(quantity) .
    dec
      enabled quantity>0
      changing quantity:=pred(quantity) .
    reset
      changing quantity:=0 .
    destroy death .
end template Step

The next example describes Counter objects which memorize natural numbers. This way, it can be counted how often a specific action has been executed. The remainder attribute can be initialized with a natural number and each execution of a tic action decreases it until it equals zero.

template Counter
  attributes remainder:nat initialized 0 .
  actions
    create birth .
    init(n:nat)
      changing remainder:=n .
    tic
      enabled remainder>0
      changing remainder:=pred(remainder) .
    destroy death .
end template Counter

To illustrate now the idea of independent actions of transactions and their arbitrary interleaving imagine the following four transactions T1, T2, T3, T4 composed of Step and Counter actions. St, Ct1, Ct2 are objects of type Step and Counter, respectively (; is sequential compositions and || denotes parallelism):

T1 = St.create
T2 = Ct1.create; Ct1.init(2); Ct1.tic||St.inc; Ct1.tic||St.inc
T3 = Ct2.create; Ct2.init(1); Ct2.tic||St.dec
T4 = St.reset

Executing these transactions in sequence cause the following observations qt after each Ti, (i = 1, ..., 4) about the attribute quantity:

\[ q_1 = 0; q_2 = 2; q_3 = 1; q_4 = 0 \]

Thus, after initializing the quantity with zero in T1, two is added to it by T2, afterwards T3 subtract one, and finally quantity is reset to zero by T4. We will elaborate on this later on (when these transactions become reifications of actions).

Without changing the overall effect on the attributes priority may be given to some actions through interleaving (see Fig. 1).

As depicted in Fig. 1 actions may be interleaved as long as they do not access the same resources, i.e., using the same attributes of objects. For the running example we have St.quantity, Ct1.remainder, Ct2.remainder. Therefore, actions on different objects are independent and, furthermore, actions operating on different attributes of one objects are also independent.

Assuming a dependency relation on actions it will be possible to deduce from a given sequence of transactions every correct interleaving. In the full paper we will describe in more detail how this can be technically derived. The idea of introducing the notion of dependency of actions to weaken serial composition has been investigated already in different contexts, as for example in process algebra [RW94] or in database theory [Kat94].
3 Event based Semantics

Semantics to objects is given by expressing language concepts through distributed temporal logic (DTL) expressions and give a denotational semantics in terms of event structures. DTL is a modification of the temporal logic for describing agents defined in [Thi94]. We will only give a rough idea how TROLL specifications can be explained using this logic since our main concern are reification issues.

The basic idea is that for each template actions are locally specified and that the global system may execute several (local) actions in parallel. Thus, we will distinguish between local and global actions. Semantically we deal with events, i.e., executions of global actions at a specific moment, by this uniquely fixing the local states of the participating states.

Every template specification establishes a specific object sort. Assuming a given object sort \( b \in \mathcal{S}_O \) (\( \mathcal{S}_O \) set of all object sorts) and a data signature \( \Sigma_D = (\mathcal{S}_D, \Omega) \) a specification of a template of sort \( b \) consists of three parts: an \( \mathcal{S}_D \)-indexed family of sets of local action symbols, an \( \mathcal{S}_D \times \mathcal{S}_D \)-indexed family of sets of component symbols, and a set of axioms formulated in distributed temporal logic. Furthermore, given a set of system-wide unique identities \( \mathcal{I}_b \) for every object sort \( b \), local actions of objects of this sort \( (\mathcal{L}_b) \) determine in the following way: \( \mathcal{L}_b = \{ \alpha \mid \alpha \in \mathcal{I}_b, \alpha \text{ local action of object of sort } b \} \) (local actions are generated by applying local action symbols to data terms).

In the given example we have two object sorts Step and Counter. The local actions symbols for Step are \( \{\text{create, dec, inc, reset, destroy}\} \). Therefore, with St as an identity of sort Step we get \( \{\text{St.create, St.inc, St.dec, St.destroy}\} \) as local actions.

Objects are semantically defined in terms of event structures \( E_i \) (\( i \in \mathcal{I}_b \)), where each local event structure \( E_i = (E_{\mathcal{V}_i}, \rightarrow_i) \) is a pair of a set of events and a causality relation constituting a partial order on events. Given an event structure for an object, every maximal totally ordered subset of it represents one possible life cycle of the mentioned object and causally independent events are always in conflict, i.e., single objects are sequential. A community of objects is established by gluing local event structures together at shared events. We will not go in detail how the events and their causality relation are precisely concluded from the given syntactical description of objects in DTL, but rather illustrate the intended semantics of objects with the running example.

Let's have a look at a Step object (see Fig. 2). The associated event structure (events are framed, arrows denote causality) reflects the idea that a step object may perform inc and reset actions at any point in time, whereas dec actions may only be performed when enough inc actions already have been occurred (since then the quantity attribute is greater than zero).

In the following we will specify a new template containing Step and Counter objects as components. By this we will also later on explain the semantics of aggregated objects (or say systems of objects) with shared events.

4 Reifying Actions to Transactions

Let us now come to the reification of actions by transactions. We already pointed out the main problems of synchronizing transactions in Sect. 2. Additionally we mentioned in passing, that the specified trans-
actions $T_1,\ldots,T_4$ simulate initialization ($T_1$), addition of two ($T_2$), subtraction of one ($T_3$), and reset ($T_4$). This way, we implemented the behavior of an Adder presupposing the following specification:

```
template Adder
    attributes quantity:nat initialized 0 .
    actions
        create birth .
        add(n:nat)
            changing quantity:=quantity+n .
        sub(n:nat)
            enabled quantity-n>=0
            changing quantity:=quantity-n .
        reset
            changing quantity:=0 .
            destroy death .
end template Adder
```

Therefore, we could employ Step and Counter to implement Adder. To achieve this we have to express abstract actions e.g., add, sub through transactions combined of Step and Counter actions (as already touched on by defining $T_1,\ldots,T_4$). More generally, it could be done as follows (Adder-R stands for reified Adder):

```
template Adder-R
    components
        St: Step
        Ct: Counter set
    attributes quantity:nat derived St.quantity.
    actions
        create TRAC CREATE.
        add(n:nat) TRAC ADD(n:nat) .
        sub(n:nat) TRAC SUB(n:nat) .
    ...
    process declaration
        CREATE = St.create .
        ADD(n:nat) = Ct.create -> Ct.init(n) -> INCSEQ .
        INCSEQ = {Ct.remainder>0} Ct.tic || St.inc -> INCSEQ .
    ...
end template Adder-R
```
As a composed object Adder-R is semantically described through a combination of event structures from Step and Counter objects by identifying some events. In more detail every transaction definition implies that occurrences of abstract actions (like add, sub etc.) are identified with the occurrences of the first concrete action of each involved object in the specified process (e.g., create and St.create, add and Ct.create, St.inc, etc.). Precisely, each transaction specification is projected to the sequence of actions of one engaged object and the corresponding abstract action is identified with each first action of those projections:

\[
\text{ADD}(n)_{\text{Step}} = \text{St.inc} \rightarrow \text{St.inc} \rightarrow \ldots
\]
\[
\text{ADD}(n)_{\text{Counter}} = \text{Ct.create} \rightarrow \text{Ct.init}(n) \rightarrow \text{Ct.tic} \ldots
\]

Thereby events of the corresponding event structures are shared as illustrated in Fig. 3. We only depicted parts of the involved event structures by focussing on the sequence of transaction defined in Sect. 2 (arrows pointing to emtyness indicate to the rest of the local event structures). The independence of actions of different objects is preserved as far as possible by composing the local event structures in a concurrent manner. Only necessary causalities are propagated through event sharing. E.g., in Fig. 3 the \text{create;init}(1) event of Ct2 and the \text{create;init}(2) event of Ct1 are concurrent and may be performed in any order as intuitively wanted (cf. Fig 1).

The intended event structure of the reified object Adder-R is the combination of all maximal life cycles of the components selected by shared events. How this semantics of the reified object is constructed has to be worked out in more detail in the full paper.

5 Conclusions and Future Work

In this paper we roughly gave an idea how transactions can be introduced in object-oriented specifications. We proposed a semantics for objects which is able to adequately capture the idea of action reification and, moreover, allows to integrate some aspects from concurrency control. This way, we treated reification semantically.

After we have now presented our ideas by example a lot work remains to be done. For example, the relationship between abstract models and concrete ones has to be defined more precisely. Furthermore, it is still open in which way the model of the reified object can be deduced from the ones of the components and the additionally specified axioms.

We only considered the semantical aspect of reification up to now. On the syntactical side we intend to work on the nature of reification functions. We will establish a language for describing (systems of) objects in which reification is declared. We are interested in the kind of formulas for which validity remains after application of reification, i.e., proofs of formulas commute with applications of reification functions. There exists some work on syntactical and semantical action refinement in process algebras done by the Hildesheim group of U. Goltz (cf. [Web89] and some work in progress). Using the semantical framework of event structures they provide results on the refinement of formulas formulated in a variant of the trace based extension of PTL [Thi94]. We wish to apply their outcomes to our model to characterize the class of formulas for which reification works out. We will give more details in the full paper.
Acknowledgements

Thanks to our Braunschweig colleagues who helped a lot to improve this paper by critical and constructive remarks. Furthermore, our partners from the IS-CORE and COMPASS projects contributed to this work with fruitful discussions on semantics and reification topics. Special thanks are due to J. Fiadeiro, U. Goltz, M. Huhn, P. Niebert, and H. Wehrheim for their valuable support in clarifying our understanding of reification issues.

References


