An Object-Oriented Airport: Specification and Refinement in Maude

Ulrike Lechner[1], Christian Lengauer[1] and Martin Wirsing[2]

[1] Fakultät für Mathematik und Informatik, Universität Passau, D–94030 Passau, Germany, email: {lechner,lengauer}@fmi.uni-passau.de
[2] Institut für Informatik, Ludwig-Maximilians-Universität, D–80802 München, Germany, email: wirsing@informatik.uni-muenchen.de

Abstract. An object-oriented model of an airport has been developed to assess the parallel object-oriented specification language Maude. The model includes airplanes, gates, baggage handling, ground control and tower and has been implemented on the OBJ3 system, which serves as a rudimentary interpreter for Maude. We discuss two ways of specifying objects and present two notions of behavioral refinement in Maude.

1 Introduction

Maude [15, 16, 18] is a language for object-oriented specification. Its semantics is based on concurrent rewriting and can be viewed as a natural extension of the semantics of the OBJ3 rewriting system [9]. Our example is based on a case study at the University of Passau [21], whose goal was the development of a detailed model of an airport in order to gain experience in modeling with Maude and develop techniques for object-oriented specification. As far as possible, the model was implemented and validated in OBJ3. We present a refinement from an abstract to a concrete specification in Maude and propose two different notions of refinement: behavioral transition refinement and behavioral simulation refinement.

2 Overview of Maude

We concentrate on the object-oriented part of Maude and on its module concept. For a more detailed description, see [15, 16, 18].

In Maude, a *class* is declared by an identifier and a list of attributes and their types. `OId` is the type of Maude identifiers reserved for all object identifiers.

The object-oriented concept in Maude is the *object module*. The declaration of an object module (keyword `omod`) consists of an import list (`protecting`, `extending` or `using`), a number of class declarations (`class`), message declarations (`msg`), variable declarations (`var`), rewrite rules (`rl`) and, possibly, equations (`eq`).

A *message* is a term that consists of the message's name, the identifiers of the objects the message is addressed to and, possibly, parameters (in mixfix notation).

A Maude program makes computational progress by rewriting its global state, its *configuration*. (In the rest of this paper, we use the term "state" only in the airport world that we are modeling.) A configuration is a multiset, or bag, of objects and messages. A rewrite step transforms a configuration into a subsequent configuration.

Objects are declared together with in rules for rewriting the configuration. An *object* is represented by a term—more precisely by a tuple—comprising a unique object identifier, an identifier for the class the object belongs to and a set of attributes with their values. E.g., the term `< P : Plane | Delay: D >` represents an object with object identifier `P` belonging to class `Plane`. The attribute `Delay` has value `D`.

The following specification of class `Plane` plays a central role in our case study:

```
omod M-PLANE is
   protecting  "some modules where basic types are declared" .
   class Plane |
       FlightNo  : Nat,    Destination: OId,
       PlaneType : String, Runway     : Nat,
       TOLControl: OId,    Tower      : OId,
       Departure : Time,   Arrival    : Time, Delay: Time .
   msg to _ NewDelay _ : OId Time -> Msg .
   msg to _ NewArrivalTime of _ is _ : OId OId Time -> Msg .

   var P T : OId .
   var A D N : Time .

   rl (to P NewDelay N)
       < P : Plane | Arrival: A, Delay: D, Tower: T >
   =>  < P : Plane | Delay: D + N >
      (to T NewArrivalTime of P is (A + D + N) ) .
endom
```

Class `Plane` has several attributes. The attributes `Destination`, `TOLControl` (take-off-and-landing control) and `Tower` are used to store identifiers of objects to which a plane sends messages. The attributes `FlightNo`, `Runway`, `Departure`, `Arrival` and `Delay` contain the data of the current flight of the plane. Objects of class `Plane` accept one message: `(to P newDelay N)` indicates that a new delay of duration `N` has occurred in the schedule of plane `P`. Sending the message triggers a state change of plane `P` and initiates a message to the tower `T` to indicate that the plane has a new arrival time. The arrival time is computed from the scheduled arrival time `A`, the delay `D` and the new delay `N`. The rewrite rule is applicable in configurations which contain a message and a plane with matching object identifier.

Note that, in the rewrite rule, only those attributes are mentioned whose values are changed or needed for some computation. For example, the attributes `FlightNo`, `Destination`, `PlaneType`, `TOLControl` and `Departure` of class `Plane` are not mentioned. The value of the attribute `Tower` is needed as an address for the initiated message. Additionally, we are allowed to omit attributes whose values remain unchanged at the right-hand side of rewrite rules, e.g., the attribute `Arrival`.

One distinguishing feature of Maude is that it supports two structuring mechanisms: class inheritance and module inheritance.

Class inheritance is subtyping. The declaration `subclass FlyingPl < Plane` states that class `FlyingPl` (assumed to be previously declared) is a subclass of `Plane`; then, `FlyingPl` inherits all attributes from `Plane`, and all rules that are applicable to an object of class `Plane` are also applicable to objects of class `FlyingPl`.

Module inheritance is the concept of importing code pieces into a module. Maude has three ways of importing modules, denoted by the keywords `protecting`, `extending` and `using`. `protecting` prohibits all changes of the sorts of the imported module ("no junk, no confusion" [15]), `extending` allows adding new constants to the declared sorts and `using` (`by` f) allows arbitrary changes and extensions of the defined sorts according to a stated signature morphism f.

The semantics of Maude is based on concurrent term rewriting. We present the four rewriting rules for the unsorted case of unconditional rewrite rules. For a more detailed description, see [13]. A specification in Maude is a rewrite system $R = (\Sigma, E, L, T)$, where $\Sigma = (S, C, \leq, F, M)$ is a signature (consisting of sort names, class declarations, subsort ordering, function symbols and message names), E a set of equations, L a set of labels of rewrite rules and T a set of rewrite rules for state transitions. The rewrite rules in T model transitions between configurations, i.e., between equivalence classes $[t]$ of terms, here equivalence classes modulo the set of equations E.

$T(\Sigma, X)$ denotes the set of all terms over the signature Σ with free variables X. $T^E(\Sigma, X)$ is the same set partitioned into equivalence classes by the equations in E.

$R \vdash [t] \to [t']$ denotes that $[t]$ can be rewritten to $[t']$ by finitely many applications of the rules in R and the four rules of the rewriting calculus below.

Maude's semantics has a static and a dynamic part. The static part, the semantics of the function symbols and the objects and their classes, is defined by the signature Σ and the set E of equations of the rewrite system. The dynamic part is defined by the set of rules T for concurrent rewriting of configurations.

There are four rewrite rules of the rewriting calculus:

Reflexivity. For each $[t] \in T^E(\Sigma, X)$:

$$\frac{}{[t] \to [t]}$$

Transitivity.

$$\frac{[t_1] \to [t_2], [t_2] \to [t_3]}{[t_1] \to [t_3]}$$

Congruence. For each $(f : s_1 \times \ldots \times s_n \to s) \in F$

$$\frac{[t_1] \to [t'_1] \ \ldots \ [t_n] \to [t'_n]}{[f(t_1, \ldots, t_n)] \to [f(t'_1, \ldots, t'_n)]}$$

Replacement. For each rewrite rule $r : [t(x_1, \ldots, x_n)] \to [t'(x_1, \ldots, x_n)]$ in T:

$$\frac{[w_1] \to [w'_1] \ \ldots \ [w_n] \to [w'_n]}{[t(\overline{w}/\overline{x})] \to [t'(\overline{w}'/\overline{x})]}$$

The congruence and the replacement rule allow simultaneous rewrites. Rewrite rules can only be applied concurrently when their redexes are disjoint.

3 Modeling States

The state of an object plays a decisive role in the way the object reacts to a message sent to it—most fundamentally, whether it reacts at all! In the literature, a popular example for an object whose behavior differs depending on the internal state is a buffer that accepts a "get" message only if it is not empty [12, 16]. We use planes to illustrate how the states and behaviors of objects might be represented and how their representation interacts with inheritance (Sect. 3) and with the refinement of specifications (Sect. 4).

The behavior of a plane depends on two properties: its type and its state. There are two plane types: cargo planes and passenger planes. We use five different kinds of states to model the behavior of a plane. Passenger and cargo planes behave differently on the ground but identically from take-off to landing. That is, the tower and the take-off-and-landing (TOL) control treat all planes equally. We model four different kinds of states of a (cargo or passenger) plane that are relevant for the tower: Starting (waiting for take-off; this kind of state will be used in Sect. 4), Takeoff, Flying and Landing. One further kind of state is of relevance for the ground control and has different meanings for cargo and passenger planes, respectively: OnGround. Fig. 1 depicts a plane's behavior in terms of state transitions.

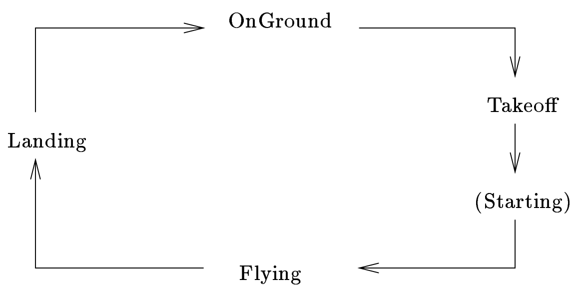

Fig. 1. The states of a plane

We consider two different ways of modeling states. In one model, states are represented by attributes; this model can be adopted similarly in other specification languages. In the other model, states are represented by classes; we argue that this model is more object-oriented—and typical for Maude because it makes better use of Maude's concept of class inheritance and Maude's convenient provisions for letting an object change its class membership.

Both models represent the same behavior of planes with respect to airport institutions like the tower and the ground control.

In the following sections, we illustrate both models on the transition from Flying to Landing.

3.1 States as Attributes

In conventional object-oriented programming languages, the state of an object is often modeled by a (hidden) attribute, or attributes. We use boolean attributes, i.e., we install one "flag" for each state. Assume we have declared two subclasses of `Plane`: `CargoPl` and `PassengerPl`. Among others, a plane has the two attributes

`Flying` and `Landing`. Then, e.g., the transition from the state that represents a plane in flight to the state that represent the landing of that plane can be defined by the following, self-explanatory rewrite rule:

```
rl (landing P)
    < P : Plane | Flying: true, Landing: false, Runway: X, Tower: T >
=>  < P : Plane | Flying: false, Landing: true >
   (to T from P landing on X) .
```

The advantage of this solution is that the rule is applicable to planes belonging to both subclasses `CargoPl` and `PassengerPl`.

The disadvantage is that the different kinds of states of an object are modeled in the same way as other properties of the object: by attributes. We believe that, because of their central role and, potentially, high number, states deserve a special treatment. Explicit state changes by attribute redefinition seems cumbersome and error-prone.

Furthermore, adding a new kind of state in the states-as-attributes model does not observe the principle that previously written code and its properties should not change in an extension.

3.2 States as Classes

One can also model state change by making the object change class. This model requires (at least) one class per state.

There are several kinds of classes: classes that determine the plane type (`CargoPl` or `PassengerPl`), classes that determine the state of the plane (`OnGroundPl` ...) and classes that contain the actual planes and inherit type and state.

The advantages of the states-as-classes model are:

- The messages that are applicable only in a certain state can be defined in the scope of the class that models the state. The result is a more modular and more transparent module structure.
- The specification of the data and of the dynamic behavior are quite independent from each other. A refinement of the dynamic structure often induces a refinement of the data by new attributes and new classes (see Sect. 4). The fine-grained class structure of states as classes increases the safety of the specification (see Sect. 6), and it is supported by behavioral refinement (see Sect. 5).

Let us sketch the class structure and the corresponding module structure of our plane model. Fig. 2 depicts the class hierarchy for one concrete class `FlyingCargoPl`.

In the following, we sketch how we build up the class and module hierarchy. Each class is wrapped inside a module. For the declaration of the class plane, see Sect. 2.

All subclasses of `Plane` inherit the attributes of `Plane`. The two kinds of planes have different attributes, e.g., the class `CargoPl` has one additional attribute, `Weight`, the maximal weight it is allowed to transport.

A module `MK-CARGOPLANE` imports all modules in which the whole class hierarchy for cargo planes is set up. This class hierarchy is reproduced for passenger planes. The module concept of Maude facilitates duplication of the entire class hierarchy with a

```
class FlyingPl .
subclass FlyingPl < Plane .

class CargoPl | Weight: Int .
subclass CargoPl < Plane .

class FlyingCargoPl .
subclass FlyingCargoPl < CargoPl FlyingPl .
```

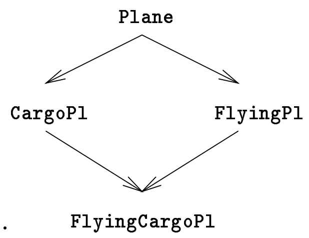

Fig. 2. The class hierarchy for a flying cargo plane

single signature morphism. This morphism is applied to the encapsulating module `MK-CARGOPLANE`. Additionally, the attribute `Weight` is renamed to `#Passengers`, the maximal number of passengers.

```
omod MK-PASSPLANE is
using MK-CARGOPLANE *(class CargoPl to PassengerPl,
                      class FlyingCargoPl to FlyingPassPl,
                      "more renamings of classes"
                      att Weight to #Passengers) .
```

Up to now, we have developed a class hierarchy without any messages or rewrite rules. Let us discuss how message declarations and rewrite rules fit into the class concept with multiple inheritance and states as classes.

The states-as-classes model has one drawback in the specification of the rewrite rules. [16] discusses inheritance of rewrite rules for objects that do not change class. The following rewrite rule is problematic because the resulting class `LandingPl` is not declared properly:

```
rl (landing P)
    < P : FlyingPl  | Runway: X, Tower: T >
=>  < P : LandingPl >
   (to T from P landing on X) .
```

In the denotational semantics of Maude [9, 15], the class hierarchy of planes corresponds to a hierarchy of sets. To make a special point, we have added an additional refinement of passenger planes in which we specify the company producing the plane:

```
        Plane ⊇ FlyingPl, LandingPl, PassengerPl
  PassengerPl ⊇ Boeing, Airbus
 FlyingPassPl = FlyingPl ∩ PassengerPl
```

Assuming the parameters of a message are fixed, rewrite rules induce mappings between these sets, e.g., the rule `landing` between the sets `FlyingPl` and `LandingPl`.

The denotational semantics of Maude permits object `P` to change in any way consistent with the rule, as long as it does not violate the requirements on objects of class `FlyingPl` and `LandingPl`. This becomes a problem when different subclasses are characterized by the same attributes.

In our example, passenger and cargo planes (both subclasses of `Plane`) do have different attributes and, thus, can be distinguished, but Boeing and Airbus planes (both passenger planes) cannot be distinguished. Consider the rule `landing`. It specifies a transition of one object from class `FlyingPl` to class `LandingPl` without specifying the subclass of `LandingPl` that the object should belong to. Given that the object is a passenger plane, it is perfectly consistent with the rule that it change its type from `Boeing` to `Airbus`, say, since both types are indistinguishable in the rule.

This problem can be avoided in two different ways. One alternative is to use a different object model in which, instead of a single class identifier, a set or a record of class identifiers specifies the state transitions more precisely. The rule `landing` would specify that the class `PlaneType` remains the same, while the class representing the state changes:

```
... < P : {PlState: FlyingPl , Type: PlaneType} >
=>  < P : {PlState: LandingPl, Type: PlaneType} > ...
```

Another alternative is to add explicit subtyping to Maude and specify the change of the class explicitly:

```
rl (landing P)
    < P : C | Runway: X, Tower: T >
=>  < P : C' >
   (to T from P landing on X)  if C < FlyingPl and C' < LandingPl and
        (forall D : (C < D  and not(D <= FlyingPl)) implies C' < D) .
```

For an implementation, this extension is a serious step. However, the implementation of Maude in OBJ3 already goes part of the way by providing class variables for subsort polymorphism. Both solutions reduce the number of classes needed because they do not refer to the concrete class, e.g., `FlyingCargoPl`, but to its superclasses. In the rest of the paper, we assume that a rewrite rule specifying a state transition changes only the class mentioned in the rule and leaves other superclasses the objects belong to unchanged.

4 From Synchronous to Asynchronous Communication

Maude supports two different ways of communication: synchronous and asynchronous communication. A communication is called *synchronous* when a rewrite step involves more than one object on the left-hand side; an *asynchronous* communication is defined by a rewrite rule with only one object (but possibly several messages) at the left-hand side [15, 18].

We investigate the model of the take-off of a plane in either communication style. The synchronous model describes the change of the states of the objects. The asynchronous model specifies, additionally, the implementation of the communication protocol. Thus, the asynchronous model can be viewed as a refinement of the synchronous model. We present formal definitions of refinement in Sect. 5.

One important difference between the two styles of communication is the degree of parallelism (our metric is the number of replacement rules applicable in parallel [16]). An object is only allowed to be involved in one rewrite rule per rewrite step, even if its state is not changed. Thus, objects that are involved in many communications reduce the concurrency in a specification.

We illustrate the take-off of a plane on a runway. A plane that is ready for take-off sends a request to the TOL control. The TOL control is responsible for the scheduling, the tower for the use of the two (independent) runways. A plane is cleared for take-off when it is next in line according to the schedule and a runway is free. Take-off is modeled by change of the plane's state to Flying. A plane is identified by the plane number (the object identifier) and the flight number because it is possible for one plane to make several flights from one airport in one day.

4.1 Synchronous Communication

In a synchronous communication, we can represent the entire take-off procedure by a single rewrite rule:

```
rl (takeoff?(P, S, T))
    < P : TakeoffPl  | FlightNo: No, Runway: X, Destination: D >
    < S : TOLControl | Schedule: [(P,No),RestSchedule] >
    < T : Tower      | FreeRunways: X FreeSet >
=>  < P : FlyingPl >
    < S : TOLControl | Schedule: RestSchedule >
    < T : Tower      | FreeRunways: X FreeSet >
   (toDestination D flying (P,No)) .
```

This is not a faithful model of an airport because data are exchanged without communication—not explicitly via messages as in the "real world". Moreover, the tower and the TOL control are unable to deal with other planes simultaneously when handling a take-off.

4.2 Asynchronous Communication

In our asynchronous model, the single transition of the synchronous model is divided into a number of rewrite steps, each involving only one object. Thus, the model reflects not only the state change of the plane, the TOL control and the tower but also the communication protocol that leads to this change.

A new plane state, Starting, ensures that, during the time interval in which the plane is waiting for permission to take off, no other action can be taken that is possible in the states Takeoff or Flying. This new state simplifies the correctness proof of the refinement—the extension of the synchronous model—because none of the rules that are applicable in the states of the abstract specification is applicable in the new state Starting.

A request to the TOL control for clearance for take-off (right-hand side of the rule) is accompanied with all relevant flight data:

```
rl (takeoff?(P,S,T))
    < P : TakeoffPl | FlightNo: No, Runway: X, TOLControl: S >
=>  < P : StartingPl >
   (toTOLControl S clearance? (P,No) on X) .
```

The TOL control checks that the plane is scheduled as the next plane to take off and sends an inquiry concerning the state of the runway to the tower:

```
rl (toTOLControl S clearance? (P,No) on X)
    < S : TOLControl | Schedule: [(P,No),RestSchedule], Tower: T >
=>  < S : TOLControl >
   (toTower T clearance? (P,No) on X) .
```

If the requested runway is free, the tower stores the data of the flight in a list and informs the plane that it is cleared:

```
rl (toTower T clearance? (P,No) on X)
    < T : Tower | FreeRunways: X FreeSet, OnRunway: OnRunwaySet >
=>  < T : Tower | FreeRunways: FreeSet,
                  OnRunway: (X,(P,No)) OnRunwaySet >
   (to P cleared at X) .
```

The plane, in state Takeoff, takes off when receiving the message `cleared`. The take-off is modeled by messages from the plane to itself and to the TOL control:

```
rl (to P cleared at X)
    < P : StartingPl | Runway: X, TOLControl: S >
=>  < P : StartingPl >
   (toTOLControl S takeoffack of P at X) (airborne P) .
```

Message `airborne` triggers a state transition from Starting to Flying. Also, the plane sends a message to the tower that the runway is free and a request to the target airport `D` to send the new object identifiers of, e.g., the TOL control and the tower:

```
rl (airborne P)
    < P : StartingPl | FlightNo: No, Runway: X,
                       Destination: D, Tower: T >
=>  < P : FlyingPl >
   (toTower       T  (P,No) airborne at X)
   (toDestination D  flying (P,No)) .
```

TOL control and tower of the source airport react to the messages of the plane:

```
rl (toTOLControl S takeoffack of P at X)
    < S : TOLControl | Schedule: [(P,No),RestSchedule] >
=>  < S : TOLControl | Schedule: RestSchedule > .

rl (toTower T (P,No) airborne at X)
    < T : Tower | FreeRunways: FreeSet,
                  OnRunway: (X,(P,No)) OnRunwaySet >
=>  < T : Tower | FreeRunways: X FreeSet, OnRunway: OnRunwaySet > .
```

In both models, synchronous and asynchronous, a take-off results in the same state: the plane is in state Flying, the flight is removed from the list of planes waiting for take-off and the runway is free.

The increase in parallelism we obtain in the asynchronous model is that two planes may take off or land independently on the two runways and that the tower can answer other communication requests, e.g., for the control of planes waiting and circling in the air. The asynchronous model is more realistic because the communication is explicit and only one object is involved in one rewrite step and, thus, both tower and TOL control can handle several planes at a time.

5 Behavioral Refinement of Specifications

In this section, we present some ideas on how to refine specifications written in Maude. Consider the refinement of a synchronous by an asynchronous specification in the previous section. The refined specification contains a new class, i.e., a new intermediate state, new messages that model the asynchronous exchange of data and new intermediate state transitions.

We abstract from the way a state transition is deduced from rewrite rules and consider only the existence of a state transition. We regard a rewrite system (Σ, E, L, T) as an algebraic specification with an additional relation $\rightarrow$. In this paper, we restrict ourselves to the syntactic characterization of refinement. We will develop the corresponding semantic notions in future work.

A refinement is a relation between two specifications. We refer to them as the abstract and the concrete specification. In our setting, both specifications are written in Maude.

Our work on refinement is based on the loose approach of Wirsing [24]. In this approach, the semantics of a specification is the set of all algebras that satisfy the equations and contain the state transition relation. Refinement can be characterized by implication: all equations and state transitions deducible in the abstract specification have to be deducible in the concrete specification as well.

$T_s(\Sigma)$ is the set of all ground terms of sort s over the signature Σ. One constant for sorts we use is "Conf", an abbreviation of the sort `Configuration`, the sort of the configurations of a rewrite system.

Note that $\Sigma \subseteq \Sigma'$ if and only if $(S \subseteq S') \wedge (C \subseteq C') \wedge (\leq \subseteq \leq') \wedge (F \subseteq F') \wedge (M \subseteq M')$.

Definition 1 (Refinement). A rewrite system R is *refined* by a rewrite system R', written $R \rightsquigarrow R'$ iff

$$\begin{aligned} & \Sigma \subseteq \Sigma' && (1) \\ \wedge\ & (\forall\, t_1, t_2 \in T(\Sigma) : R \vdash t_1 = t_2 \ \Rightarrow\ R' \vdash t_1 = t_2) && (2) \\ \wedge\ & (\forall\, c, c' \in T_{\mathrm{Conf}}(\Sigma) : R \vdash c \rightarrow c' \ \Rightarrow\ R' \vdash c \rightarrow c') && (3) \end{aligned}$$

The step from synchrony to asynchrony in Sect. 4 involves a refinement of the object model, the communication and the order in which the objects reach their common final configuration. Since we use the loose approach, we are allowed to refine an abstract state transition specified by a single application of some rewrite rule to a concrete state transition specified by multiple applications of rewrite rules—as long as every abstract state transition is represented by a (sequence of) concrete state transitions.

The observable part of a specification consists of the primitive sorts, classes and messages that we would like to observe.

Definition 2 (Observable part). The *observable part Obs* of a rewrite system R with $\Sigma = (S, C, \leq, F, M)$ is a triple (S', C', M') such that $S' \subseteq S$, $M' \subseteq M$ and $(\texttt{c'|atts'}) \in C' \Rightarrow (\exists\ \texttt{atts} : ((\texttt{c'|atts}) \in C\ \wedge\ \texttt{atts'} \subseteq \texttt{atts}))$.

We presume that all sorts of observable attributes and the class identifiers are observable.

We write $\texttt{<o:c|a1:v1,...,an:vn>} \in C'$, where o is an object of class $(\texttt{c|a1:s1,...,an:sn}) \in C'$, and $(\texttt{m(p1,...,pn)}) \in M'$, where m is a message built using $(\texttt{m:s1...sn -> Msg}) \in M'$.

Subclasses do not inherit observability properties from their superclasses: the attributes that we observe in a class are not necessarily observed in all subclasses or all superclasses.

Note that the configurations are terms of the sort `Configuration`, which consists of both observable and unobservable messages and objects. In order to be able to observe part of a configuration, we define an observation function obs_R on the configurations of rewrite system R that projects the observable part $Obs = (S', C', M')$. We call the images of the observation function *observations*. Let us state the definition and then explain it.

We write $\cup$ for the multiset union of configurations and ϵ for the empty configuration.

Definition 3 (Observation function). The observation function obs_R of a rewrite system R for an observable part *Obs* is defined by:

$$\mathrm{obs}_R : \texttt{Configuration} \rightarrow \texttt{Set of Configuration}$$

$$\mathrm{obs}_R(\epsilon) = \{\epsilon\} \tag{1}$$

$$\mathrm{obs}_R(\texttt{c1} \cup \texttt{c2}) = \{a \cup b \mid a \in \mathrm{obs}_R(\texttt{c1}) \wedge b \in \mathrm{obs}_R(\texttt{c2})\} \tag{2}$$

$$\mathrm{obs}_R(\texttt{<o:c|atts>}) = \begin{cases} \{\texttt{<o:c'|atts'>} \mid \texttt{<o:c'|atts'>} \in C' \wedge \texttt{c} \leq \texttt{c'}\} & \\ \qquad\qquad \text{if } (\exists\ c' \in C' : c \leq c') & \\ \{\epsilon\} \qquad\quad \text{otherwise} & \end{cases} \tag{3}$$

$$\mathrm{obs}_R(\texttt{(message)}) = \begin{cases} \{\texttt{(message)}\} & \text{if } \texttt{(message)} \in M' \\ \{\epsilon\} & \text{if } \texttt{(message)} \notin M' \end{cases} \tag{4}$$

Each observation of a configuration is a set of observable parts. Each object may be observable in a number of superclasses, and its observations may differ depending on which superclass observes it. Thus, the range of the observation function is a set of configurations. Remember, that a configuration is itself a multiset of messages and objects.

The observations in the abstract specification form a subset of the observations in the concrete specification because a refinement into new subclasses may generate new subtype dependences between observable classes. This may lead to added observations (see our example in Sect. 6).

Our concept of an observable part refines the concept of an observable part defined for algebraic specifications in [3, 4, 20, 24] and the concept of a visible action in process algebras in [19]. Our extension covers the addition of attributes and messages and the refinement of the class hierarchy and of communication protocols.

In the following, we make two suggestions for relating the behavior of an abstract specification to a concrete specification: behavioral transition refinement and behavioral simulation refinement. The refinement of the static part of the language is the same for both refinement relations. See Sect. 6 for a discussion and a comparison of the two refinement relations.

Definition 4 (Behavioral transition refinement). A rewrite system R' is a *behavioral transition refinement* of a rewrite system R with respect to an observable part *Obs*, written $R \leadsto^{\mathrm{tr}}_{Obs} R'$, iff

$$
\begin{array}{ll}
\quad \Sigma \subseteq \Sigma' & (1)\\
\wedge \ (\forall\ s \in Obs,\ t_1, t_2 \in T_s(\Sigma) : R \vdash t_1 = t_2 \Rightarrow\ R' \vdash t_1 = t_2) & (2)\\
\wedge \ (\forall\ c, c' \in T_{\mathrm{Conf}}(\Sigma) : & (3)\\
\qquad (\exists\ d, d' \in T_{\mathrm{Conf}}(\Sigma') : \mathrm{obs}_R(c) \subseteq \mathrm{obs}_{R'}(d)\ \wedge\ \mathrm{obs}_R(c') \subseteq \mathrm{obs}_{R'}(d')\ \wedge & \\
\qquad\quad (R \vdash c \to c'\ \Rightarrow\ R' \vdash d \to d'))) &
\end{array}
$$

We call this relation transition refinement because it relates a transition in the abstract specification to a transition in the concrete specification. This relation takes neither traces, i.e., sequences of transitions nor choices into account. Thus, one configuration can be refined into several configurations that are not connected appropriately by transitions: each concrete configuration may have fewer successor configurations than the abstract one, or transitions between the concrete configurations may be missing. This relation is, in some sense, the weakest possible relation between two specifications. In process theory, a relation that ensures that the behavior of the concrete specification has at least the same traces and the same choices is called a simulation. We have a similar refinement relation here.

Definition 5 (Behavioral simulation refinement). A rewrite system R' is a *behavioral simulation refinement* of a rewrite system R' with respect to an observable part *Obs*, written $R \leadsto^{\precsim}_{Obs} R'$, iff (1) and (2) of Def. 4 hold and, in addition,

$$
\begin{array}{ll}
(\exists\ \precsim\ \subseteq T_{\mathrm{Conf}}(\Sigma) \times T_{\mathrm{Conf}}(\Sigma') : (c \precsim d\ \Rightarrow & (3)\\
\quad (\forall\ c' \in T_{\mathrm{Conf}}(\Sigma) : R \vdash c \to c' \Rightarrow (\exists\ d' \in T_{\mathrm{Conf}}(\Sigma') : R' \vdash d \to d' \wedge c' \precsim d'))) & \\
\quad \wedge\ (\forall\ c \in T_{\mathrm{Conf}}(\Sigma) : (\exists\ d \in T_{\mathrm{Conf}}(\Sigma') : c \precsim d)) & \\
\quad \wedge\ (c \precsim d\ \Rightarrow\ \mathrm{obs}_R(c) \subseteq \mathrm{obs}_{R'}(d))) &
\end{array}
$$

In the refinement step in Sect. 4, the observable part consists of the classes with the attributes and the messages that appear in the abstract specification. Not observable are the class that models the intermediate state Starting, the attributes necessary to store object identifiers and the messages necessary for implementing the asynchronous communication.

The observation functions are the identity for the abstract (synchronous) specification and the corresponding projection in the concrete (asynchronous) specification.

The only non-trivial state transition defined in the abstract specification is deducible for observably equal configurations in the concrete specification. Thus, the concrete specification is both a behavioral transition refinement and a behavioral simulation refinement of the abstract specification. Since our approach ensures only that the state transitions in the abstract specifications exist in the concrete specification we admit additional transitions in the concrete specification. To ensure that the effect of the refinement of this single state transition of the take-off remains local we have introduced the class Starting. No rules that are applicable in the two states Take-off and Flying are applicable in the state Starting. The changes of the attributes of the TOL-Control and the Tower is rather deterministic and is done by messages only. The safety of the refinement step relies thus on the States-as-Classes concept.

6 Behavioral Transition vs. Behavioral Simulation Refinement

Let us illustrate that both notions of behavioral refinement presented in the previous section—transition and simulation refinement—have reasonable applications. We expect the concrete specification to be at least as powerful as the abstract specification, i.e., to offer all choices and all transitions that exist at the abstract level. From this point of view, behavioral simulation refinement is the better choice.

Here is an example for a behavioral transition refinement which demonstrates that simulation refinement can be too strong and that behavioral transition refinement can be a good choice as well. We omit the Maude code; we illustrate the state transitions graphically in Fig. 3 instead.

Let us consider a new state, Loading, for planes on the ground. This state differs for passenger and cargo planes. A passenger plane loads passengers, a cargo plane cargo. All loading planes have the additional attributes `cargo` and `pass`. An attribute is `true` if the corresponding entity has been loaded and `false` otherwise. The process of loading is modeled by two different messages, `(to P load cargo)` and `(to P load passengers)`. These messages are sent from the ground control to the plane to be loaded.

In the abstract specification, a class, `LoadingPl`, represents a plane that is ready to load cargo or passengers. Both cargo and passenger planes accept both requests, although one of them contradicts the value of attribute `PlaneType`.

Let us refine this specification by introducing two subclasses, `LoadingCargoPl` and `LoadingPassPl`. Both subclasses accept only the message compatible with the plane's type. This refinement increases safety because the plane can reject messages of the ground control that are not compatible with its type.

Of our specification, we would like to observe the class `LoadingPl` with its two attributes and the two messages. We would also like to make some observations of

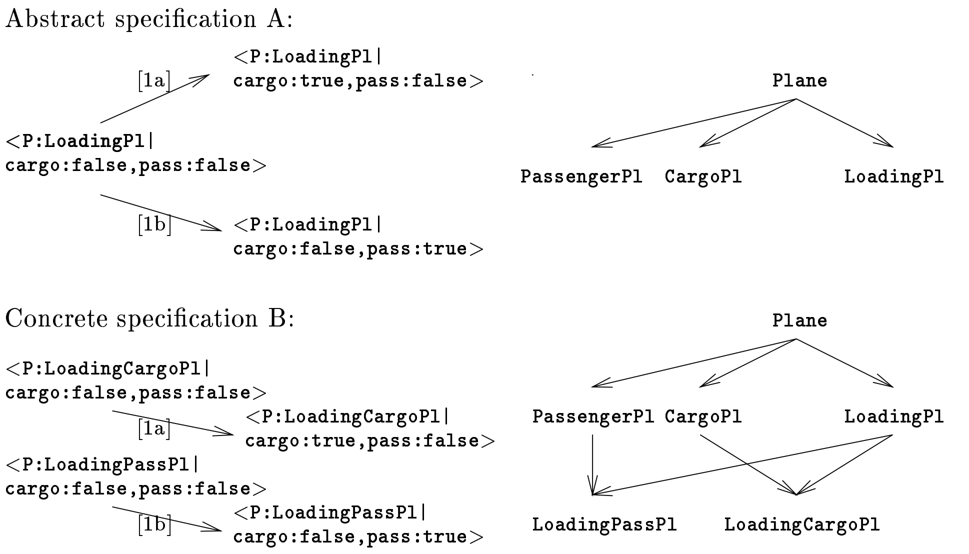

Fig. 3. State transitions and class hierarchy of the abstract and the concrete specification

class `Plane` and classes `PassengerPl` and `CargoPl`. The observational part of the specifications consists of:

S' = "all primitive sorts"
C' = { `(Plane | FlightNo: Nat, Destination: OId)`, `(PassengerPl | #Passengers: Nat)`, `(CargoPl | Weight: Nat)`, `(LoadingPl | cargo: Bool, pass: Bool)` }
M' = { `to _ load cargo : OId -> Msg`, `to _ load passengers : OId -> Msg` }

Let us consider the configuration D (the variable `ATTS` subsumes attributes of `P` not needed in this example):

```
D  =  < P : LoadingPl | cargo: false, pass: false,
                        PlaneType: Passenger, Destination: D,
                        FlightNo: N, #Passengers: 100, ATTS >
      (to P load passengers)
```

In the abstract specification, the plane is observable in two classes, `LoadingPl` and `Plane`. Our observation is a set of two configurations that are different projections of D:

$\text{obs}_A(D)$ =
{ `< P : LoadingPl | cargo: false, pass: false > (to P load passengers)`,
`< P : Plane | FlightNo: N, Destination: D > (to P load passengers)` }

In the concrete specification, the smallest sort of plane `P` is `LoadingPassPl` (where the attribute `cargo` is false). Configuration D', the refinement of configuration D

which we observe in the concrete specification, differs from configuration D in one point: the class the plane belongs to is `LoadingPassPl`, not `LoadingPl`. By (3) of Def. 3, the plane is observable in one more class, `PassengerPl`. Thus, our observation of configuration D' is:

$\mathrm{obs}_B(D') = \mathrm{obs}_A(D) \cup$
$\{$ `< P : PassengerPl | #Passengers: 100 > (to P load passengers)` $\}$

The refinement is a behavioral transition refinement. Every state transition in the abstract specification is implemented by a corresponding state transition in the concrete specification. In Fig. 3, corresponding state transitions are labelled identically.

The refinement is not a behavioral simulation refinement. In the abstract specification, a plane in state Loading, with both attributes false (as in configuration D), can accept two different messages. In the concrete specification, there are two different states in each of which only one message can be accepted.

7 Related work

Several papers of Meseguer et al. cited earlier point out that synchronous and asynchronous specifications define**c:[???]** different levels of abstraction. In [11], an asynchronous implementation of a synchronous specification of a spreadsheet is presented and related issues of the transformation from synchrony to asynchrony, of deadlock and fairness are discussed. In general, Meseguer refers to a categorical semantics of Maude and relates the rewriting logic to other logical frameworks [17]. In [14], a rewrite relation similar to the configuration transition relation used by us is defined and a mechanism for an implementation relation between Lawvere categories is presented. **c:[???]**

A different approach to integrating the specification of static data types and of dynamic process behavior, more closely related to the traditional approach of algebraic specification, is SMoLCS [1, 2]. The models are *dynamic algebras*, i.e., algebras with relations. A hierarchy of labelled transition systems specifies data types and sequential behavior, communication, parallelization and abstraction. In [5], temporal logic and the SMoLCS approach are used in the specification of a distributed program semantics.

In [6, 7, 8], *action reification* describes the process of implementing an action of an object at an abstract level by a number of actions of objects at a lower level of abstraction. The specification language is Troll [10], and the formal basis for the refinement of the actions is event structures. Action refinement for event structures is described in [22, 23] A logical framework for the algebraic specificationsof classes at different levels of granularity is proposed in [8].

8 Conclusion

In future work, we plan to develop model-theoretic definitions of our refinement notions. In order to apply the loose approach in a more reasonable setting, we shall also admit concepts that Maude does not provide like a way of disallowing state transitions. Furthermore, we plan to admit observation functions that allow us to make substantial changes to objects and messages in the process of refinement.

Acknowledgements

We are indebted to José Meseguer for clarifying and explaining several issues after a careful reading and, especially, for pointing out the idea of the representation of states by sets of classes. Thanks to Friederike Nickl for carefully reading a preliminary version and to Rolf Hennicker for helpful discussions. Thanks also to the unknown referees for their helpful comments.

This work is part of the DFG project Osidris, contract no. Wi 841/3-1.

References

1. E. Astesiano and G. Reggio. An outline of the SMoLCS approach. In *Mathematical Model for the Semantics of Parallelism*, Lecture Notes in Computer Science 280. Springer-Verlag, 1986.
2. E. Astesiano and G. Reggio. Direct semantics of concurrent languages in the SMoLCS approach. *IBM Journal of Research and Development*, 31(5):512–534, 1987.
3. M. Bidoit and R. Hennicker. A general framework for modular implementations of modular system specifications. In M.-C. Gaudel and J.-P. Jouannaud, editors, *TAPSOFT '93: Theory and Practice of Software Development*, Lecture Notes in Computer Science 668, pages 199–214. Springer-Verlag, 1993.
4. M. Bidoit, R. Hennicker, and M. Wirsing. Characterizing behavioural and abstractor semantics. In *Programming Languages and Systems – ESOP '94*, 1994.
5. G. Costa and G. Reggio. Abstract dynamic data types: a temporal logic approach. In A. Tarlecki, editor, *Mathematical Foundations of Computer Science (MFCS '91)*, Lecture Notes in Computer Science 520, pages 103–112. Springer-Verlag, 1991.
6. G. Denker and H.-D. Ehrich. Action reification in object oriented specifications. In *Proc. ISCORE Workshop Amsterdam, Sep. 94*. Vrije Universiteit Amsterdam, 1994. To appear.
7. H.-D. Ehrich, G. Denker, and A. Sernadas. Constructing systems as object communities. In M.-C. Gaudel and J.-P. Jouannaud, editors, *Theory and Practice of Software Developmen (TAPSOFT '93)*, Lecture Notes in Computer Science 668, pages 453–467. Springer-Verlag, 1993.
8. J.L. Fiadeiro and T. Maibaum. Sometimes "tomorrow" is "sometime" – action refinement in a temporal logic of objects. In D.M. Gabbay and H.J. Ohlbach, editors, *Proc. First Int. Conf. on Temporal Logic (ICTL)*, Lecture Notes in Artificial Intelligence 827, pages 48–66, Bonn, Germany, July 1994. Springer-Verlag, 1994.
9. J. A. Goguen and J. Meseguer. Order-sorted algebra I: Equational deduction for multiple inheritance, overloading, exceptions and partial operations. *Theoretical Computer Science*, 105:217–273, 1992.
10. R. Jungclaus, G. Saake, T. Hartmann, and C. Sernadas. Object-oriented specification of information systems: The TROLL language. Technical Report Version 0.01, Report 91-04, December 1991.
11. P. Lincoln, N. Martí-Oliet, and J. Meseguer. Specification, transformation, and programming of concurrent systems in rewriting logic. In G. Blelloch, K. M. Chandy, and

S. Jagannathan, editors, *Proc. DIMACS Workshop on Specification of Parallel Algorithms*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science. American Mathematical Society, May 1994.
12. D. R. Maeng, J. W. Cho, and K. Ryu. Concurrency and inheritance in actor-based object-oriented languages. *Systems Software*, 20:53–67, 1993.
13. J. Meseguer. A logical theory of concurrent objects. *ACM SIGPLAN Notices*, 25(10):101–115, Oct 1990.
14. J. Meseguer. Rewriting as a unified model of concurrency. Technical Report SRI-CSL-90-02R, SRI International, February 1990.
15. J. Meseguer. A logical theory of concurrent objects and its realization in the Maude language. Technical Report SRI-CSL-92-08, SRI International, July 1992.
16. J. Meseguer. Solving the inheritance anomaly in concurrent object-oriented programming. In O. Nierstrasz, editor, *ECOOP '93 – Object-Oriented Programming*, Lecture Notes in Computer Science 707, pages 220–246. Springer-Verlag, 1993.
17. J. Meseguer and N. Martí-Oliet. From abstract data types to logical frameworks. In this volume.
18. J. Meseguer and T. Winkler. Parallel programming in Maude. In J.-P. Banâtre and D. Le Métayer, editors, *Research Directions in High-Level Parallel Programming Languages*, Lecture Notes in Computer Science 574, pages 253–293. Springer-Verlag, 1992.
19. R. Milner. *Communication and Concurrency.* Series in Computer Science. Prentice/Hall, Int., 1989.
20. F. Orejas, M. Navarro, and A. Sanchez. Implementation and behavioural equivalence: A survey. In M. Bidoit and C. Choppy, editors, *Recent Trends in Data Type Specifications*, Lecture Notes in Computer Science 655, pages 93–125. Springer-Verlag, 1993.
21. B. Salmansberger. Objektorientierte Spezifikation von verteilten Systemen in Maude am Beispiel eines Flughafens. Diplomarbeit, Fakultät für Mathematik und Informatik, Universität Passau, 1993.
22. R. J. van Glabbeek and U. Goltz. Equivalence notion for concurrent systems and refinement for actions. In A. Kreczmar and G. Mirkowska, editors, *Mathematical Foundations of Computer Science (MFCS '89)*, Lecture Notes in Computer Science 379, pages 237–248. Springer-Verlag, 1989.
23. R. J. van Glabbeek and U. Goltz. Refinement of actions in causality based models. In J. W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Stepwise Refinement of Distributed Systems*, Lecture Notes in Computer Science 430, pages 267–300. Springer-Verlag, 1990.
24. M. Wirsing. Algebraic specification. In J. V. Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 675–788. Elsevier (North-Holland), 1990.