FORMAL METHODS FOR DISTRIBUTED PROCESSING

An OO Approach

Edited by
Bowman and Derrick

Contents

1

The Unified Modeling Language

Stuart Kent
University of Kent, UK

A subset of the UML is presented which has been found useful for notating what may loosely be called *specification* models. A model of aspects of the ODP trader case study is developed (a) to provide a vehicle for introducing the notation, and (b) to demonstrate how the notations can be used together in harmony. In the course of the presentation, some issues concerning the precise definition of UML, and its possible future status as a formal method are discussed.

1.1 Introduction

Since the UML first emerged in 1997, its popularity has grown beyond all recognition. It has become the de-facto language for informally modeling object-oriented systems. Although its success can be attributed to a number of factors, one of the most important has been the input of the Object Management Group (OMG) which has led a major exercise to provide the UML with a standard definition. Currently, this is at version 1.3, with new versions already in the pipeline. The standard definition also incorporates a semantics document which aims to give a precise description of the language. By providing users with a standard description of the language, the OMG has encouraged the development of a language that can be shared and understood uniformly throughout industry and academia. The benefits that result cannot be overstated - practitioners, teachers, trainers, tool vendors, and methodologists have now got a single language they can concentrate their efforts on, with the result that significant advances are likely to made in all its aspects. Whilst this is an encouraging start, there are still many problems that need to be addressed before the UMLs true potential can be realized. In our view the six most serious issues are:

(i) **Size.** UML is a collection of notations that have been found to be of practical use to developers of software intensive systems. This encompasses a wide range of notations. In addition, the stereotype mechanism has encouraged modelers to add their own, often ad hoc, extensions to the language. There are also plans to develop various UML profiles, which will collect together packages of stereotypes which have found to be useful. In short, UML is large and growing.

(ii) **Incoherence.** UML has brought together a number of notations from different fields, but has failed to integrate these notations based on a common set of core concepts. For example, it is not clear how state diagrams relate to class diagrams and sequence diagrams.

(iii) **Different interpretations.** UML is interpreted differently by different people. For example, there has been long standing discussion on the meaning of aggregation and composition, the notions of subsystem/model/package are very unclearly specified, there are at least two very different interpretations of state diagrams, and so on.

(iv) **Frequent subsetting.** Our experience is that organizations tend to define their own UML subset – guidelines on which parts to use, which not to use, own definitions of semantics where the standard is unclear, inconsistent or untenable for the organization concerned, and so on. This mitigates against a goal of UML to increase shared understanding amongst developers.

(v) **Constant evolution and extension.** As indicated in the first point the stereotype mechanism is being used (some say *abused* [BGJ99]) to continuously extend the language. Combining this with subsetting and multiple interpretations, the language is really still in a state of evolution and change.

(vi) **Limited tools.** Most commercial tools focus on diagramming, perhaps model exchange, and naive code-generation. Some consistency checks are applied, but these are generally restricted to syntactic checks and applied in an ad-hoc fashion. A small number of tools (e.g. Rose Real Time, Project Technology's Bridgepoint tool) work with executable models notated using UML constructs where possible, and their own constructs where UML does not provide what is needed (e.g. an action language). Because of the executable nature of these models, such tools permit some simulation and testing of models, and usually generate code to a variety of platforms. There are a virtually no automated analysis tools, which allow non-executable models to be simulated/animated, inspected, tested, checked etc., although some prototypes are starting to emerge [RG00, Bol00].

Constructing automated analysis tools for a language requires the language to be formally defined. Of course, the language may obtain a formal definition by virtue of tools being constructed to support it. Those tools which do support analysis of models, such as those cited in (vi), will have to have formalized the fragments of UML they use. Unfortunately, that formalization is usually only implicit in the source code of the tool. Addressing (ii) and (iii) requires a formalization which is explicit and agreed upon.

There are a number of options to consider in trying to address the problem of formalizing the UML. One option is just to treat it as a lost cause, and we reject this out of hand. Another is to provide translations to existing formal languages. This translational approach deals with the semantics of the language – it still requires the syntax to be formalized, which is non-trivial given the diagrammatic nature of the notations. A third option is to define the semantics from the ground up, migrating and adapting ideas and techniques from formal methods as appropriate.

One advantage of the translational approach is that one could make use of tools developed to support the target formal language. However, this is also its weakness. For the practioner, it is important to have analysis tools that give feedback to the engineer in the same language as (s)he is using to construct the model, in this case the UML. A problem with the translational approach to semantics, for example to an existing formal specification language, is that one is then required to work with that language during the analysis phase. At the very least this requires the engineer to learn two languages rather than one, and, presumably, (s)he is more familiar with and prefers to use the UML.

Furthermore, to address (iii), the definition of any aspect of the UML requires agreement; at the very least, the UML community needs to be able to observe the differences between two definitions. This mitigates against the translational approach: the definition needs to be written in a language the is accessible to those who need to agree it, that is people who have the experience to know whether the definition supports the modeling scenarios found in practice. This is probably the reason that the meta-modeling approach to the definition of the UML, where the UML is used to define itself, has proved so popular – anyone with a knowledge of class diagrams can understand the essentials of the definition.

(i), (iii), (iv) and (v) pose another challenge to the formalization of the UML: to develop a language definition architecture that not only allows the language to be defined incrementally, but also permits variations and specializations of the language to be constructed. In essence, find a way of formally defining *families of languages*.

It turns out that this is similar to the problem of defining software (or model) product-lines. The extensions required to UML to support product-lines (chiefly more powerful model-management mechanisms) can be used in the definition of UML itself as a product line. Combine these with a precise subset of UML for expressing object structures and constraints on those structures (essentially class diagrams and OCL), and the result is a language which seems suitable for defining families of languages, both syntax and semantics. This is a variation of the meta-modeling approach to language definition, so has the advantage of being accessible to the OMG. It also seems a very good language within which to define diagrammatic syntaxes. A possible weakness is that the semantics of constructs for expressing dynamic behaviour may be more verbose than if more traditional mathematical syntax was used. A more detailed overview of this approach can be found in [CEF+99, EK99].

An important aspect of this research is that we take UML as it is, only making changes to the language when the formalization process uncovers inconsistencies and errors, or where striking improvements to the language are identified. In particular, the visual flavour which makes it so attractive to engineers should not be lost.

In line with the incremental approach, a first step is to pare the language down to its barest essentials. This chapter describes a subset of UML which, we believe, can be given a precise semantics with little difficulty. Fragments of this semantics have already been developed, together with some tool support [RG98, RG00]. The subset forms the basis of the meta-modeling sub-language itself, and has also proven to be useful in modeling abstract views of network services and their realization onto concrete network configurations such as an IP network. The subset has much in common with the subset used in the Catalysis method [DW98] which has been applied on a number of real projects by its architects. We introduce this subset through a series of sections, with the trader case study used as the running example. The focus of the presentation is on the engineering utility, rather than the formality of the subset, as this, we believe, is the main contribution of the UML.

1.2 Language versus method

UML is a language not a method. It provides a collection of notations that may be used for different sorts of modeling. Its definition gives little advice on what notations are suitable for what kind of modeling, or on what models to build and in what order to achieve a particular goal.

We make use of a general purpose subset of UML which, we hope, captures the core concepts of object modeling, can be applied in different modeling circumstances, and can be given a precise definition. The essence of the method for building a single model is as follows:

(i) Explore the situation to be modeled by exploring *scenarios*, potential traces through the state of the model. Document these with *filmstrips*. Group the scenarios by *use case*.
(ii) Use use cases to separate out the model into overlapping packages (1 package per use case).
(iii) Use the scenarios as a basis for developing the model. Source class diagrams and invariants from the object diagrams in the filmstrips. Source operations and their pre and post conditions from the transitions between object diagrams in the filmstrip. Develop state diagrams showing important transitions and changes of state from an object-centred viewpoint.
(iv) Recurse through steps (i) to (iii) until the model is fit for purpose.

We have built models of software specifications, telecomms networks and services using this method. We are exploring its use in modeling business processes. In the sequel, we build a model of the ODP trader specification.

We have found that the same modeling techniques can be used to model at different levels of abstraction and to build models which specify how an abstract model is *realized* onto a concrete model, for example how an abstract model of network services, expressed in terms of end-to-end virtual connections, and involving different levels of service, is mapped onto a model of an IP network.

The process of software development can be perceived in a similar way: as mappings from abstract to more concrete models, where, here, a model is more abstract than another if the granularity of the operations in the interfaces specified by the model (the public operations on classes) is coarser than the granularity of the interface operations in the more concrete model, and/or the object structures supported by the abstract model are less detailed than the object structures supported by the concrete model.

Note that realization is different to *implementation*, where, once one has reached a concrete model which fixes the granularity of the interface for the actual software that is to be constructed, that model can be further extended with implementation information. If the implementation is in an OO programming language, then sequence diagrams can be used to identify new, private operations required to implement the operations on the interface, and these, in turn, may require their own supporting operations,

classes and so forth. The translation from such an implementation model to, say, a program in Java would be relatively straightforward.

Within this context, the ODP trader specification, modeled in the sequel, is a relatively abstract view of the dialogue between service consumers, service providers and service traders, the intermediaries between consumers and providers. It provides the specification of operations that consumers, providers and traders might perform, which, in turn, requires these three concepts to be treated as objects, in addition to concepts such as service, service offer and so on. A more concrete model, would begin to detail the actual mechanisms in a particular technology (e.g. CORBA) by which these objects would communicate. There would become a point where the translation of a concrete model to the language(s) of the implementation technology would be relatively systematic, though probably less straightforward than to a single Java program.

Our model is loosely based on [ISO96]. When modeling in an industry setting, we would recommend that the model be constructed *with* the domain experts. Any informal documentation should be regarded only as a starting point, and be subject to change as the model is being developed. It should not be regarded as sacrosanct and rigid, otherwise there is little point in developing a more precise model. The end goal is for all descriptions of the system to be consistent with one another.

Finally, we should highlight some deficiencies of our set of modeling techniques.

- UML is weak in its expression of concurrent and real-time behaviour. Some concrete syntax has been inserted into the language, for example asynchronous message passing on sequence diagrams, but the semantics is poorly specified, unclear and confusing. Therefore, the set of techniques we use does not include any of these constructs. Some attempts are being made to resolve this problem, in submissions being prepared in response to the OMG's Request for Proposals on a UML Profile for Scheduling [OMG99b]. We suspect that these submissions will at least bring into focus the detailed problems involved.
- There is, at the time of writing, still no formal definition of the subset used here. However, we are working on providing a formal definition of the subset using the approach proposed in the introduction. Fragments of this subset have been formalized elsewhere and are supported by tools [RG98, RG00].
- To use these techniques successfully on an industrial scale, requires much better tools than are currently available. For example, we would like

tools that check the consistency of models, that assist with the generation of filmstrips from a model and a model from filmstrips, that support composition and separation of models, that support realization of models, that support model templates (patterns), that support model refactoring and so on. Building such tools requires a precise definition for the UML.

1.3 Use cases and packages

A model is recorded as a UML package. Thus all modeling is done within the context of a package. Packages may be constructed by importing other packages. There may be other relationships between packages (e.g. refinement/realisation). Packages are declared and related through package diagrams. Figure 1.1 is the package diagram for the trader case study. The model is recorded as the `Trader` package which has been constructed by extending (importing) two smaller, overlapping packages, one concerned with exporting services to a trader, the other focusing on importing services advertised on a trader. The packages correspond to our chosen primary use cases: `Export Service` and `Import Service`.

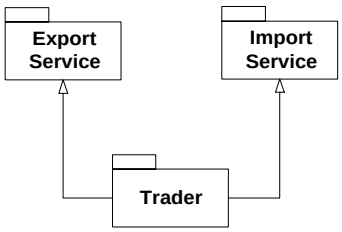

Fig. 1.1. Package diagram for ODP trader case study

The semantics of import/extension between packages in UML is still under discussion – recent submissions to the UML 2.0 RFI (see OMG website for details) criticized the model management aspects of UML 1.3. Our working semantics is taken from Catalysis [DW98], which treats imports a little like class inheritance, where things with the same name in two parent packages are merged, unless they are explicitly renamed on import. Of course, this can give rise to an inconsistent child, and there are some remaining research issues concerning how to merge some elements of a package, such as method contracts, sequence diagrams and state diagrams.

Use cases are a useful discovery technique when modeling. A use case focuses on a particular slice of the behaviour being modeled, related to a particular process in the system being modeled. For the trader case study

we have chosen two use cases, corresponding to the processes of exporting and importing a service, respectively.

There are two styles for characterizing a use case, the goal-oriented style, where the goals of the process that the use case captures are set-out, and the scenario-oriented style, where the process is described is described in terms of particular scenarios, sequences of steps, that the process goes through.

There is no prescribed syntax for use cases in UML, though a number of different ways of presenting use cases have been proposed e.g. [Lar97, Coc00]. Common practice is to state both goals and scenarios informally, with scenarios often written out as dialogues or scripts involving the various actors (including computer systems) involved. UML only prescribes the use case diagram syntax. A use case diagram introduces use cases by name, identifies participants (actors) in those use cases and expresses some (not very clearly specified) relationships between uses cases. Use case diagrams can be useful for giving a 30000ft overview of some kinds of system, typically those that have external, human actors. However, if packages are organized by use case then package diagrams can serve a similar purpose.

In order to make use cases more precise, it is necessary to formalize the goals and scripts that accompany it. Goals can be formalized to a certain extent by building a model which treats the use case as a single action with pre and post conditions written in OCL (the goal is the post condition), supported by appropriate class diagrams, etc. However, this tends to lead to a very abstract model and it is questionable whether the effort is worthwhile. Therefore, we will focus on formalizing use case scripts. This requires identifying the actions involved in the script, the participants of those actions, and how those actions affect the state of the system whenever they are performed. This is no more and no less than building a model. Thus our attention returns to focus on the construction of a model as a UML package.

1.4 Scenarios, filmstrips and scripts

A model of a use case must stipulate, in general, what are the admissable scenarios. One way to achieve this is to explore some example scenarios. These can be documented using filmstrips and scripts. A filmstrip is a sequence of object diagrams (*snapshots*), which accompanies a script identifying what happens at each step. UML does not itself support filmstrips, though object diagrams are defined (they are collaboration diagrams without messages). Scripts are commonly used to describe use cases, though, again, UML does not directly support them. Scripts are most often expressed as informal text,

though they can be expressed more formally as a list of action invocations which can be visualized via a UML sequence diagram.

1.4.1 Filmstrips

A filmstrip for the Import Service use case is given in Figure 1.2. Scripts will be discussed in more detail in Section 1.4.2. For the time-being we provide just the informal script which should assist with understanding the filmstrip.

(i) The scene starts with an importer `i1` and a trader `t1`. `t1` has already had some service offers registered with it. `i1` already has some import policies set up, but not one for use with this trader.
(ii) `i1` creates an import policy to be used with `t1`; `i1` creates a service request
(iii) `i1` creates an import request, which carries with it its own import policy.
(iv) `i1` sends the import request to the trader, and matching service offers are identified.
(v) The selection criteria on the import request are then applied to find the best matching service offer.

Each frame in the filmstrip has been numbered to indicate its position. Due to the formatting limitations, the strip is layed out left to right, top to bottom. Each frame of the filmstrip comprises an object diagram. Objects are rectangles; the class of the object appears after the colon in the label. An optional, arbitrary identity for the object appears before the colon. We have chosen only to name two objects, `i1` and `t1` so that they can be referred to in the script. Links between objects are shown as lines between rectangles – links are instances of associations. Directed links are instances of one-way associations.

The main points of note concerning this filmstrip are:

Frame 1 The service offers for a trader are divided into contexts, where a context is a set of service offers. Contexts may intersect, so may share service offers. The details of service offers are dealt with in the section below which discusses the details of matching. How service offers get created is part of the `Export Services` use case.

Frame 2 The new import policy set up by `i1` for trader `t1` is added to the list of import policies that `i1` might use when issuing import requests. It is difficult to imagine an import policy in isolation from

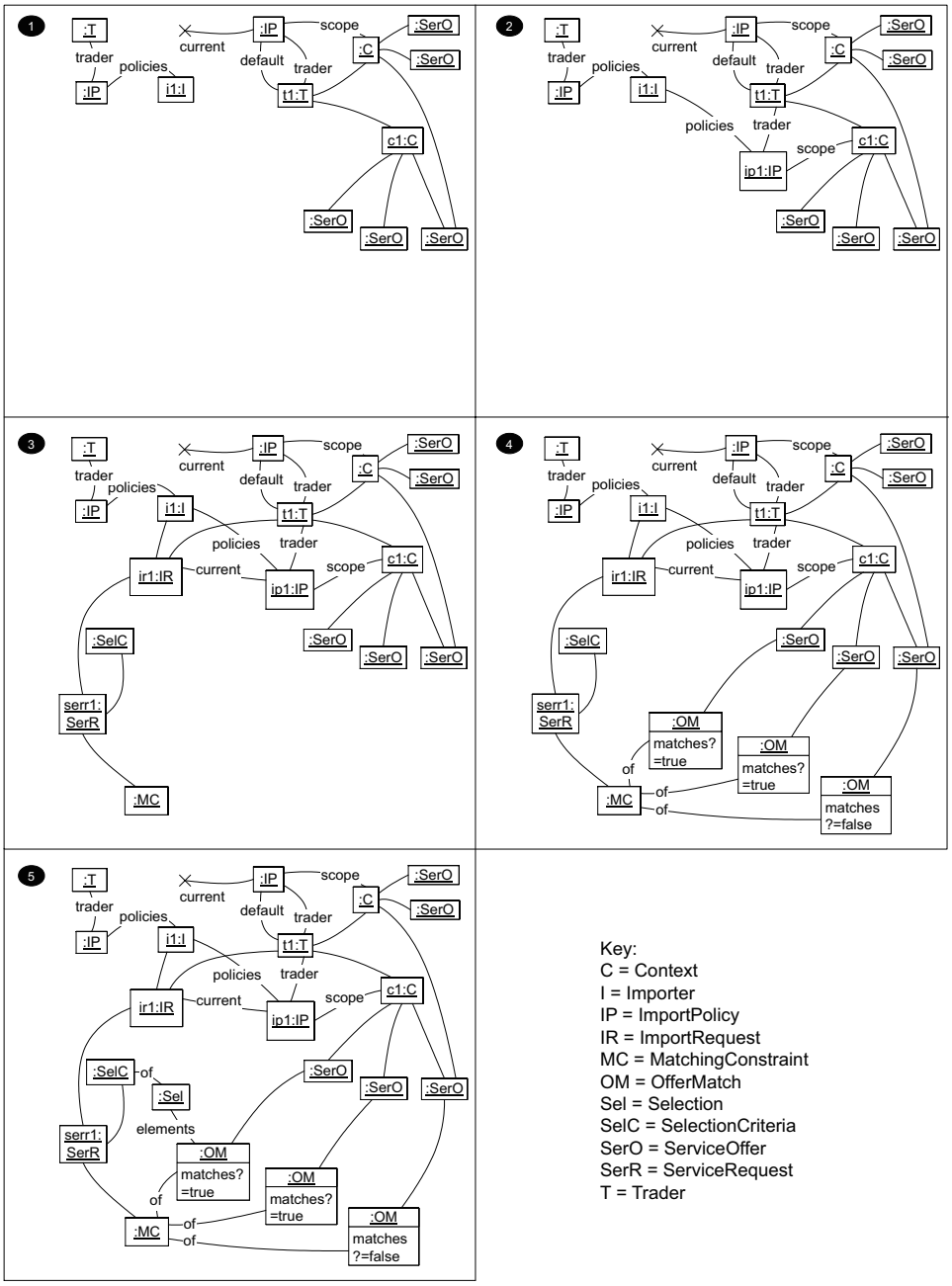

Fig. 1.2. Main filmstrip for import service

a trader, because the policy needs to have some knowledge of the contexts of that trader.

Frame 3 An import request comprises a service request and may come with its own import policy (it does so in this case), which will be one of those identified by the importer to be used with this trader. A service request comes with some selection criteria and a matching constraint. Further details about what a service request comprises will be be dealt with later.

Frame 4 Matches, based on the matching constraint, are created for every service offer in a context belonging to the search scope of the import policy. The import policy used is the one that comes with the import request (this case), or the default policy associated with the trader if no policy comes with the request.

Frame 5 Application of the selection criteria creates a new selection object whose elements are true offer matches that also match the selection criteria. In this case there is only one offer match that matches the selection criteria. The spirit of the text [ISO96] that we are using as the basis for this model, suggests that only a single set of selections are applied, and, indeed, this is the situation illustrated in this filmstrip. However, we observe that one could create many selections for an import request, each derived from different selection criteria. This will be reflected in the class diagram introduced in the next section.

[ISO96] mentions two further complications when matching offers to import requests. We sketch how these could be modeled.

Time limits The idea that searches may have a time limit is mooted. This could be modeled by associating an import request with a time limit, and every `OfferMatch` object with a time stamp. Then when a match is performed there will be exactly one `OfferMatch` object with a time stamp that exceeds the time limit. It could also be required that the amount the time limit is exceeded by is also limited. Any efficient implementation of such a specification would stop as soon as it stamps an `OfferMatch` object with a time that exceeds the time limit. An inefficient implementation might find all matches, then discard all but one of those that exceeds the time limit. So this specification does presume, to some extent, that only sensible implementations will be built.

Search order The idea that contexts could be searched in order (presumably because there is a time limit) is also mooted. This could be modeled by associating an `ImportPolicy` object with a queue of contexts, representing the order in which contexts must be considered.

There is then a constraint on the result of matching that there must be an `OfferMatch` object for every context up to a certain (unspecified) point in the queue and none for contexts thereafter. The last context considered may have some offer matches missing, as it may only have been partially dealt with before the time expired. The offer match with the time stamp exceeding the time limit must be in this last context. Again, any efficient implementation will go through the contexts in the order specified.

The filmstrip in Figure 1.2 indicates the overall structure of the model corresponding to the `Import Service` use case. However, one aspect still needs further clarification, specifically the conditions that make an offer match true or false. This is illustrated by Figure 1.3, which shows one service offer matching an importing request and one which does not.

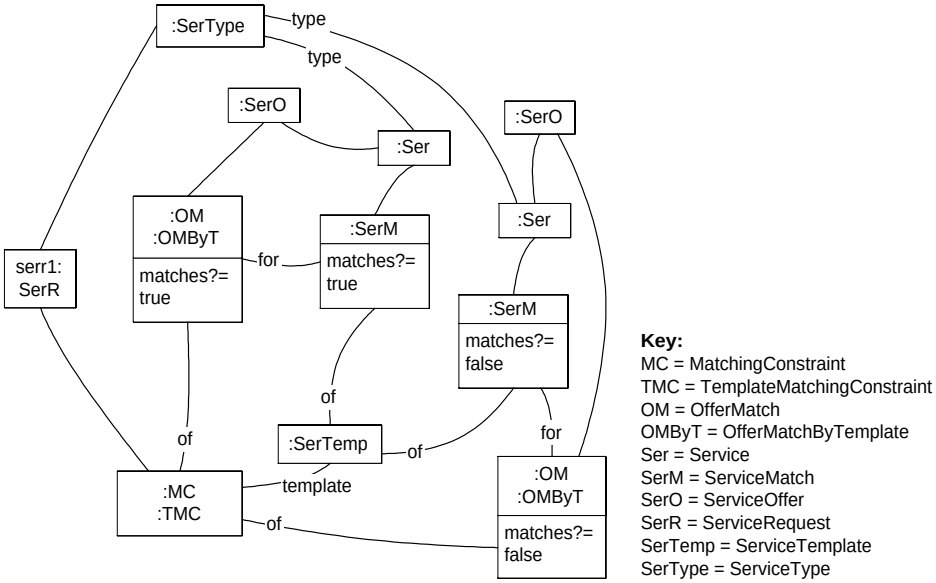

Fig. 1.3. Details of matching

The first condition for a match is that the service type of the service offer must be the same as the service type of the service request. It is for both service offers, in this case. The second condition is dependent on the exact nature of the matching constraint. You will notice that the `MatchingConstraint` object plays a second, more specific role, indicated by it being declared to be of two types. There are likely to be many kinds

of matching constraint, expressed in many different ways (the most general probably being an expression in first order logic). This suggests objects that both play the general role of being a `MatchingConstraint` (so can be attached to a `ServiceRequest`, for example) and also plays the more specific role which governs the specific kind of constraint to be used in the match. The ability of an object to play more than one role is the essence of polymorphism.

In this case, we have identified a `MatchByTemplate` role, where the match is performed by comparing the service of the service offer to the service template associated with the `MatchByTemplate` object. The service template acts as a filter, accepting only those services which match the service template.

You will notice that the results of matches are recorded as objects. There are many ways to model matching. We have chosen this approach as it keeps information about the matches once they have been performed. That then leaves the option of discarding the information, or examining it. For example, if an importer did not get any matches to a request, it may be willing to alter the request to get a match based on information gleaned from the failed matches.

Another approach would have been to define a matching function on the matching constraint object which, when provided with a service offer, would return true or false depending on whether a match was made or not. This approach could be modeled on a class diagram as a query operation on the class or a qualified association. It has been suggested [DW98] that for specification modeling attributes with arguments should be allowed, but these are not currently part of the UML. There is some discussion to be had as to whether attributes with arguments, qualified associations and query operations are different syntaxes for essentially the same concept (query/function/accessor).

It would be possible to continue to add further detail to how matches are made, and to what goes into making up a service offer. For example, the description of ODP trader, from which we have been working, suggests the following:

- A service offer identifies the exporter or provider of the service, its time of registration and its shelf-life, and then the service being offered.
- A service type is comprised of two parts: an interface type and a service property type. Services are comprised of instances of the latter pair of types, i.e. an interface and a service property.
- Service properties (hence their corresponding types) can be composite, in which case they have other properties (which may be composite or atomic) as their parts.

To complete the modeling of this use case we would also need to explore a little more how selections are made.

Filmstrips and further supporting snapshots could be drawn up for the Export Service use case in a similar way. This would focus on the interaction between exporters and traders, and handle actions such as construct service offer, export service offer to specified trader, withdraw service offer, and so on.

As our purpose is not to cover every aspect of the ODP trader case study, but rather to use this case study to illustrate how UML can be used to specify open distributed systems, we will refrain from considering the `Import Service` use case in any more detail, and will not pursue the `Export Service` use case at all in this paper.

1.4.2 Scripts

The script accompanying the filmstrip can be formalized as a sequence of action invocations, which, in turn, can be visualized using a sequence diagram. In practice the formalized script and filmstrip evolve together (and indeed did as the model for this case study was developed during the writing of this paper). We are just presenting the finished article.

The informal script for the ODP trader is repeated below, now interleaved with formal action invocations. The script is visualized by the *sequence diagram* in Figure 1.4.

(i) The scene starts with an importer `i1` and a trader `t1`. `t1` has already had some service offers registered with it. `i1` already has some import policies set up, but not one for use with this trader.

```
start
```

(ii) `i1` creates an import policy to be used with `t1`; `i1` creates a service request

```
createImportPolicy(i1,t1,c1)
```

(iii) `i1` creates an import request, which carries with it its own import policy.

```
createImportRequest(i1,serr1,ip1)
```

(iv) `t1` handles the import request, sent from `i1`, and identifies matching service offers.

```
t1.handleRequest(ir1)
```

(v) The selection criteria on the import request are then applied to find the best matching service offer.

```
ir1.applySelectionCriteria()
```

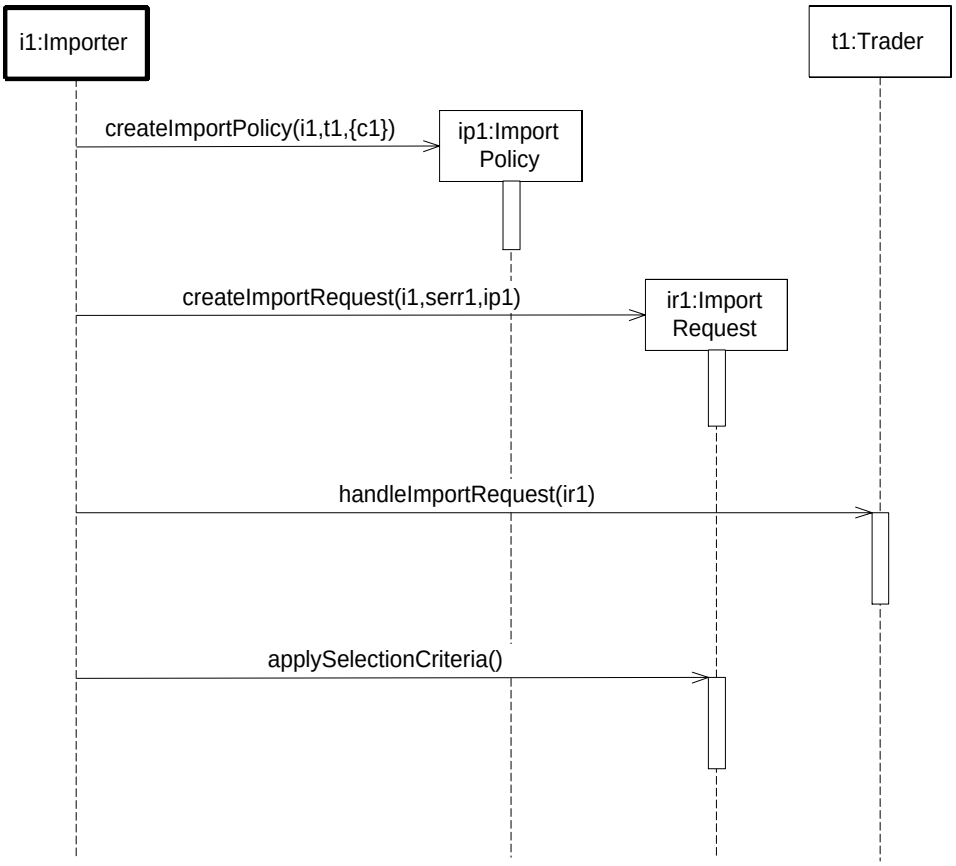

Fig. 1.4. Sequence diagram for ImportService use case

The script is best read in conjunction with the accompanying filmstrip (see Figure 1.2), which provides more information about the objects referred to by name in the script. In the UML, actions are restricted so that they always have a receiver, unless they are creation actions. That is, all actions are assigned to a class, as creation operations – constructors – or as normal operations. UML does not define any textual notation (formal or otherwise) for writing out scripts as we have done. All we have done is write out "instantiations" of the operations, by instantiating the arguments with objects involved in the particular scenario under consideration, using the ubiquitous 'dot' notation to prefix an action with its particular receiver.

There is no generally accepted textual notation for indicating the sender or invoker of an action. However, this is shown on the sequence diagram whose notation is summarized as follows:

- Arrows represent invocations of actions with sender at the source and receiver at the target. These are sometimes called *messages*.
- As in object diagrams, objects are shown as rectangles. The *lifeline* of an object, representing the life of an object over time, is represented by a vertical line protruding downwards from the object.
- An *active object* is shown with a thick border. Active objects can initiate, as well as receive, messages.
- Creation of objects is shown by targeting the creation message directly on the object that is created (e.g. `ip1`) as opposed to the lifeline of the object.

One may consider that identifying senders and receivers of actions is a little premature in an analysis/specification model (this probably does not apply for this particular model). Catalysis [DW98] proposes a slight extension to UML, and to sequence diagrams in particular, which allows the participants of actions to be identified without, necessarily, indicating who the sender and receivers are.

If we produced full models of both the Export Service and Import Service uses cases, it is likely that we would end up with at least three kinds of active object: importers, traders and exporters. It is also likely that these objects would work concurrently and the communication between them would not be wholly synchronous. Although UML does provide some syntax for e.g. distinguishing synchronous from asynchronous messages in sequence diagrams, its semantics is far from clear. Its handling of concurrency is weak.

We have also used the sequence diagram to illustrate a specific scenario. Sequence diagrams can also be used to specify behaviour in general. Our experience is that they can provided the behaviour can be expressed purely in terms of a prototypical instance. For sophisticated behaviours this is usually not the case. For more discussion on this topic see [BGH$^+$98].

An action language for UML is currently under development, in submissions being prepared in response to the OMG's Request for Proposals on a UML Profile for Scheduling [OMG99b]. This may also cure some of the issues surrounding concurrency, at the very least bring into focus the detailed problems involved.

1.5 Structure

There are two key concerns when building a model. Specifying the structure of and constraints on the state of a system, and specifying the dynamic behaviour of that system. This section deals with the structural aspects.

As indicated in snapshots that make up a filmstrip, the structure of the state of the system is recorded as configurations of objects. In order to specify, in general, what states are and are not admitted by a model, it is necessary to specify what object configurations are and are not admissible. This requires a combination of *class diagrams* and *invariants*.

1.5.1 Class diagrams

A class diagram sets limits on the kinds of objects and kinds of links that can appear in an admissible object configuration. The class diagram corresponding to the snapshots appearing in the filmstrip of Figure 1.2 is given in Figure 1.5. A class diagram has two main kinds of element: classes (the boxes) and associations (the lines).

Classes may also have attributes. For example, the class `OfferMatch` has an attribute `isMatch?` of type `Boolean`.

Associations may impose restrictions on the cardinality of links between objects. This is indicated by a numerical range on either or both ends of the association, where `*` represents a range of zero to infinity (there is no constraint on the number of links). For example, the cardinalities of the association ends of the association between `Importer` and `ImportRequest`, indicate that an `ImportRequest` must be associated with exactly one Importer, whereas an Importer may be associated with zero, one or more `ImportRequest` objects.

A further class diagram can be constructed corresponding to the snapshot in Figure 1.3. This is given in Figure 1.6, and illustrate two aspects of class diagramming:

- Inheritance or generalization, shown by the arrow between, for example, the classes `OfferMatchByTemplate` and `OfferMatch`. This means that the child class, at the source of the arrow, has all the features e.g. attributes (and possibly more) as the parent class, at the target of the arrow. Provided this previous statement is carefully defined (see e.g. [LW94]) the upshot is that objects of the child class may behave as if they are objects of the parent class (polymorphism).
- It is fine for classes and associations to be appear in more than one class diagram. If the class diagrams are in the context of *different* packages, then elements are different. If they are in the context of the same package (for example we have tacitly assumed that the class diagrams appearing so far are all in the context of the `Import Service` package), then the

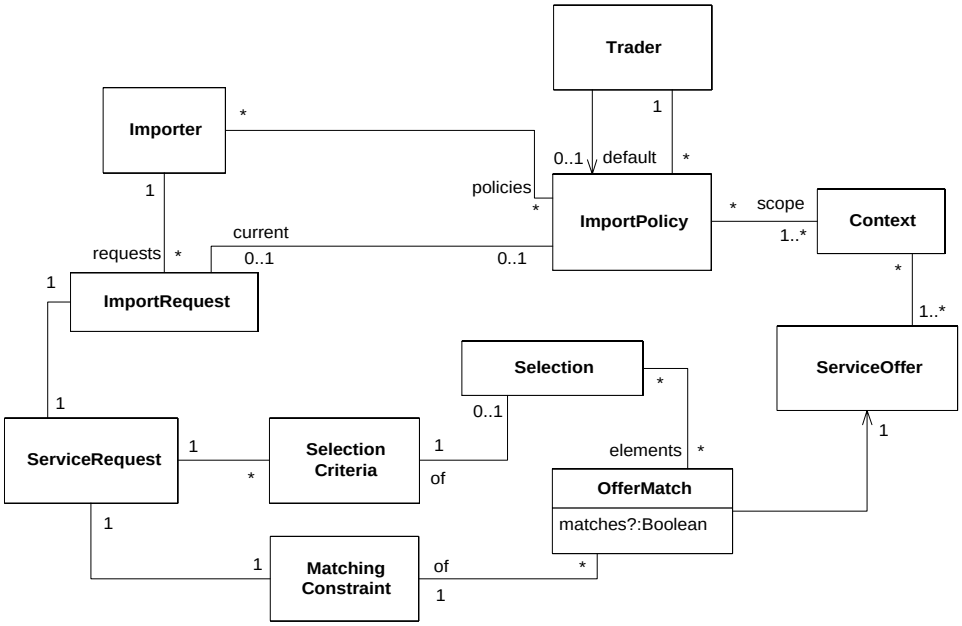

Fig. 1.5. Main class diagram for import service

elements (classes etc.) in that package are obtained by *merging* all the diagrams.

In general, a snapshot is admissible for a particular class diagram, according to the following rules:

- The type of every object appearing in the snapshot is a class in the class diagram.
- Every link in the snapshot corresponds to an association between classes in the class diagram. A link corresponds to an association if its label or labels at each end correspond to the labels at each end of the association and the objects are from classes connected by the association.
- Links do not flout cardinality constraints on associations. (A detailed and precise specification of this can be found in [KH99].)
- Any attribute mentioned in an object on the snapshot must be declared in the class for that object. The value given to the attribute must be of the type declared for that attribute in the class diagram.

These rules can be used to guide the construction of a class diagram from a snapshot. The first rule means that one puts a class in the class diagram for every type of object in the snapshot. The second rule means that one

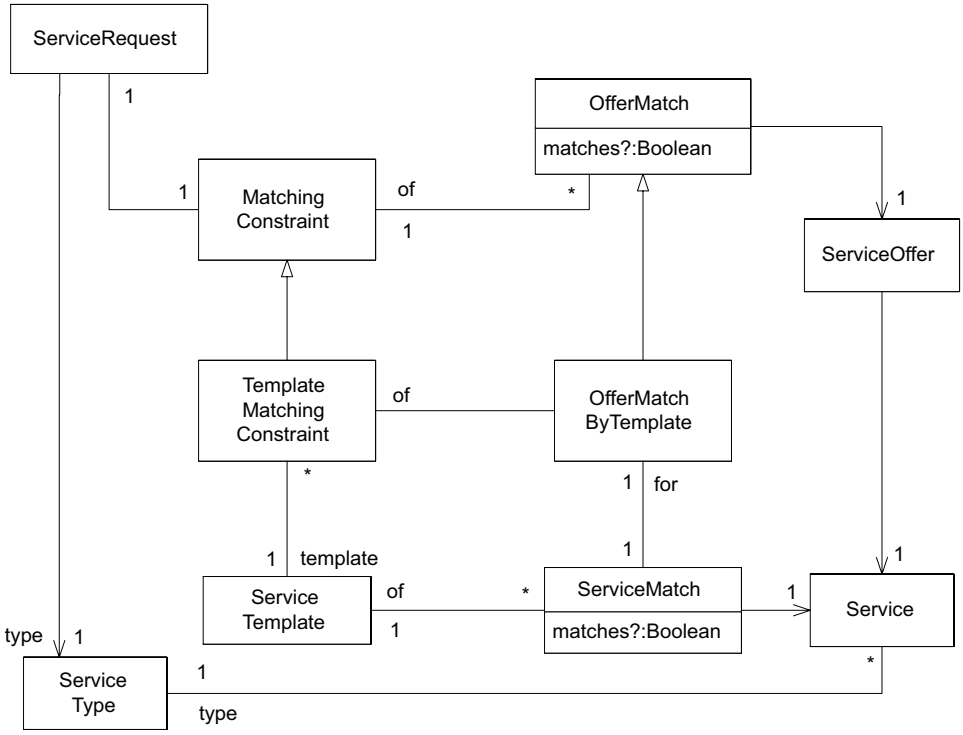

Fig. 1.6. Matching import requests to services: class diagram

puts an association in the class diagram for every different kind of link in the snapshot, where a link x is of the same kind as another y, if x connects the same types of object as y and has the same labels at each end, where ends are matched on the type of object. And so on for the other rules.

Of course, these rules do not specify completely what must appear in the class diagram – specifically they do not stipulate exactly what the cardinality of associations should be. Indeed, what tends to happen is that, as the class diagram is drawn, new concepts emerge causing new versions of the snapshots to be elaborated, which in turn might reveal other changes, and so on. Also, an experienced modeler may well construct snapshots in his or her head, without ever making them explicit. Nevertheless, they can always be made explicit to anyone challenging the model, in supporting documentation intended to explain the model, in communicating to domain experts, and to help understand particularly tricky behaviour.

There is some debate as to the relationship between attributes and associations. A popular view is that an association can be reduced to a

pair of attributes with an additional constraint. For example, the association between the classes `ServiceRequest` and `SelectionCriteria` could be thought of as a pair of attributes `selectionCriteria:SelectionCriteria` and `serviceRequest:ServiceRequest` of `ServiceRequest` and `SelectionCriteria`, respectively, with the additional constraint that each attribute is the inverse of the other (such a constraint could be written as an invariant – see next section – if so desired). Attributes tend to be used instead of associations when the type of the attribute refers to a basic value type, such as `Boolean` or `Integer` rather than a class.

We have only shown the most basic form of association. There are other kinds of association, in particular aggregates and qualified associations. Qualified associations are akin to attributes (functions) with arguments, such as described in [DW98]. We will not enter into a discussion of aggregation here. Suffice it so say that there is still some debate on this topic [HSB99].

In other forms or modeling, such as design/implementation modeling, the situation gets complicated by the introduction of operations on classes, in particular query operations. In this form of modeling one is concerned with distinctions such as: whether a result-returning query is stored or calculated; or whether an operation or attribute/association-end is visible or not outside a class (public or private). Most modelers at this level tend to treat attributes and associations as (private) storage, and define query operations to access the data stored within. One must be slightly careful if adopting this approach. For example, if one is deriving a design from a specification model one has to be careful to remember that attributes and associations at that level may well correspond to calculated queries (hence operations) in the design model. This means that a developer must carry around two different interpretations of the same construct (associations and attributes). One must also decide how qualified associations (naturally thought of as functions) should be interpreted in the design model, given that associations are assumed to correspond to stored data: As arrays? As dictionaries?

On balance, our preferred mental model is one where attributes and query operations are treated as the same thing–a query, and an association as a pair of queries. A qualified association can then be thought of as a query with arguments. When design/implementation modeling, a distinction can be made between whether a query is stored or calculated, or whether it is public or private. If one chooses to have the default rule that, unless stated otherwise, all attributes and associations will be treated as stored and private then that is quite acceptable.

These issues may seem minor, but can be very confusing to modelers,

and what tends to happen is that different organizations, often different individuals, construct their own interpretations which only serves to block shared understanding (a goal of the UML). They are the kinds of issues that will only be fully sorted out when the UML has an agreed, precise definition. On the other hand, if a team is prepared to agree on a common interpretation which need only be documented informally, then they are issues which need not get in the way of the modeling activity.

1.5.2 Invariants

The class diagram can not express all constraints that one would wish to impose on the structure of admissible object configurations. Invariants are constraints that all admitted configurations must satisfy. In UML, the Object Constraint Language (OCL) [WK98] has been defined to allow these constraints to be written in a precise syntax.

Some examples of invariants from the trader case study are given below:

(i) *The trader associated with the default import policy of a trader is the trader itself*

```
context t:Trader inv:
   t.default->isEmpty or t.default.trader=t
```

The preamble **context** `t:Trader` **inv**: indicates that the constraint which follows applies to all objects `t` of class `Trader`. `t.default` returns the set containing the object(s) found by navigating the `default` link(s) from `t`. `->isEmpty` indicates that this set is empty. `t.default.trader` returns the set of object(s) obtained by navigating first the `default` link(s) from `t` and then the `trader` link(s) from all the objects found through the first step of the navigation. For the clause `t.default.trader=t` to be true, that set must contain only a single object which is `t`. `or` is the standard logical connective.

(ii) *The service offers matched for the current request being handled by an import policy are within the scope of that policy.*

```
context ip:ImportPolicy inv:
   ip.scope.serviceOffers->asSet->containsAll(
      ip.current.serviceRequest.matches.serviceOffer->asSet)
```

This invariant illustrates navigation expressions that return collections with more than one element. Any navigation expression that

spans more than one association returns a bag, by default. Thus `ip.scope.serviceOffers` returns a bag. This expression is evaluated as follows. Navigating `scope` from `ip` returns a set of `Context` objects. Navigating `serviceOffers` from each member of this set, results in a set of `ServiceOffer` objects. The bag is created by merging these sets, being careful to keep repeated items. In this case, there are likely to be repeated elements as contexts may share service offers. Similarly, `ip.current.serviceRequest.matches.serviceOffer` returns a bag. `->asSet` coerces a bag into a set.

(iii) *For an offer match to be true, the service type of the service offer must be the same as the service type of the service request.*

```
context om:OfferMatch inv:
   om.matches?=true implies
      om.of.serviceRequest.type=om.serviceOffer.service.type
```

(iv) *For a* `OfferMatchByTemplate` *to be true, the service template must match the service of the service offer.*

```
context omt:OfferMatchByTemplate inv:
   omt.matches?=true implies
      omt.serviceMatch.matches?=true
```

(v) *The service template for a service match must be the service template for the template matching constraint of the* `OfferMatchByTemplate` *which the service match is for.*

```
context sm:ServiceMatch inv:
   sm.matches?=true implies
      sm.template=sm.for.of.template
```

Combined with the constraint, imposed by the class diagram in Figure 1.6, that the matches of a template matching constraint are always template offer matches, the last three invariants ensure that when the appropriate matches are constructed (see section 1.6 on dynamic behaviour), they will be designated true or false as appropriate. Of course we have not stipulated the detailed circumstances under which a service type matches a service; as indicated earlier that would require further investigation into the detailed structure of services and service templates.

These last two invariants also illustrate how, with inheritance, we are able to push specific behaviour onto the more specific classes. We are at liberty to create a number of other subclasses, with different invariants, capturing

different variants of matching constraints. This not only provides a way of separating out the behaviour into appropriate chunks, but also allows other behaviour to be specified which is decoupled from the specific variations. So, in section 1.6 on dynamic behaviour, we are able to specify the result of performing the action `handleRequest`, which results in the creation of the required matches for a request, only referring to the `MatchingConstraint` and `OfferMatch` classes; no mention of their subclasses is made.

The last invariant could have been written in a number of different ways, depending on the class to which the invariant is tied, the class that appears in the **context** part. Other candidate classes are `TemplateMatchingConstraint` and `TemplateOfferMatch`. This illustrates a problem when writing invariants, knowing which class is the best place to put the invariant. A factor which influences this decision is the coupling of classes: if an invariant means unnecessary coupling between classes then this will mitigate against reuse of the owning class in other models. In this case, the three classes come "as a package" and are already quite tightly coupled, so it probably does not matter where the invariant is placed. More practical application of writing OO specifications with invariants and the like is required to identify a set of guidelines, or patterns, to support the practicing modeler.

Recently there has been considerable work on formalizing and improving OCL. For pointers to some of this work see [pUM00] and [Ric00].

A visual notation, called *constraint diagrams* has been defined for expressing constraints, though this is not (yet) part of the UML, though it is compatible with the UML – it may be regarded as a visual alternative to a (sub-language) of OCL. This language was first introduced in [Ken97], and has been further applied to the expression of action contracts [KG98]. It is currently undergoing revision as it is defined formally [GHK99, HMTK99], and work is continuing on making it a practical technique to be used in harmony, not in conflict, with other approaches to writing constraints [KH99].

1.6 Dynamics

In the modeling context we have chosen (abstract specification), dynamic behaviour is captured in terms of pre/post conditions on operations. These can be (partially) visualized using state diagrams.

We illustrate the use of OCL to express pre/post conditions with an example taken from the import service use case. Section 1.4.2 identified a number of action based on the script for the import service use case. One of these actions was `handleRequest`, which, if we examine the filmstrip in Figure 1.2, has the effect of creating an offer match between the import request and

each service offer of the trader handling the request, as governed by the import policy used. In our simplified version, the import policy just identifies a context from which the set of service offers are drawn. The specification of this operation is given below:

context `Trader::handleRequest(ir:ImportRequest):`
pre: *The import request has an import policy, or the trader has a default policy. If the import request has a policy, then the trader for the policy is* `self`*. No attempt has been made to handle this request, or any previous attempt has been cleared.*

```
let policy=if ir.importPolicy->notEmpty then
   ir.importPolicy else self.default in
      policy->notEmpty and policy.trader=self
and ir.serviceRequest.matchingConstraint.oclInState(matchCleared)
```

post: *The policy of the request has been set to be the trader's default policy if the request has no policy. The import request has been matched against service offers according to the request's policy.*

```
ir.importPolicy@pre->isEmpty implies ir.importPolicy=self.default
and let offers=ir.importPolicy.scope.serviceOffers@pre->asSet in
   let matches=ir.matchingConstraint.offerMatches@pre in
      offers->size=matches->size and matches.serviceOffer=offers
and ir.serviceRequest.matchingConstraint.oclInState(matchCleared)
```

This time the **context** preamble identifies the action concerned together with any arguments. The pre and post conditions illustrate a number of additional OCL constructs:

- `let` and `if_then_else_` expressions, as found in formal specification languages such as VDM.
- `@pre` in a post-condition which allows reference to the state when the action is invoked. One could argue that, in this case, `@pre` is not necessary as e.g. `ir.policy` should be the same in both states. However, OCL does not have any notation for expressing frame rules, which is hard in OO models due to the ability to navigate across object structures. One also can make no assumptions about other actions which may occur at the same time as this action, and which may affect objects referred to in the action spec. Our use of `@pre` is therefore a safety measure. The expression of frame rules in OCL is an open issue; we are not sure that a satisfactory solution yet exists for OO specification modeling. Some sources of inspiration might be JML [LB99, LBR99].

- `->size` returns the size of the collection to which it is applied.
- `oclInState(_)` used to express whether an object is in a particular state or not, where here we are referring to states in state diagrams.

The use of `oclInState(_)` in the specification of `handleRequest` must be supported by a *state diagram*. This is given in Figure 1.7.

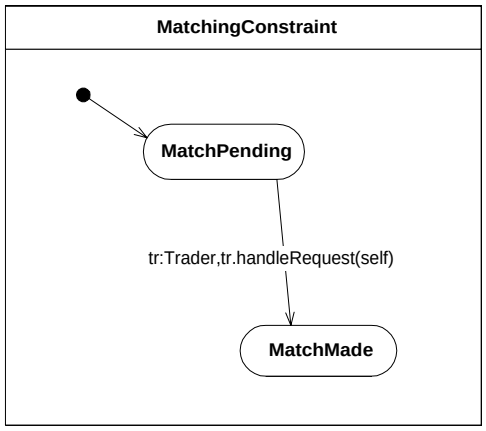

Fig. 1.7. State diagram for MatchingConstraint class

A state diagram applies to a class. It specifies (or visualizes) aspects of the dynamic behaviour of any object of that class. States are shown by rounded rectangles, transitions between states by arrows.

An interpretation for this diagram is as an abstraction of the state space and of dynamic behaviour expressed using pre/post conditions. That is, when a trader performs the action `handleRequest`, with the request associated with the matching constraint under consideration as argument, and the matching constraint has no matches with service offers, then the result will be that the matching constraint has made a match to service offers. This captures a fragment of the behaviour expressed more completely by the pre/post conditions above.

The navigation expression to identify the action is non-standard UML, but is appropriate if state diagrams are to be used for specification purposes.

Under this interpretation state diagrams can be integrated with class diagrams and OCL constraints. One model is to view states as dynamic subclasses of the class they are assigned to in state diagrams: objects belonging to a dynamic class may move to a different (dynamic) class, and vice-versa. In which case `oclInState(_)` is just syntactic sugar for `oclIsKindOf(_)`, with a dynamic class as argument, as opposed to a static class.

It is useful to use invariants to tie states to the detailed state space of an object. For example:

Definition of a matching constraint having no matches.

```
context mc:MatchingConstraint inv:
   mc.oclInState(matchPending)=mc.offerMatches->isEmpty
   and mc.oclInState(matchMade)=mc.offerMatches->notEmpty
```

These invariants work, as the cardinality of associations on the class diagram in Figure 1.5 ensures that a context will at least identify one service offer to attempt a match against. They make the last conjunct of the post condition of `handleRequest` redundant. Some of the invariants given in Section 1.5.2 could be made more transparent by rewriting parts to involve these states.

The interpretation of state diagrams used here is not the one that is detailed in the UML 1.3. standard which does not recognize the value of state diagrams for modeling at the specification level. The standard interpretation is one where state diagrams are viewed in an operational rather than declarative way: as a specification of the order in which actions must occur to the point where the state diagram can be executed, rather than a specification of the how actions behave in certain situations. (However, it is recognized that the latter may, as a side effect, constrain the order in which actions occur.) A number of responses to the recent UML 2.0. RFI [OMG99a] have argued the case for an interpretation of state diagrams suitable for specification modeling. This interpretation is similar to that used in Catalysis [DW98].

1.7 Future

This paper has introduced a subset of UML which potentially can be used to produce precise object-oriented specifications. The subset remains to be formalized, although there are already tool-supported formalization of part of it: for example [RG98, RG00] provides a tool supported formalization of class diagrams and OCL constraints. There is now even a commercial tool [Bol00] which does much the same.

The author is currently (July 2000) engaged in work as part of the precise UML (pUML) group [pUM00] to rearchitect the UML as a family of languages. This work is influencing the revision of UML within the OMG, to the point that there is now a strong likelihood of a request for proposals for UML 2.0 that will have the goal of our work at its heart. Our approach

to formalization is a variation of the meta-modeling approach to language definition, which has been adopted by the OMG. Essentially we define a family member of UML (a meta-modeling language – MML) that is then used to define itself and other family members. The MML is grounded by an external definition, which in our case is the provision of a tool to support the various features of the language. A key aspect of this approach is that we are able to define, in MML, concrete syntax (both graphical and textual), abstract syntax *and* semantics. An overview of this approach can be found in [CEF+99, EK99].

By recognizing UML as a family of languages, the job of defining domain-specific subsets of UML, and/or introducing new notations should become more systematic and precise. Especially if, as intended, there is a tool-supported framework for defining new family members, by extending and specializing existing language fragments, including a process for signing off, standardizing and evolving language definitions.

This could benefit those working in the distributed systems domain, by providing a platform to support the definition of languages appropriate for modeling in that domain. For example, the Common Information Model (CIM) standard under development under the Distributed Management Task Force of the IETF [DMT99], which is a standard approach to modeling in support of intelligent network management, makes use of a language that is essentially UML class diagrams with its own specializations. Similarly, proposals for using UML as a language for notating Enterprise Viewpoint models in ODP [AM99, Lin99], generally make use of a subset of UML specialized with stereotypes. Of course, as is the way with stereotype usage in UML [BGJ99], the intended meaning of the specializations is, at best, informally explained. We would fully expect these languages to be definable as part of the UML family, and there are clear advantages in doing this. In particular, effort put in for one domain can often be reused in other domains. Thus if one takes the trouble to precisely define a constraint language for use with object models, say, in software specification, that constraint language can be reused in modeling networks, services, policies and the like. If it turns out that the language needs to be extended, and/or the concrete syntax is not appropriate for the domain in question, then the appropriate extensions to the base language and/or a new concrete syntax can be provided. On the other hand it should still be possible to use tools, training materials and so on, that support the base language, with the extended/specialized language. Thus if only a different concrete syntax is required then any semantic checking tools will be unaffected. The purpose of the framework we

are developing is to manage such language development and evolution in a systematic way.

To conclude, we return to one aspect of UML that we identified as a weakness in the subset we have chosen: concurrency and real-time. Part of the problem is that the 1.3 documentation is so ambiguous and contradictory [KER99], that it is hard to know where to start. This might be remedied somewhat in the forthcoming submission to the Scheduling Profile RFP [OMG99b]. The core of this submission is an attempt pin down an "action semantics" which directly addresses issues of concurrency and real-time. This will identify many of the problems and suggest solutions. However, the submission will still be informal in nature, in the same style as the UML 1.3 standard. One of the goals of the proposed rearchitecting of the UML will be to rework this submission into a more rigorous and organized definition. This will make it much easier to see where existing research results in concurrency and real-time could be used to further improve the UML in this area.

Acknowledgements

Thanks to Andy Evans and John Derrick for feedback on early drafts of this paper. This work what partly supported by the EPSRC grant GR/M02606.

Bibliography

[AM99] J. Øyvind Aagedal and Z. Milosevic. Odp enterprise language: Uml perspective. In C. Atkinson, editor, *Proc. of The 3rd International Conference on Enterprise Distributed Object Computing*. IEEE, 1999.

[BGH+98] R. Breu, R. Grosu, C. Hofmann, F. Huber, I. Krüger, B. Rumpe, M. Schmidt, and W. Schwerin. Exemplary and complete object interaction descriptions. *Computer Standards & Interfaces*, 19(7):335–345, November 1998.

[BGJ99] S. Berner, M. Glinz, and S. Joos. A classification of stereotypes for object-oriented modeling languages. In R. France and B. Rumpe, editors, *UML'99 - The Unified Modeling Language. Beyond the Standard. Second International Conference, Fort Collins, CO, USA, October 28-30. 1999, Proceedings*, volume 1723 of *LNCS*, pages 249–264. Springer, 1999.

[Bol00] BoldSoft. ModelRun case tool (beta). Available from http://www.boldsoft.com, July 2000.

[CEF+99] A. Clark, A. Evans, R. France, S. Kent, and B. Rumpe. The puml response to the omg uml 2.0 rfi. Available from http://www.cs.york.ac.uk/puml, December 1999.

[Coc00] A. Cockburn. *Writing Effective Use Cases*. Addison-Wesley, 2000. To appear. Draft available from http://members.aol.com/acockburn/.

[DMT99] DMTF. The common information model (CIM) specification v2.2. Available from http://www.dmtf.org/spec/cims.html, June 1999.

[DW98] D. D'Souza and A. Wills. *Objects, Components and Frameworks With UML: The Catalysis Approach.* Addison-Wesley, 1998.

[EK99] A. Evans and S. Kent. Core meta-modelling semantics of UML: The pUML approach. In Robert France and Bernhard Rumpe, editors, *UML'99 - The Unified Modeling Language. Beyond the Standard. Second International Conference, Fort Collins, CO, USA, October 28-30. 1999, Proceedings*, volume 1723 of *LNCS*, pages 140–155. Springer, 1999.

[GHK99] J. Gil, J. Howse, and S. Kent. Formalizing Spider Diagrams. In *Proceedings of IEEE Symposium on Visual Languages (VL99).* IEEE Computer Society Press, December 1999.

[HMTK99] J. Howse, F. Molina, J. Taylor, and S. Kent. Reasoning with Spider Diagrams. In *Proceedings of IEEE Symposium on Visual Languages (VL99).* IEEE Computer Society Press, December 1999.

[HSB99] B. Henderson-Sellers and F. Barbier. Black and white diamonds. In Robert France and Bernhard Rumpe, editors, *UML'99 - The Unified Modeling Language. Beyond the Standard. Second International Conference, Fort Collins, CO, USA, October 28-30. 1999, Proceedings*, volume 1723 of *LNCS*, pages 550–565. Springer, 1999.

[ISO96] ISO/IEC JTC1/SC21. ODP Trading Function. Draft Rec. X.950:1, ISO/IEC DIS 13235, May 1996.

[Ken97] S. Kent. Constraint Diagrams: Visualizing Invariants in OO Modelling. In *Proceedings of OOPSLA97*, pages 327–341. ACM Press, October 1997.

[KER99] S. Kent, A. Evans, and B. Rumpe. UML Semantics FAQ. In *ECOOP'99 Workshop Reader.* Springer Verlag, LNCS, December 1999.

[KG98] S. Kent and Y. Gil. Visualising Action Contracts in OO Modelling. In *IEE Proceedings: Software*, number 2-3 in 145, pages 70–78, April 1998.

[KH99] S. Kent and J. Howse. Mixing visual and textual constraint languages. In R. France and B. Rumpe, editors, *UML'99 - The Unified Modeling Language. Beyond the Standard. Second International Conference, Fort Collins, CO, USA, October 28-30. 1999, Proceedings*, volume 1723 of *LNCS*, pages 384–398. Springer, 1999.

[Lar97] C. Larman. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design.* Prentice Hall, 1997.

[LB99] G. Leavens and A. Baker. Enhancing the pre- and postcondition technique for more expressive specifications. In J. M. Wing, J. Woodcock, and J. Davies, editors, *FM'99 – Formal Methods. World Congress on Formal Methods in the Development of Computer Systems, Toulouse, France.*, volume 1708 of *LNCS*. Springer-Verlag, September 1999.

[LBR99] G. Leavens, A. Baker, and C. Ruby. JML: A notation for detailed design. In H. Kilov, B. Rumpe, and I. Simmonds, editors, *Behavioral specifications for businesses and systems*, chapter 9, pages 175–188. Kluwer Academic Publishers, Norwell, MA, September 1999.

[Lin99] P. Linington. Options for expressing ODP enterprise communities and their policies by using UML. In C. Atkinson, editor, *Proc. of The 3rd International Conference on Enterprise Distributed Object Computing.* IEEE, 1999.

[LW94] B. Liskov and J. M. Wing. A behavioural notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, 1994.

[OMG99a] Analysis and Design Task Force of the OMG. UML 2.0 RFI. Available from http://www.omg.org, 1999.

[OMG99b] Analysis and Design Task Force of the OMG. UML profile for scheduling RFP. Available from http://www.omg.org, 1999.

[pUM00] The precise UML group. http://www.cs.york.ac.uk/puml, July 2000.

[RG98] M. Richters and M. Gogolla. On formalizing the UML Object Constraint Language OCL. In Tok Wang Ling, Sudha Ram, and Mong Li Lee, editors, *Proc. 17th Int. Conf. Conceptual Modeling (ER'98)*, volume 1507 of *LNCS*, pages 449–464. Springer, 1998.

[RG00] M. Richters and M. Gogolla. Validating UML models and OCL constraints. In A. Evans and S. Kent, editors, *The Third International Conference on the Unified Modeling Language (UML'2000), York, UK, October 2-6. 2000, Proceedings*, LNCS. Springer, 2000.

[Ric00] M. Richters. The university of bremen UML bibliography. http://www.db.informatik.uni-bremen.de/umlbib/, July 2000.

[WK98] J. Warmer and A. Kleppe. *The Object Constraint Language: Precise Modeling with UML*. Addison-Wesley, 1998.

Index