Distributed Abstract Data Types

Gian Pietro Picco[1], Matteo Migliavacca[1],
Amy L. Murphy[2], and Gruia-Catalin Roman[3]

[1] Dip. di Elettronica e Informazione, Politecnico di Milano, Italy
[2] Faculty of Informatics, Univ. of Lugano, Switzerland
[3] Dept. of Computer Science and Engineering, Washington Univ. in St. Louis, USA

Abstract. In this paper we introduce the concept of *Distributed Abstract Data Type* (DADT), a new programming language construct specifically designed to support the development of distributed, context-aware applications. Through a DADT instance, a program gains access to both aggregate and individual ADT instances throughout the system. The semantics of distribution and sharing is specified by DADT operations using dedicated and novel programming constructs. These include also the ability to declare at run-time partitions, called *views*, over the target ADT instances based on their application or context state, to restrict operation scope.
Interestingly, DADT constructs can be used to specify not only application data, but also the space where it resides. This leads to a uniform treatment of the data and space concerns, simplifying the development of context-aware applications and providing the programmer with considerable flexibility and expressive power. We argue that DADTs are amenable to incorporation in existing object-oriented programming languages, as supported by our prototype implementation.

1 Introduction

Modern distributed computing places new demands on application programmers, not only because of the increasing scale, decentralization, and dynamicity, but also because of novel application requirements demanding control and visibility of the physical space where the application executes. A paradigmatic example are the applications falling under the umbrella of pervasive, ubiquitous computing, and ambient intelligence. In these settings, programmers must derive information through simultaneous access to a plethora of devices, sensors, or application objects, dispersed in the environment, e.g., to gather aggregated information to base further decisions upon. Furthermore, the way this processing is organized is often dependent on application or contextual information—which is itself distributed (e.g., the mutual position of the entities involved, their residual power, or their application state).

Despite the popularity of the field, the models and systems available often treat the physical space—or context—where the application executes as something external to the application, therefore requiring dedicated and specialized constructs increasing the programming effort. Similarly, the abstractions for dealing with distribution are usually quite primitive, often forcing the programmer to deal explicitly with the details of individual remote accesses.

```
datatype Sensor {
  data:
       int sensorType;
      bool isActive;
    double value, resolution;
  operations:
    double read();
      void reset();
}
```

Fig. 1. ADT interface for a simple sensor.

In the approach we describe here, we provide programming constructs to simplify the access to distributed state, by explicitly taking into account physical space as well. We accomplish this by extending the well-established programming notion of abstract data type (ADT) into a *distributed abstract data type* (DADT). The state of multiple ADTs disseminated in the system is made available collectively through the interface of a DADT, whose behavior in terms of distribution is defined by the programmer with appropriate and dedicated constructs. Inside the DADT interface, properties over ADT instances enable the definition of partitions, called *views* over the distributed state, which can be used to dynamically restrict the scope of distributed operations according to the application needs. Interestingly, the definition of DADTs and views is not limited to application objects, as in conventional programming language, but is extended also to the representation of space. Our model provides full integration of the spatial and data concerns involved in the definition of context under a single, unified programming framework revolving around the notion of DADT. Data and space become two different, and yet intimately related, perspectives enabling the distributed manipulation of application entities.

The paper is structured as follows. Section 2 introduces a simple and yet realistic example, used throughout the paper for illustration purposes. Section 3 introduces the reader to the basic concepts of our DADT model. Section 4 discusses the constructs enabling the programming of distributed access, while Section 5 introduces the notion of a DADT *view* enabling the programmer to specify declaratively and dynamically the scope of an operation. Section 6 reports about the design and implementation of our proof-of-concept prototype. Section 7 places DADTs in the context of related work and Section 8 ends the paper with brief concluding remarks.

2 A Reference Example

This section introduces an example used throughout the paper to make the concepts we present more concrete. Imagine an environment where several sensors are deployed to report about some physical parameter (e.g., temperature). A monitoring station may access the sensors to aggregate raw data (e.g., computing the average temperature), or reset the sensor in case of problems.

Interaction with a sensor is naturally modeled by an ADT, as shown in Figure 1. In a conventional setting, this interface is used to access a sensor (i.e., a `Sensor` instance) at a time. Even a simple computation such as determining the average sensed value entails a considerable programming effort. The programmer must have an explicit notion of the sensor configuration, or at least of their identity, to invoke the `read` operation remotely on each of them, fetch the corresponding value, and compute the average locally.

It can be argued that a skilled programmer would probably define a new ADT, e.g., `SensorProxy`, that embodies this distributed processing. However, currently available languages do not provide any syntactic or semantic feature to keep track of the fact that the two ADTs are conceptually related. Moreover, in the absence of proper middleware facilities, the behavior of this latter ADT (`SensorProxy` in our case) must be redefined from scratch every time. Also, the programmer does not retain any control over the portion of the system that should be affected by an operation. Thus, for instance, there is no easy way to compute the average only on sensors that are within a given range.

In the remainder of the paper, we use our sensor example to introduce the novel notion of *distributed abstract data type* (DADT) which allows the programmer to define a new, distributed ADT as a refinement of the original one, define how it is possible to restrain the set of DADT instances involved in an operation, and therefore provide an expressive solution to the problems we just outlined.

3 Basic Concepts

The driving motivations for our work are to manage distribution and context-awareness while simultaneously minimizing the invasiveness for the programmer. Therefore, we decided to cast our ideas into the notion of *abstract data type*, as it represents a well-understood and commonly used programming concept, and is general enough to allow us to present our novel constructs without being distracted by the idiosyncrasies of a specific programming language or model.

This latter aspect is reflected in the presentation style we adopt. The DADT concept is illustrated through code examples which allow the reader to appreciate directly how the programmer can exploit DADTs in practice. The syntax, although inspired by modern programming object-oriented languages, serves illustration purposes and is not tied to any particular language. We describe an instantiation of the DADT concept for the Java language in Section 6.

Data and space ADTs. At the core of our model is the notion of ADT. We draw a sharp line between the data necessary to the application behavior, and the space where such data resides. Accordingly, we distinguish between data ADTs and space ADTs.

Data ADTs are conventional ADTs encoding the application logic, like the `Sensor` ADT shown in Figure 1. Instead, space ADTs (or *sites*) are ADTs representing and characterizing an abstract notion of the computational environment (e.g., a computer, a virtual machine, a car, a person) hosting a data ADT. Many notions of space are meaningful, depending on the application. For instance, the network topology may be irrelevant for an Internet application, but it is fundamental in the context of mobile ad hoc networks. Similarly, physical location in space (e.g., gathered through a GPS system) is usually irrelevant, but it becomes fundamental for many context-aware applications. Traditionally, the structure of space is somehow hard-coded in the run-time of the distributed application, and programmers retain only limited—if any—control over it. Instead, in our approach we strive to empower the programmer with the ability to use

```
spacetype GPSSite extends Site {
  data:
    Location l;
  operations:
    Location getLocation();
    double getBatteryLevel();
}
```

Fig. 2. A space ADT representing a physical location.

(and even define) the notion of space that is most appropriate for the application. As such, the notion of site built in our approach is minimalistic, and consists of an ADT `Site`, which must be specialized by the programmer with the proper notion of space. For instance, Figure 2 shows a space ADT whose position in space is characterized by a physical location. How the latter is physically acquired and defined (e.g., from GPS) is entirely encapsulated in the ADT implementation. In addition, it returns the current battery level. One could similarly define a `Host` ADT representing a network host, or any other notion of site, and "export" through the definition contextual data. Our implementation, described in Section 6, provides some built-in site definitions.

The only syntactic difference between data and space ADTs is the use of the keywords `datatype` and `spacetype` introducing the declaration, essentially for enabling type checking and improving code readability and understanding.

From ADTs to DADTs. Distributed ADTs specialize the notion of ADT by providing the ability to treat a set of homogeneous ADT instances as a collective unit, accessed through the operations defined on the DADT interface. To make our presentation more concrete, we return to the reference example introduced in Section 2 and consider a scenario in which the application programmer frequently needs to compute the average of sensed values, as well as to reset all the sensors.

Figure 3 is our starting point in illustrating by example the programming model we put forth in this paper. The figure contains the declaration of a DADT called `DSensor` that enables distributed, collective access to instances of the ADT `Sensor` we defined in Figure 1. The interface of the DADT specifies the signature of two operations providing the aforementioned functionality.

The application programmer—which in large development efforts is likely to be different from both the ADT and DADT programmer—can then access the distributed collection of ADTs by creating a DADT instance and using its operations. An instance of a DADT is created using the constructs for conventional ADTs provided by the target language, e.g., the `new` operator and the notion of constructor. Thus, the application programmer can write

```
datatype DSensor distributes Sensor with {
  operations:
    void resetAll();
    double average();
}
```

Fig. 3. A data DADT providing access to multiple sensors.

```
DSensor ds = new DSensor();
double v = ds.average();
```

to create a `DSensor` instance and use it to request the execution of the distributed processing encapsulated in its `average` operation, as specified by the DADT programmer. Note how the operation invocation above is indistinguishable from any invocation on a conventional ADT instance: the application programmer may even be unaware of the distributed nature of the reference `ds`.

Similarly, one could define a space DADT providing collective access to a set of sites. For instance, one could define a `WirelessNetwork` space DADT distributing `Site` and providing distributed operations to test reachability of a given site or to change the properties of the nodes involved, e.g., modify the wireless range.

The `distributes` relation and the member set. As the reader may have noticed, the declaration of a DADT is similar to the one of an ADT. This is not surprising, since DADTs are ADTs themselves. The only difference is the presence of the `distributes` relation, which extends the range of relations usually defined among ADT types, like inheritance. A `distributes` B, where A is a DADT and B an ADT, states that a set of instances of B^4 can be collectively accessed through the operations defined in A's interface. Clearly, if A `distributes` B and A is a space DADT, B must be a space ADT, i.e., it must be a subtype of `Site`.

The set of ADT instances available for distributed computation through a DADT constitutes the *member set* of the DADT. The member set is effectively contained in the data structure encapsulated by the DADT: every DADT definition implicitly defines a member set, whose elements are ADT instances of the type on the right hand side of the `distributes` relation. In principle, the content of the member set is the same across all the DADT instances in the system. Fulfilling this requirement, however, is impractical not only in context-aware applications, which often involve mobile components and other sources of dynamicity, but in general in any truly distributed system, as asynchrony and concurrency complicate enormously the task of maintaining a globally consistent state. Moreover, different applications may require different, weaker notions of consistency of the member set. For this reason, here and in the rest of the paper we assume that the underlying run-time provides only "best effort" guarantees for what concerns the consistency of the member set. As a consequence, different DADT instances may have different values of the DADT member set (e.g., because of transient disconnections). At the same time, however, as we discuss in Section 6, our flexible architecture of the DADT run-time provides mechanisms enabling customization of the middleware with alternative consistency algorithms.

Binding an ADT into a DADT. Although we defined the notion of member set, we still did not explain how an ADT instance can become part of it. In our model, the binding of an ADT instance into a DADT is requested explicitly by the application programmer (i.e., the "client" of both the ADT and the DADT distributing it) by using the dedicated programming construct `bind`. In our example `bind(new Sensor(),"DSensor")` binds the newly created `Sensor` instance to the member set of the DADT named

4 Or any other type compatible with B according to the typing rules of the target language.

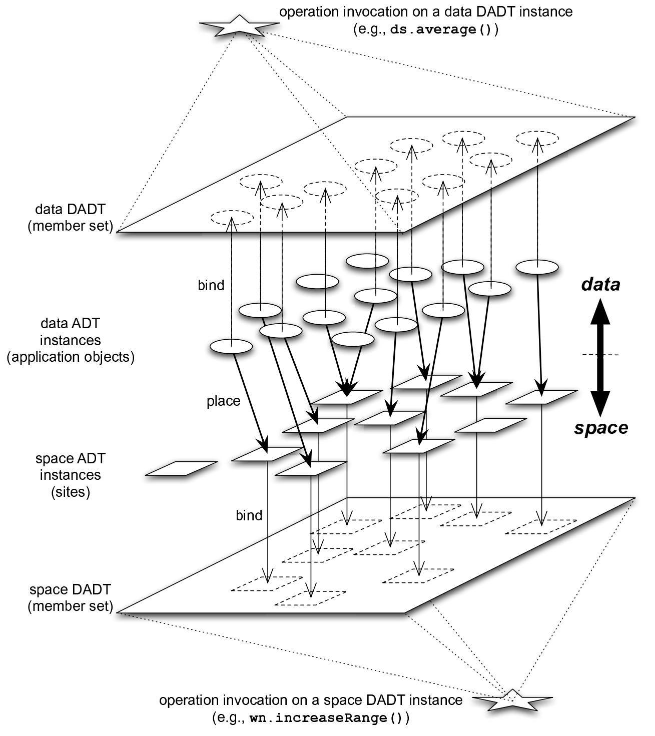

Fig. 4. Data and space in the DADT model.

`DSensor`. The `bind` operation is idempotent. The dual effect of removing a given ADT instance from the member set is obtained by `unbind`, with the same syntax.

The effect of these two operations is global. However, the propagation of the state change, as discussed previously, is not necessarily synchronous w.r.t. the operation invocation. Also, note how an ADT instance is bound to a DADT *type*, not to a specific DADT *instance*. In fact, it is the DADT name that serves to identify uniquely and collectively a group of ADT instances; DADT instances, instead, serve only to provide an "entry point" towards this group.

Putting it all together: The interplay of data and space. Figure 4 provides a graphical representation of the concepts discussed thus far and introduces some new concepts.

The ADT instances in the figure can be bound to a DADT and therefore become part of its member set. The latter is represented visually as a plane, onto which ADT instances (the solid circles and squares) are "projected" as a consequence of a `bind` operation. Note how only those ADTs that are explicitly bound by the programmer (i.e., for which an arrow towards the plane exists) become globally accessible through the member set. Collective access to the member set is enabled through a DADT instance (represented as a star), which serves as the "portal" towards the member set.

This allows access to either data or space, but, at this point in the discussion, the two are not related, meaning application objects are not associated to sites in space and therefore no spatial context can be associated with data ADTs. Conventional programming languages implicitly make this association when an object is created, including it in the local "computational environment", however, not only is this environment not formally defined, but the programmer typically has no control over object placement. Our model does both, defining a new operation, `place`, to define the binding between a data ADT and a space ADT. This operation, represented by the thick lines in Figure 4, can be performed explicitly by the programmer. For example, if `g` is a `GPSSite`, then `place(new Sensor(),g)` binds a newly created sensor to the existing site `g`. While this yields great degrees of flexibility for the programmer, some applications may not require it. In this case, a default site can be provided to the runtime system and every newly created data ADT can be automatically bound to this site. Although this is very similar to the conventional approach, the explicit definition of a site allows symmetric treatment of data and space in the model. It is worth noting that placement of an ADT on a space ADT is not necessary until that ADT is bound to a DADT. Prior to this, the ADT can only be accessed as a regular *object* and no notion of location is needed.

As shown in the figure, multiple ADTs can be placed on the same site. Also, the same ADT can be bound to multiple DADTs; an option that could be represented in the figure by drawing another plane (data or space) for the new DADT. By creating instances of both DADTs, the same ADTs can be accessed through multiple application perspectives at different times. To see why this is useful for sites, consider a host with multiple network interfaces, e.g., Bluetooth and WiFi. One option to support this is to create two space DADTs, one for each network interface. Each host supporting both interfaces should bind to both DADTs. However, for the sake of simplicity the remainder of this paper assumes a site is bound to a single space DADT, however the same is not true for data.

The figure also demonstrates that ADT instances can be accessed by going through instances of either a space DADT or a data DADT, depending on the application needs. Indeed, the power of the abstractions discussed thus far is unleashed when we introduce the ability to restrict the scope of invocation by relying on *both* the data and space perspectives. Consider a laptop-based monitoring application that needs to obtain the average sensed value only for temperature sensors and only in its immediate proximity. Specifying this behavior with conventional programming constructs tends to be cumbersome. Instead, in our DADT language this can be expressed simply and declaratively, as in

```
double v = ds.average() on temperature within proximity;
```

where `temperature` and `proximity` are *views* defined and computed over the member sets associated to the data DADT `DSensor` and a space DADT (e.g., `Network`). Before delving into the details of how this is accomplished, however, we first discuss which constructs are made available to the programmer for specifying the distributed processing embodied in the operations of a DADT. We return to the topic of DADT views in Section 5.

```
1  void DSensor::resetAll() {
2    (all in targetset).reset();
3  }
4  double DSensor::average() {
5    double sum = 0;
6    double[] readings = (all in targetset).read();
7    for(int i=0; i<readings.length; i++)
8      sum += readings[i];
9    return sum/readings.length;
10 }
```

Fig. 5. Aggregating sensor data through DADTs.

4 Distributed Access to ADTs

After the DADT's interface and target ADT are declared, as shown for instance in Figure 3, the DADT behavior realizing distributed, transparent access must be defined by specifying the body of the DADT methods. Since a DADT is associated to the member set, appropriate constructs are necessary to access and manipulate the state of the ADTs in this set both collectively and individually. To accomplish this, we introduce two programming constructs: *operators* and *actions*.

4.1 Operators

Let us focus on the simple task of implementing the `resetAll` method of `DSensor`, whose intended behavior is simply to reset all the sensors in the system. Since the DADT operates on a set of ADT instances, one would like to be able to specify the desired behavior by operating on the set in a declarative way. For instance, in a Z-like formal language, one would express the semantics of the operation with something like:

$$\forall x \mid x \in \mathcal{M} \bullet \texttt{x.reset}()$$

where $\mathcal{M}$ is the member set of `DSensor`. This is expressed in our DADT language as shown in Figure 5, where the expression above is represented by the statement on line 2. In this statement, the invocation target—normally a reference to an ADT instance—is replaced by an expression denoting the set of instances on which the method `reset` is executed. The semantics of execution is such that `reset` gets invoked independently and concurrently on each of the ADT instances belonging to the set in the invocation target. Figure 5 also shows the implementation of the `average` method, where the results of the various invocation are collected and used by the DADT implementation.

Selection vs. condition operators. Next we look more closely at the expression representing the invocation target in line 2 of Figure 5. The variable `targetset`, to which every DADT operation has implicit access, is the set of ADT instances that are available for distributed processing. At this point of our presentation, the target set always coincides with the member set, however this is no longer true when DADT views will be introduced in Section 5. The keyword `in` plays the role of the mathematical membership operator $\in$. Finally, the operator `all` allows one to extract a collective reference to the instances in the target set.

Other operators also make sense. The dual operator `any`, for example, is such that the effect of

```
(any in targetset).reset();
```

is to reset one among the sensors in the target set, chosen non-deterministically. Both `all` and `any` are *selection operators* or, shortly, *selectors*, in that they allow selection of a subset of the instances contained in the target set. In the following, we describe other selectors that add expressive power to our DADT language.

Selection operators essentially enable the programmer to specify declaratively a reference to a distributed invocation target constituted by multiple actual ADT instances. Interestingly, this is achieved transparently, i.e., the programmer does not require any knowledge of the actual *identity* of the instances. In addition, we also provide *condition operators*, which can be used to make the code of a DADT method dependent on a global condition on the target set. The operator `in?` tests whether one set is contained in another, while # returns the number of elements currently in it. With reference to Figure 5, it would be possible to rewrite `resetAll` so that it resets all the sensors only if a given "master" sensor (whose identifier below we assume known) is not available:

```
if (!({master} in? targetset)) (all in targetset).reset();
```

Similarly, we could rewrite `average` so that an average is effectively computed only if, say, more than 3 sensors are around:

```
if ((# targetset)>3) readings = (all in targetset).read();
```

Clearly, other operators could be defined, beyond those discussed here. Examples are a variant of `any` that non-deterministically selects a given number of instances (e.g., as in `any(4)`), or selectors relying on contextual information (e.g., a `nearest` operator that returns the geographically closest instance). Our current prototype provides a built-in implementation for the operators we described thus far, as well as the required mechanisms to enable the programmer define her own, as we illustrate in Section 6.

Iteration operators. The ability to send multiple, concurrent remote invocations empowers the programmer with a high degree of expressiveness. However, in some cases it may lead to inefficient use of communication resources. Examples are situations where only a limited number of the nodes must be contacted, but their number is not known in advance. In these situations, the `all` selector is clearly overkill. The problem is tackled by using *iterators* and the associated operators. Figure 6 shows an example, which would return the same value[5] as the one shown in Figure 5, although likely with a greater latency. The core of the computation is in line 6, where the `next` selector enables iterations over the members of the target set. As with all selectors, `next` operates by picking one of the instances in the target set and returning a reference on which an operation can be invoked. The (mandatory) parameter of `next` is an instance of the `Iterator` ADT (line 4), which embeds the logic used to perform the iteration as well as its current state, and is an argument for all iteration operators. These are, besides `next`, `prev`, `first`, `last`, `cur`, and the conditional operator `more?` used in the loop condition of line 5.

[5] Provided the network remains stable throughout the computation.

```
double DSensor::average() {
  double sum=0;
  int nodes=0;
  Iterator i = new Iterator() on targetset;
  while (more?(i) in targetset) {
    sum += (next(i) in targetset).read(); nodes++;
  }
  return sum/nodes;
}
```

Fig. 6. The `average` method rewritten using iterators.

```
1 void DSensor::resetMax() {
2   <double data, id source>[] readings;
3   readings = (all in targetset).read();
4   int m = 0;
5   for(int i=1; i<readings.length; i++)
6     if (readings[m].data < readings[i].data) m = i;
7   (readings[m].source in targetset).reset();
8 }
```

Fig. 7. Explicitly accessing ADT instances.

As with other operators, the programmer may provide her own implementations of iterators as described in Section 6, for instance to select the required iteration item according to application or physical information (e.g., distance).

Enabling access to a specific ADT instance. The notion of DADT effectively abstracts from the details of distribution, and enables the programmer to treat sets of ADT instances as if they were one. Nevertheless, it is sometimes desirable or necessary, for application needs or performance reasons, to access a given ADT instance. This requires a means of identifying—and therefore distinguishing—it from the rest of the set, and a means to target the instance and manipulate it. Returning to our example, we assume that a new functionality must be added, namely, the ability to switch off the sensor currently reading the maximum value. The task is clearly composed of two parts: finding such sensor, and resetting it.

As for the first problem, we can define a new DADT method `max` that returns such identifier. However, none of the operators described thus far are suited for specifying its behavior satisfactorily. We could use iterators, but that would affect latency linearly in the scale of the system, which is usually not desirable. On the other hand, the `all` operator would help in identifying the maximum value, but not the sensor who read it, which is necessary to solve the second problem. Figure 7 shows our solution to the problem. Line 3 uses `all` to retrieve all the readings, but this time the `readings` variable is an array of pairs `(double,id)` instead of an array of `double`. Both alternatives are available to the programmer, which can therefore request the retrieval of the bare values, or of the corresponding source as well. At translation time, the static declaration is sufficient to perform the necessary translation into the proper data structures of the target language.

```
double DSensor::average() {
  double[] readings; int tries = 3; bool found; int i;
  while (tries > 0) {
    readings = (all in targetset).read();
    found = false; i = 0;
    while (!found && i < readings.length)
      found = (readings[i++] == ERROR);
    if (found) --tries;
    else break;
  }
  if (found) {
    (all in targetset).reset();
    readings = (all in targetset).read();
    found = false; i = 0;
    while (!found && i < readings.length)
      found = (readings[i++] == ERROR);
  }
  if (!found) {
    double sum = 0;
    for (int i=0; i<readings.length; i++)
      sum += readings[i];
    return sum/readings.length;
  } else /** report fault to the application **/;
}
```

Fig. 8. Access to remote ADTs: a naive solution.

Once the identifier is obtained, we are left with the problem of accessing explicitly the corresponding sensor to reset it. This is naturally encompassed in our language by using the identifier of the sensor as a selector (line 7). In general, a set of identifiers can be used, thus supporting a specialization of the `all` selector where the invocation target is a subset of programmer-specified instances.

4.2 Actions

The use of the aforementioned operators enables concurrent access to remote ADT instances. Thus far, we have assumed that such access occurs only through one of the ADT's operations. However, in many cases, this is not sufficient.

For instance, let us assume that the ADT's `read` operation is capable of signaling a malfunction by returning an `ERROR` value (e.g., a `double` value outside the range of meaningful physical values sensed). In this case, it may be reasonable to circumvent transient faults (e.g., due to interference of the sensor with physical phenomenons) with simple countermeasures. A reasonable behavior could then be to retry the read operation a number of times, after which the sensor is reset and the read repeated again. If also this last attempt fails, the fault is reported to the application.

A naive implementation of this DADT behavior is shown in Figure 8. This solution is clearly highly inefficient, since every time a fault is reported, the `read` operation

```
 1 double DSensor::average() {
 2   action double reliableRead() {
 3     double reading; int tries = 3;
 4     while (tries > 0) {
 5       reading = local.read();
 6       if (reading == ERROR) --tries;
 7       else break;
 8     }
 9     if (reading == ERROR) {
10       local.reset();
11       reading = local.read();
12     }
13   }
14   double[] readings = (all in targetset).reliableRead();
15   bool found = false; int i = 0;
16   while (!found && i < readings.length)
17     found = (readings[i++] == ERROR);
18   if (!found) {
19     double sum = 0;
20     for (int i=0; i<readings.length; i++)
21       sum += readings[i];
22     return sum/readings.length;
23   } else /** report fault to the application **/;
24 }
```

Fig. 9. Access to remote ADTs using an action.

is retried on *all* sensors. Explicitly accessing a remote instance, as described in Section 4.1, would only partially solve the problem by enabling the programmer to limit communication only towards those sensors that reported failure. Nevertheless, even in this case a single interaction may result in several exchanges across the network, since for each sensor the `read` and `reset` operations need be invoked from the (remote) DADT instance. Moreover, this solution still requires a lot of bookkeeping, to keep track of which sensors are still faulty and need another try and which instead started working again.

Both solutions are inherently unsatisfactory because they ignore a fundamental point: the sequence of failing `read` and corrective `reset` operations do not require intervention of the DADT instance and instead can be controlled local to the ADT instance. In other words, a distinction is necessary between the application logic that determines how to recover from a fault (which is entirely local to a sensor) and its distribution across the system.

This separation can be achieved elegantly and efficiently with the notion of *action*. An action is essentially an operation that is defined in the DADT but whose execution occurs on the ADT instance on which it is invoked. Loosely speaking, actions enable the programmer to write DADT code that operates on ADT instances as if they were exporting a richer interface, whose content is under control of the DADT programmer.

Figure 9 illustrates the concept. The action declaration is contained in lines 2–13, and is identical to the declaration of a standard programming language routine, prepended by the keyword `action`. The only difference is the use of the keyword `local`, which is bound at runtime to the ADT instance on which the action is currently being evaluated. Note how `local` is different from the traditional `this` keyword, pointing in this case to the DADT instance on which the operation containing the action (`average`) is being invoked. This should not be surprising, in that although the action *definition* belongs to the DADT and its execution is triggered through one of the DADT operations, the action *execution* is entirely local to the sensor, as if it were just another operation[6] of the ADT. These semantics can be "visualized" by considering actions as mobile code [5] being shipped dynamically and remotely evaluated on the ADT instances. However, mobile code is only one of the options available for their distributed execution.

The action code in Figure 9 performs the local read and, if an error is reported, retries the read and possibly resets the sensor. The `average` DADT operation exploits this action definition by simply invoking the action over the sensors in the target set using the notation we described in Section 4.1 (line 14). The remainder of the operation scans the obtained readings for error codes (returned by sensors with a persistent fault) and either computes and returns the average as in Figure 5 or reports the error to the application.

Actions declared in an operation block are not visible outside, according to lexical scoping rules. However, an action can also be declared in the DADT interface. The action becomes visible and can be reused by any of the DADT operations and by client objects calling these operations. The action code becomes encapsulated in the DADT, thus providing a beneficial form of information hiding.

5 Restricting the Scope of Operations

Distributed sharing of ADTs as we have defined it thus far is a powerful concept. However, in many situations the invocation of an operation over *all* the instances in the member set may not be desirable. Performance reasons may render it impractical. Application needs may suggest better policies that restrain the effect of an operation only to a subset of the instances.

Our reference example helps in clarifying these concepts. In our scenario, different kinds of sensors (e.g., temperature, light, humidity) may be present. In Figure 1 this fact is taken into account by the attribute `sensorType`. If the only notion of sharing available is one where all instances of a given type are effectively considered for the distributed computation of `average`, then values belonging to sensors of different kinds are averaged together, yielding a meaningless result. In this case, we would like to be able to operate only on a subset of the sensors available—those of a given kind. One could argue that the sensor kind could (or should) be encoded as a different ADT, e.g., inheriting from a supertype representing an abstract notion of a sensor. Unfortunately, the same requirement may hold for other characteristics of a sensor that represent a portion of its state, as opposed to a static characteristic like the sensor type, e.g., to reset all the sensors that are currently inactive or to compute the average only from sensors

[6] For what concerns distributed execution, ADT operations can be regarded as and are effectively treated as a special case of action.

```
datatype DSensor distributes Sensor with {
  properties:
    bool isActive();
    bool isSensorType(int sensorType);
    bool isPrecise(double resolution);
  actions:
    double[] reliableRead();
  operations:
    double average();
    void readAll();
}
bool DSensor::isActive() {
  return local.isActive;
}
bool DSensor::isSensorType(int sensorType) {
  return local.sensorType == sensorType;
}
bool DSensor::isPrecise(double resolution) {
  return local.resolution >= resolution;
}
void DSensor::resetAll()  {/* ...as in Figure 5... */}
double DSensor::average() {/* ...as in Figure 5... */}
```

Fig. 10. Restricting the scope of operations over data.

that guarantee a given resolution. In essence, in these cases one would like to be able to specify something to the effect of

$$\forall x \mid \neg(x \text{ is active}) \wedge\ x \in \mathcal{M} \bullet x.\texttt{reset}()$$

or even

$$\mathcal{A} \overset{\text{def}}{=} \{x \mid x \text{ is active } \wedge\ x \in \mathcal{M}\} \qquad \forall x \mid x \notin \mathcal{A} \bullet x.\texttt{reset}()$$

We provide this level of flexibility and expressiveness in DADTs by introducing the notions of property and view. Conceptually, a *property* is a characteristic of a DADT defined in terms of an ADT's data and operations, and evaluated local to an ADT instance. In programming terms, properties are specified as part of the DADT interface as operations returning a boolean. Figure 10 shows the DADT we defined earlier in Figure 3 augmented with properties. For instance, `isActive` returns true if the local value of the attribute `isActive` (see Figure 1) on the `Sensor` instance where the property is being evaluated is true as well. Similarly, `isSensorType` and `isPrecise` return true if the target sensor is of the kind specified or provides a sufficient resolution, respectively. The definition of these properties in Figure 10 relies on the `local` keyword to access the ADT instance they are currently being evaluated upon, similarly to actions in Section 4.2. Indeed, like actions, properties are defined on the DADT but evaluated on (remote) ADTs.

This simple concept enables the definition of partitions of the member set into programmer-defined subsets, which we call DADT *views*. Properties define the membership function for the elements in such subset. Views are defined by the programmer using dedicated constructs. For instance, the $\mathcal{A}$ subset in the formula above effectively represents a view, which can be declared as:

```
dataview active on DSensor as isActive();
```

In this case, the view `active` is defined as the subset of the member set of `DSensor` and contains only those (`Sensor`) instances for which the evaluation of the property `isActive` yields true. The DADT name is used to refer implicitly to its member set.

Properties can have parameters. For instance, the view containing all temperature sensors can be defined as:

```
dataview temperature on DSensor as isSensorType(TEMP);
```

After a view is defined, it can be used for restricting the scope of a DADT operation. For instance, the following code snippet

```
ds.resetAll() on !active;
```

resets all the sensors that are currently inactive. The semantics of execution is such that the `targetset` in line 2 of Figure 5, which in Section 4.1 was bound to the member set, is here bound at invocation time to the identifiers of the ADT instances belonging to the view. Of course, the resulting view may be empty, i.e., no instance satisfies the view definition. In this case no operation is performed.

Note how in the statement above the boolean negation operator is used to obtain the subset of instances that are *not* in the view `active`. This is an example of the more general ability to compose views by connecting their properties using boolean operators. The mechanics are trivial, since the properties defining the views are essentially logic predicates over the state of an ADT instance. Yet, this feature is powerful as it allows one to express views using set union and intersection. For instance the following definition

```
dataview preciseOn on DSensor
      as isPrecise(0.1) && isActive();
```

captures the subset of sensors that are active and provide a resolution greater than 10%.

It is worth noting how all the views we defined thus far are symmetric, i.e, regardless of the specific DADT instance requesting the operation whose scope is restricted by the view, the latter always contains the same elements. In some cases, however, it is useful to define views that depend on the *state* of the DADT instance target of the distributed operation. As we briefly mentioned in Section 3, DADTs may define attributes, e.g., to store intermediate, aggregated data. Imagine a variation of the declaration of `DSensor` in Figure 10 where an attribute `lastAverage` of type `double` is declared. This attribute may be used to cache the last average value read through `average`, and can be exploited to define a property

```
bool DSensor::isBelowAverage() {
  return local.read() < this.lastAverage;
}
```

which returns true for all sensors whose currently read value is below this cached average value. Any view involving this property is asymmetric, in that the subset of the member set it denotes depends on the value of `lastAverage` associated to the

```
spacetype WirelessNetwork distributes WirelessSite with {
  properties:
    bool isReachable(int hops);
  operations:
    void modifyRange(double percent);
}
```

Fig. 11. A space DADT providing distributed access to sites.

`DSensor` instance involved in the distributed computation, which can be different since different instances may have invoked `average` at different times. The semantics of execution is such that the value of the attribute accessed through `this` is "frozen" at invocation time, as this enables a straightforward distributed implementation without losing significant expressive power. The use of the `this` keyword, however, must be limited to enabling access to attributes of the DADT, since DADT operations cannot be performed from within the property, as its evaluation takes place on the ADT and not on the DADT instance. Both the ADT data and operations can instead be accessed freely through `local`.

Multiple views may coexist in the same code fragment, and be used at different times on the same DADT instance. For example, based on some of the previous declarations, one could write:

```
ds.resetAll() on temperature;
double v = ds.average() on (preciseOn && temperature);
```

The statements above operate on the same DADT instance, but on different target sets and with different operations. Moreover, the definitions of `resetAll` and `average` are unchanged from those we provided in Section 4.

If no view is specified at invocation time, the operation is performed on the whole member set, as we discussed in Section 4.1. Indeed, the member set defines the most general view containing ADT instances bound to the DADT. Moreover, all the constructs we discussed in Section 4 can be used with views, as these are ultimately sets of ADT instances. This holds not only for the DADT programmer *inside* the definition of an operation, but also for the application programmer and therefore *outside* the DADT definition. For instance, the following program fragment reliably retrieves the values sensed by all the active temperature sensors using the action whose code[7] appeared in Figure 9, provided that there are at least three of these sensors provide sufficient resolution:

```
if ((# (preciseOn && temperature)) >= 3)
  (all in (active && temperature)).reliableAverage();
```

All the considerations we made thus far clearly hold not only for *data views* like those we used in our examples, but also for *space views*, i.e., views that are defined over space DADTs. At different times, it may be necessary to access different sets of sensors based on the configuration of the space where they reside. In a resource-constrained environment, for instance, most of the operations may involve sensors that are close, e.g., two hops away from the object requesting the probe. Figure 11 shows the definition[8]

[7] Note how, contrary to Figure 9, in Figure 10 the action had been declared in the interface of the `DSensor`.

[8] We omit the definition of `WirelessSite` for the sake of brevity.

of the `WirelessNetwork` space DADT we briefly mentioned at the end of Section 3, addressing this requirement. `WirelessNetwork` defines a property `isReachable` that yields true if the target host is within a specified number of hops. Similarly to data views, the programmer can now define a space view, e.g.,

```
spaceview proximity on WirelessNetwork as isReachable(2);
```

and use it to restrict an operation's scope over the space DADT, e.g., to reduce by 10% the communication range of "nearby" hosts as in

```
wn.modifyRange(-0.1) within proximity;
```

Note how `proximity` is asymmetric, as it depends on the implicit location of the `DSensor` instance on which it is invoked.

Analogously to DADT types, data views are syntactically distinguished from their space counterparts using `dataview` and `spaceview`. This enables type checking to prevent incorrect use of properties in a view definition (e.g., using a property on a space DADT to define a data view), or incorrect use of a view in an operation invocation (e.g., using a data view in place of a space view).

Data views and space views can be used together. For instance

```
double v = ds.average() on temperature within proximity;
```

returns the average value of all temperature sensors in the space region defined by the view `proximity`. Similarly,

```
wn.modifyRange(-0.1) on temperature within proximity;
```

reduces the wireless communication range of all the nearby nodes hosting a temperature sensor. The content of `targetset` inside the body of the invoked (data or space) DADT operation is computed as the intersection of the subsets defined by the two views. Figure 12 illustrates the concept w.r.t. the first of the two statements above. Differently from Figure 4, data and space ADT instances are not shown—only their "projection" on the member sets is. The arrow between a data ADT (circle) and a site (square) means that the former is placed on the latter. The black components on the left make explicit the fact that DADT instances are themselves ADT instances and reside on a site, which can be relevant for the definition of asymmetric views, e.g., `proximity`.

As we mentioned in Section 3, our DADT model provides a unified representation of data and space, where they are simply two different perspectives for accessing and manipulating the application ADTs. The notion of DADT provides the mechanism for defining the application behavior manipulating the distributed state, as well as the observable state that can be used to define views. DADT instances are instead the dynamic entities through which distributed access is effected, and whose scope is restricted dynamically by means of views. The programmer is free to decide what is the best "vantage point" for accessing the distributed system. She can use a data DADT instance to operate on the distributed data and yet restrict the scope using predicates that are based on characteristics of data, space, or both, depending on the needs of the application. Similarly, she can use any kind of view to access the distributed representation of space. In our model, data and space become easily interchanged during the programming practice, with our model coherently maintaining their semantic interaction.

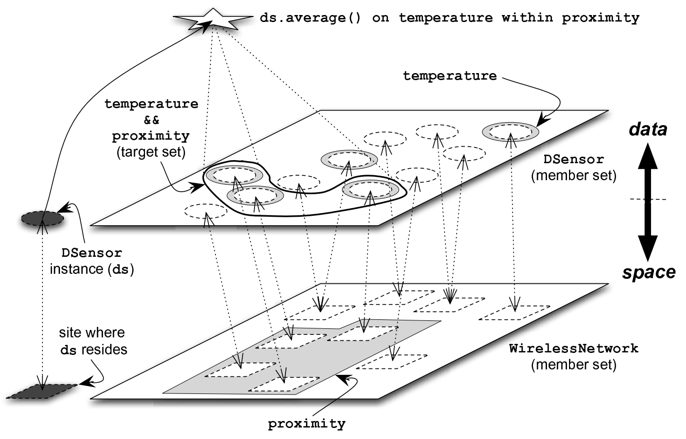
ds.average() on temperature within proximity
temperature
temperature && proximity (target set)
data
DSensor (member set)
DSensor instance (ds)
space
site where ds resides
WirelessNetwork (member set)
proximity

Fig. 12. DADT views.

6 Prototype Implementation

To verify the feasibility of the DADT model we developed a proof-of-concept prototype, providing the DADT constructs described thus far in the context of the Java language. The prototype, currently nicknamed $\mathcal{J}$DADT, is divided into two parts: a translator and a run-time. The *translator* takes care of translating a Java program augmented with statements from our DADT language into a conventional Java program, through a pre-compilation step. The code generated by the translator implements the DADT constructs by using the classes defined in the *run-time* library. Once the translation is generated, the resulting code can be directly executed on any Java virtual machine where the run-time packages are deployed.

The translator for the source-to-source transformation is implemented using the ANTLR [1] parser generator. The source grammar is a modification of the Java 1.5 grammar by Studman [13] with extensions for DADT constructs. When launched, the ANTLR generator builds the Abstract Syntax Tree (AST) with custom nodes for DADT constructs which are next modified by *tree walkers* to reconstruct a plain Java AST. This AST is emitted with a standard ANTLR Java emitter into code that contains invocations to the run-time as detailed next.

The run-time architecture is composed of several components, the main classes of which are outlined in Figure 13. The top layer is constituted by application classes, like those we used in our example. While the definition of data ADTs and DADTs is entirely up to the programmer, our implementation provides built-in notions of `Host` and `Network`.

The layer below constitutes the API of our DADT run-time. This API is not directly accessible to the programmer who codes using the constructs we defined earlier. Instead, it is used by the translator, to map the DADT constructs into the appropriate run-time objects and invocations. The class `DADTMgr` provides the methods handling the binding of ADTs to DADTs (used to execute `bind` and `unbind`), to specify the `Site`

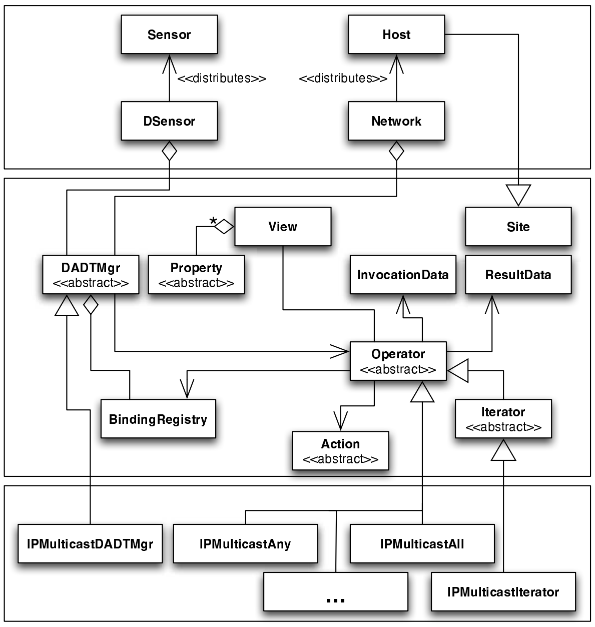

Fig. 13. The architecture of the $\mathcal{J}$DADT run-time.

instance abstracting the *node*[9] where the run-time executes, as well as other auxiliary methods managing configuration aspects of the run-time. There is only one `DADTMgr` instance per node.

The abstract classes `Property` and `Action` represent the corresponding concepts, and are similar in that both are essentially DADT methods whose execution takes place on a (remote) ADT. To understand the translation strategy, let us focus on the `isPrecise` property shown in Figure 10. The translator can straightforwardly generate the following corresponding class:

```
class isPrecise_Property extends Property {
  double resolution;
  isPrecise_Property(double resolution) {
    this.resolution = resolution;
  }
  boolean evaluate(Object o) {
    Sensor local = (Sensor) o;
    return local.resolution >= resolution;
  }
}
```

where the property parameter is now a class attribute, and the property body is contained in the `evaluate` method. The latter accepts as a parameter an instance of the ADT

[9] Hereafter, we use the term *node* to represent the physical computational environment where the run-time executes. In our implementation, the node is a JVM, and its identifier a pair `host:port`. The node should not be confused with a *site*, which is a node's abstract representation as a space ADT.

distributed by the DADT (`Sensor` in our case), which can be safely substituted for the keyword `local` we used in Figure 10. A similar strategy is used for translating actions like the one in Figure 9 into an `Action` class. `Property` and `Action` objects are created upon action invocation or view definition, serialized, and once deserialized on the node of the target ADT instance (e.g., the `s` object) their `evaluate` method is executed by using such object as a parameter. In doing this, we currently assume that all the corresponding code is pre-deployed, together with the DADT class, on the interested nodes. A more dynamic and open design can be easily obtained using a code on demand mobile code paradigm [5], to dynamically relocate the involved classes only if and when really needed.

A view (data, space, or a mixture thereof) is represented by a `View` object, which contains the set of `Property` objects associated with the view definition. To easily manage the composition of properties through boolean operators, the property objects are arranged in an abstract tree representing the logical predicate defining the view, where the leaves are the property objects and the nodes are the boolean operators used to compose them. To determine whether a given ADT instance belongs to the view, the method `isMember(Object o)` navigates the tree from the leaves to the top, invoking the `evaluate` method of each property and composing it with others through the boolean operator in each node. The process terminates at the root, (i.e., the view object) with the boolean result of the evaluation. Views are easily composed by similarly merging their property trees by means of boolean operators. Finally, `View` provides a method `apply` that allows execution of an action on the ADT instances selected by an operator.

`Operator` is the superclass of any of the DADT operators described in Section 4.1. `Iterator` specializes `Operator` by providing directly the iteration operators (e.g., `next`) as methods. The programmer can provide new operators and iterators by simply subclassing the aforementioned classes. The correct implementation can be built-in by the translator or retrieved at run-time from `DADTMgr`, designed using the Factory pattern. The behavior of an operator is specified by overriding two methods. The method `performRemote(View v)` is invoked on the node where the program requests the evaluation of an operator, and embeds the logic for distributing the information necessary to the collective evaluation of the operator on the view, and the retrieval of the invocation results. Instead, `performLocal(InvocationData d)` is performed on all the other nodes involved in the computation, and contains the logic for evaluating the operator local to a single node and sending the results back to the initiator. `Operator` essentially embeds all the distribution logic, as discussed later.

`InvocationData` is used for communication and contains the components necessary to perform an invocation, i.e., the view specification, the operator to be applied, the action to be executed, and the initiator's identifier. `ResultData` contains an array of pairs of `Serializable` and ADT identifiers[10], and is the type of return values for `performLocal` and `performRemote`. Finally, the `BindingRegistry` object associated to the `DADTMgr` singleton contains the information about which node's ADT is bound to which DADT, and provides methods to determine the local ADT in-

[10] These are global. In our implementation they are the Java object identifier with the node identifier where the object is created.

stances that satisfy a given view specification. Finally, the bottom layer contains the classes specializing our framework.

An example helps understand how the various components cooperate. Consider the program statements

```
dataview preciseOn on DSensor as isPrecise(0.1) && isActive();
spaceview proximity on Network as isReachable(2);
ds.average() on preciseOn within proximity;
```

with the fault-tolerant definition of `average` in Figure 9. The translator would first generate the `Property` subclasses for the properties defining the two views, create the corresponding objects, and insert them in the abstract tree, either directly or by invoking the methods (`and`, `or`) for logically composing properties:

```
View preciseOn = new View(new isPrecise_Property(0.1)
                    .and(new isActive_Property()));
View proximity = new View(new isReachable_Property(2));
```

Moreover, the translator modifies the signature of `average` into

```
double average(View targetset)
```

so that the representation of the view specification (conjunction of `preciseOn` and `proximity`) which effectively becomes the operation target set can be passed upon method invocation as in

```
ds.average(new View(preciseOn).and(proximity));
```

Note how a new `View` instance is generated on the fly to represent the `on...within...` portion of the invocation by merging the data view and the space view. In our implementation, both are represented using `View` objects, which are then composed like properties. Different constructors of `View` are provided to create a view out of its properties or based on already existing views. Moreover, the translator generates a subclass of `Action` representing the action `reliableRead` in the figure, translated as we described earlier. Line 14 of Figure 9, containing the action invocation in conjunction with the `all` operator, is therefore translated as

```
double[] readings =
  (double[]) view.apply("all",new reliableRead_Action());
```

The body of the `apply` method retrieves from the node's `DADTMgr` instance the proper implementation of the operator based on the name being passed as a parameter, and starts its distributed execution on the bound ADTs, as shown in the following excerpt

```
Object apply(String name, Action a) { ...
  Operator op = DADTMgr.getOperator(name);
  ResultData d = op.performRemote(
    new InvocationData(this,name,a,initiator));
... }
```

where `this` is the `View` instance requesting the invocation and `initiator` is the identifier of the corresponding node.

As we mentioned earlier, the actual implementation of the operator manages directly the communication between the initiator node and those hosting ADTs. In our prototype, for instance, the implementation of the `all` operator leverages off UDP unicast and multicast sockets. Multicast is exploited for distributing information to the ADT instances bound to a given DADT. The implementation of `IPMulticastAll.performRemote`, therefore, simply sends the `InvocationData` object to the multicast group associated to the DADT name (e.g., `DSensor`). On the remote ADT nodes, the communication run-time (which is initialized by our own specialization of `IPMulticastDADTMgr`) receives this object and, based on the information it contains, creates the appropriate instance of the `all` operator and delegates to `performLocal` the processing of the `InvocationData` object. This latter method takes care of the local processing, i.e., it queries the local `BindingRegistry` to obtain all the ADTs that are bound to the DADT and whose state satisfies the view specification in `InvocationData`. Moreover, it performs the action invocation `a.evaluate(s)` on each ADT instance returned. The results are packed in a `ResultData` object and sent back to the initiator. In `IPMulticastAll` this is done using UDP unicast.

The design we just described is conceived as *framework*, in the object-oriented sense and can be customized by changing a limited number of classes, most importantly `Operator` subclasses and `DADTMgr`. For instance, in our implementation when a `bind` operation is issued `IPMulticastDADTMgr` takes care of joining the multicast group corresponding to the bound DADT. Moreover, note how each operator can potentially define its own way to manage communication. For instance, while the `all` operator dispatches directly the `InvocationData` using multicast, the implementation of iterators effectively uses multicast to build the target set and then uses unicast communication for contacting each ADT.

Although our $\mathcal{J}$DADT run-time is a proof-of-concept prototype, its design and implementation are still non-trivial. It is fully decentralized and, although we tested it thus far only in a fixed environment, its reliance only on the most basic network facilities leaves open the opportunity for a seamless migration to more dynamic ones, using the appropriate routing algorithms (e.g., MAODV [11]). Nevertheless, it can clearly be improved in many respects. Most notably, we are currently studying distributed algorithms for managing more efficiently the distributed dissemination of actions and results, and for maintaining views. In doing this, we are supported by our previous work on data sharing middleware for mobile computing (e.g., LIME [9] and EgoSpaces [8]).

7 Related Work

The closest works are probably in the context of parallel systems. Shared ADTs (SADTs) [6] focuses on providing implementations of several standard data types, whose implementation is designed to scale well in the parallel environment. Concurrent Aggregates (CAs) [4] provides language-level support for defining both the ADT interface and the implementation of its distributed components. Each component is defined in terms of message processing and is explicitly enabled to send messages to fellow components, creating aggregate behavior. In comparison to these systems, not only DADTs target the more general distributed setting, but they also provide a unique and uniform treatment of data and space, as well as the increased flexibility coming from the view concept. In the other systems, not only is the view the same at all times, but the components contributing to it cannot change during execution.

DADTs are also somehow related to software distributed shared memory (SDSM) models, which aim at masking entirely distribution, while in our approach distribution is under the control of the programmer. Recently, SDSM has been applied to embedded systems in the Spatial Programming model [2] by making an analogy between space and memory and exposing space to applications through spatial references, which enable the definition of regions determining the components interested in a given computation. In contrast, the main advantages of DADT stem from the clean separation between data and space, and its ability to provide access to multiple instances through a single DADT reference, instead of access to a single object. Moreover, this is accomplished by defining a data view, further restraining the spatial constraints.

Some aspects of our work may look reminiscent of distributed object platforms (e.g., CORBA or Emerald [7]). However, these platforms rely on remote method invocation as a means to access explicitly identified, single object instances. Instead, in our DADT model the identity of remote objects remains hidden (unless explicitly needed), enabling transparent access. Moreover, collective access occurs through a single DADT interface, while the same aggregation would require extensive programming effort in a distributed object system. Finally, distributed object systems hide the object location in references, while DADTs foster a clean separation between data and space, hiding location when dealing with data, but enabling its (direct or indirect) access when dealing with space.

Another research connection is with process calculi exploring the representation of locality in concurrent and distributed systems. For instance, the Ambient calculus [3] represents space as a hierarchical composition of scopes (ambients) where processes dwell. Processes can dissolve ambients as well as migrate from one ambient to another. However, while a hierarchical structure of space is easy to reason about, is easily mirrored in the language syntax, and may be well-suited to model the logical mobility of agents, it is too rigid to represent the physical mobility of hosts, as physical space is rarely best modeled as a hierarchy. Similar considerations hold for other works that also adopt a hierarchical representation of space, e.g., the extension to join calculus supporting context-awareness described in [14]. A more recent and flexible proposal is τKLAIM [10], which assumes processes communicating and migrating across a (flat) network, whose dynamic changes can be described and encompassed in the calculus. DADTs currently do not have a formal model, although its definition is among our current research goals. Nevertheless, we maintain that the aforementioned languages are less expressive than DADTs in terms of their manipulation of data and space. Namely, they capture distribution at a lower level of abstraction and do not provide the ability to scope the execution of operations and actions based on arbitrary predicates over data and/or space. Moreover, the distance of all these theoretical approaches from mainstream programmers is large. We contend that by integrating our model in the mainstream abstraction of ADT and by embodying it in the popular Java language and implementation we reduce the semantic gap between our abstractions and their real use by the programmer.

Finally, the idea of using sets as a programming abstraction was pioneered in the early 70's (e.g., [12]). However, the goal was to use sets for *all* programming tasks. In our context, we use the set abstraction only to deal with some aspects of distribution and therefore we do not need the full power provided by the aforementioned languages. Instead, our goal is to blend our set-based programming constructs into those of modern languages.

8 Conclusion

Developing distributed applications is a complex task, especially when the physical space is involved in the application requirements and logic, as in the case of context-aware computing.

In this paper, we proposed DADTs as a novel distributed programming model enabling collective access to data and space entities by means of operations whose distributed behavior is encapsulated in the DADT using dedicated constructs, and whose invocation scope can be dynamically defined based on application and contextual information. We conjecture that the unified treatment of data and space concerns inside the model, together with our choice to embed these features in a well-known and widely used programming technique, is likely to improve programming practices in modern distributed and context-aware computing.

Future work will address both the model and the implementation. A formalization of the syntactic and semantic aspects of DADTs is among our immediate goals. Moreover, we will continue improving and refining our prototype, investigating efficient algorithms for view maintenance and action dissemination.

References

1. ANTLR Web page. `www.antlr.org`.
2. C. Borcea et al. Spatial programming using smart messages: Design and implementation. In *Proc. of the 24th Int. Conf. on Distributed Computing Systems (ICDCS)*, March 2004.
3. L. Cardelli and A. D. Gordon. Mobile ambients. In *Foundations of Software Science and Computation Structures: First International Conference, FOSSACS '98*. Springer-Verlag, Berlin Germany, 1998.
4. A. Chien and W. Dally. Concurrent Aggregates (CA). In *Proc. of the 2nd ACM SIGPLAN Symp. on Principles & practice of parallel programming*, pages 187–196. ACM Press, 1990.
5. A. Fuggetta, G.P. Picco, and G. Vigna. Understanding Code Mobility. *IEEE Trans. on Software Engineering*, 24(5), 1998.
6. D. Goodeve et al. Toward a model for shared data abstraction with performance. *J. of Parallel and Distributed Computing*, 49(1):156–167, 1998.
7. E. Jul, H. Levy, N. Hutchinson, and A. Black. Fine-grained Mobility in the Emerald System. *ACM Trans. on Computer Systems*, 6(2):109–133, Feb. 1988.
8. C. Julien and G.-C. Roman. Active Coordination in Ad Hoc Networks. In *Proc. of COORDINATION 2004*.
9. G.P. Picco, A. Murphy, and G.-C. Roman. LIME: Linda Meets Mobility. In *Proc. of the 21st Int. Conf. on Software Engineering*, pages 368–377, May 1999.

10. R. D. Nicola, D. Gorla, and R. Pugliese. Pattern Matching over a Dynamic Network of Tuple Spaces. In *Proc. of the 7th Int. Conf. on Coordination Models and Languages (COORDINATION)*, LNCS 3454, Namur (Belgium), April 2005.
11. E. Royer and C. Perkins. Multicast Operation of the Ad-hoc On-Demand Distance Vector Routing Protocol. In *Proc. of MobiCom*, 1999.
12. J. T. Schwartz et al. *Programming with sets; an introduction to SETL.* Springer, 1986.
13. M. Studman et al. Java 1.5 Grammar. `www.antlr.org/grammar/1109874324096/java1.5.zip`.
14. P. Zimmer. A Calculus for Context-Awareness. Technical Report RS-05-27, BRICS: Basic Research in Computer Science, August 2005.