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,
{picco,migliava}@elet.polimi.it
[2] Faculty of Informatics, University of Lugano, Switzerland, amy.murphy@unisi.ch
[3] Dept. of Computer Science and Engineering, Washington Univ. in St. Louis, USA,
roman@wustl.edu

Abstract. In this paper we introduce the concept of *Distributed Abstract Data Type* (DADT), a new programming model simplifying the development of distributed, context-aware applications. A DADT instance logically encapsulates a collection of ADT instances distributed throughout the system. DADT operations specify the semantics of access to this distributed state by using dedicated programming constructs. The scope of these operations can be restricted using DADT *views*, i.e., projections over the target ADT instances, defined declaratively and dynamically based on ADT properties. Interestingly, DADT constructs can be used to deal not only with application data, but also with 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 demonstrated by our prototype implementation.

1 Introduction

Modern distributed computing places new demands on application programmers, not only due to the increasing scale, decentralization, and dynamicity, but also because of new requirements about visibility and control over the physical space where the application executes. Examples include scenarios that fall under the umbrella of pervasive and ubiquitous computing, and ambient intelligence. Here, programmers must deal with simultaneous access to a plethora of devices, sensors, or application objects dispersed in the environment to collect the application and contextual data determining the program behavior. However, despite the popularity of the field, available models and systems often treat the physical space—or context—where the application executes as external, introducing specialized constructs and increasing the programming effort. Similarly, abstractions for dealing with distribution are often quite primitive, forcing the programmer to deal explicitly with individual remote accesses.

The model described here provides constructs simplifying the access to distributed state while explicitly taking into account spatial concerns. We accomplish this by extending the well-established notion of abstract data type (ADT) into a *distributed abstract data type* (DADT). A DADT instance logically encapsulates a collection of ADT

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

Fig. 1. ADT interface for a simple sensor.

instances distributed throughout the system. This distributed state can be manipulated through the operations in the DADT interface, whose distributed behavior is defined by the DADT programmer with appropriate constructs. Moreover, the application programmer can dynamically restrict the scope of DADT operations by defining, in a declarative way, projections over the distributed state, called *views*. The definition of DADTs and views, differently from conventional models and languages, is used also to represent space. The programmer can model the notion of space best suited for the application with a DADT, and use its properties to define views over space (e.g., proximity). Our model fully integrates the spatial and data concerns involved in context definition under a single, unified framework centered on the notion of DADT. Data and space become two distinct, and yet intimately related, perspectives enabling 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 presents the basic concepts of our DADT model. Section 4 discusses the constructs enabling distributed access, while Section 5 illustrates the aforementioned notion of view. 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 Motivating Example

This section introduces an example used throughout the paper to concretely illustrate the main concepts of the model. Consider an environment where several sensors are deployed to report about physical parameters (e.g., temperature, light). A laptop monitoring station needs to collect average temperature readings in its proximity, as well as to manipulate the configuration of the sensor nodes (e.g., reset them or change their transmission range).

In a conventional approach, interaction with sensor data can be modeled with an ADT, as shown in Figure 1. However, this enables only local access to a single sensor (i.e., `Sensor` instance) at a time. To build the functionality above, requiring computation of a global property, the programmer must deal with individual, remote access to sensors, and determine (and keep track of) which are relevant to the computation (e.g., only nearby temperature ones). Clearly, a new ADT (say, `SensorGroupProxy`) can be created to encapsulate these concerns. The point, however, is that *i)* the burden of defining the behavior of this latter ADT is entirely on the programmer; *ii)* this is accomplished by mixing the application logic with the management of where data is deployed and how it becomes dynamically accessible through physical space; therefore

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

Fig. 2. A data DADT providing access to multiple sensors.

iii) program understandability and maintainability are negatively affected, and so is the possibility of reuse across applications.

The goal of this paper is to simplify the programmer's task through a programming model that *i)* separates the application logic from the management of contextual information; *ii)* simplifies the specification of the distributed computation through composition of elementary, reusable building blocks; *iii)* is easily integrated in mainstream languages. To provide the reader with a concrete glimpse of how this is accomplished, Figure 2 shows the definition of a *distributed abstract data type* (DADT) that embeds the logic for resetting a group of sensors and computing their average reading. The behavior of the operations is specified by the programmer through the constructs we provide, which relieve her from the need to deal explicitly with the details of distribution. Moreover, the execution of the application logic embedded in the operations is decoupled from the specification of the system portion affected. For instance, the computation of the average reading detected by nearby temperature sensors is intuitively achieved by:

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

Here, `ds` is an instance of the `DSensor` DADT in Figure 2, while `temperature` and `proximity` are projections over the set of all possible sensors, based on their properties. For instance, the former is defined so that it selects only `Sensor` instances with the proper value of the attribute `sensorType`. The spatial properties related to `proximity` are handled analogously through space ADTs and DADTs, defined later.

In the following we incrementally present the concepts necessary to support this example. Our discussion starts with the basic definitions and connections between data and space ADTs and DADTs. We then move on to the support for defining the internal operations of the DADTs. Finally we bring back the concepts of views to limit the scope of DADT operations. Although each concept is introduced with the language elements supporting it, our focus and main contribution is in the DADT model they support.

3 Basic Concepts

We chose to cast our ideas into the notion of *abstract data type*, as it represents a well-understood programming concept and is general enough to allow us to present our novel constructs without the distractions and idiosyncrasies of a specific programming language or model. While our presentation remains abstract, Section 6 describes an instantiation of the DADT concept for the Java language.

Data and space ADTs. At the core of our model is the notion of ADT. As mentioned in the example, we draw a sharp line between the data necessary to the application behavior, and the space where such data resides, accordingly distinguishing between data ADTs and space ADTs.

Data ADTs are conventional ADTs encoding the application logic, e.g., `Sensor` in Figure 1. Instead, space ADTs (or *sites*) represent and characterize an abstract notion

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

Fig. 3. A space ADT representing a physical location.

of the computational environment hosting data ADT instances, e.g., a computer, virtual machine, a PDA, or a car. Many application-dependent notions of space are meaningful. 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 empower the programmer with the explicit ability to use (and even define) the notion of space that is most appropriate for the application. For instance, mobile applications may make use of network topology, and represent it as a collection of sites with appropriate operations. As such, the notion of site built in our approach is minimalistic, consisting of an ADT `Site` that must be specialized by the programmer. For instance, Figure 3 shows a space ADT whose position is characterized by a physical location. How this location is acquired and defined (e.g., from GPS) is entirely encapsulated in the ADT implementation.

The only syntactic difference between data and space ADTs is the use of the keywords `datatype` and `spacetype`, essentially to enable type checking and improve 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 in the DADT interface. Returning to our example, the declaration of the `DSensor` DADT in Figure 2 enables distributed access to instances of the ADT `Sensor` defined in Figure 1, through the `resetAll` and `average` operations. Similarly, one can define a space DADT for collective access to a set of sites. For example, later on (in Figure 8) we define a `WirelessNetwork` space DADT that allows collective access to the configuration of network hosts.

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 defines a new relationship among ADT types, analogous to 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 application programmer—which in large development efforts is likely to be different from both the ADT and DADT programmers—can manipulate the distributed state exported by ADT instances by creating a DADT instance and invoking its operations. For example,

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

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

creates a `DSensor` instance and uses it to trigger the distributed processing encapsulated in its `average` operation, as specified by the DADT programmer with the constructs illustrated in Section 4. Note how the invocation above is indistinguishable from any other on a conventional ADT instance: the application programmer may even be unaware of the distributed nature of the reference `ds`.

The DADT member set. Analogously to ADTs, a DADT may declare attributes, which may assume different values for each of its instances. However, every DADT instance always encapsulates the distributed state constituted by the set of ADT instances available for distributed computation through its operations, as a sort of implicit attribute. This set of ADT instances is called the *member set* of the DADT, and its 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 in any distributed system, as asynchrony and concurrency complicate the task of maintaining globally consistent state. Moreover, different applications may allow different, weaker notions of member set consistency. For this reason, we assume that the underlying run-time provides *best effort* consistency. As a consequence, different DADT instances may have different values of the DADT member set. At the same time, however, as we discuss in Section 6, the flexible architecture of our DADT run-time provides mechanisms to enable customization of the middleware with alternative consistency algorithms.

Binding an ADT into a DADT. Although we defined the notion of member set, we did not discuss how ADT instances become part of it. In our model, this is represented as a binding between each instance of the ADT and the DADT. This can be explicitly managed on a per-instance basis by the application programmer with bind and unbind operations, as in `bind(new Sensor(),"DSensor")`. However, this constitutes also the basic functionality for higher-level automatic mechanisms, e.g., to automatically bind ADTs to DADTs based on information about types and the `distributes` relation.

Note how an ADT instance is bound to a DADT *type*, not to a specific *instance*. In fact, it is the DADT name that serves to identify collectively a group of ADT instances; DADT instances, instead, provide an *entry point* towards this group. Moreover, while the effect of binding (and unbinding) is global, as mentioned previously the model does not guarantee a globally consistent view of the resulting member set, and therefore these entry points may have different values of the member set.

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 ones. The ADT instances in the figure are bound to a DADT and 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 binding. Note that some ADT instances are left unbound either explicitly by the application programmer or by some property of the implicit binding. Collective access to the member set is enabled through a DADT instance (represented as a star), which serves as the entry point to the member set.

While this allows access to either data or space, we have not yet discussed how the two are related. Conventional programming languages implicitly bind data to space

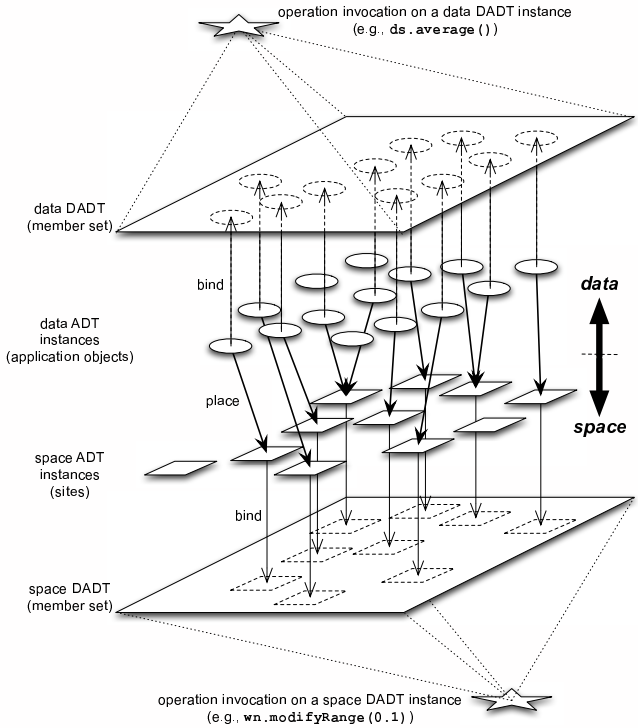

Fig. 4. Data and space in the DADT model.

when an object is created, including it in the local "computational environment". As we abstract the latter into a space ADT instance (site), we need to establish the proper binding between a data ADT instance and the site where it resides. In our model, this is achieved using the `place` operation, represented by the thick lines in Figure 4. It can be invoked explicitly by the programmer by binding data ADT and space ADT instances co-located in the same computational environment. For example, if `g` is a `GPSSite`, then `place(new Sensor(),g)` binds a newly created sensor to the existing site `g`. For applications that do not require this degree of flexibility, a default site can be specified and every newly created data ADT instance is automatically *placed* on it. In either case, the explicit definition of a site allows symmetric treatment of data and space.

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. This can be supported with two space DADTs, one for each network interface. Hosts supporting both interfaces should bind to both DADTs. For the sake of simplicity the remainder of this paper assumes a site is bound to a single space DADT, however the same assumption is not made for data.

```
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.

Data and space ADTs can be regarded as the basic elements of the world relevant to the application, that can be manipulated globally through the "knobs" provided by the DADT definition. In doing this, the application programmer is free to choose the perspective (data or space) more appropriate for the functionality at hand. For instance, she can request a global average through a data DADT but modify the communication range of nodes through a space DADT, as shown in Figure 4. The two perspectives are coherently integrated in a single model, as the `bind` and `place` operations effectively connect data and space ADTs among themselves and to the corresponding DADT planes. Moreover, the model is further enriched by the ability to limit the portion of the system affected by this global manipulation through view definition, again based on data and/or space concerns. Before delving in this subject, however, we need to illustrate distributed access to ADTs.

4 Distributed Access to ADTs

After the DADT's interface and target ADT are declared, as shown for instance in Figure 2, the DADT behavior realizing distributed, transparent access must be defined by specifying the body of the DADT methods. Appropriate constructs are needed to access and manipulate the distributed state in the DADT member set. For this, we introduce two programming constructs: *operators* and *actions*.

4.1 Operators

We start by focusing on the task of implementing the `resetAll` method of `DSensor`, whose intended behavior is to reset all the sensors in the system. Since the DADT operates on a set of ADT instances, one would like to specify the desired behavior by operating on the set in a declarative way. For instance, in a Z-like formal language, with

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

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 invocation target. Figure 5 also shows the implementation of the `average` method, where the results of the various invocations 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 available for distributed processing. At this point of our presentation, the target set always coincides with the member set, however this changes when DADT views are 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, *selectors*, because they allow selection of a subset of the instances contained in the target set.

Selectors 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 provide *condition operators*, which can be used to make the code of a DADT method dependent on a global condition over 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 is possible to rewrite `resetAll` so that it resets all the sensors only if a given "master" sensor (whose identifier we assume known) is not available, and rewrite `average` to compute the average only if more than 3 sensors are around:

```
if (!({master} in? targetset)) (all in targetset).reset();
if ((# targetset)>3) readings = (all in targetset).read();
```

We support also other operators, e.g., to explicitly access a set of ADT instances based on their identifiers as in `({a1,a2,a10} in targetset).read()`, and a notion of *iterator*. The latter enables the programmer to access the ADT instances one at a time rather than interacting with all of them at once, which in some cases may reduce communication overhead. Due to space limitations we redirect the interested reader to [10].

Other operators beyond those discussed here can be defined. Examples include a variant of `any` that non-deterministically selects a given number of instances (e.g., `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 described thus far, as well as the required mechanisms to enable the programmer to define her own, as illustrated in Section 6.

4.2 Actions

The use of operators enables concurrent access to remote ADT instances and thus far we have assumed that such access occurs by invoking one of the ADT's operations. However, in many cases, this is not sufficient. For instance, assume the ADT's `read` operation is capable of signaling a malfunction by returning an `ERROR` value. In this

```
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. 6. Accessing remote ADTs using actions.

case, it may be reasonable to circumvent transient faults (e.g., due to interference of the sensor with physical phenomenons) with simple countermeasures, e.g., 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 implement this behavior we could leverage off the operators presented, by attempting a distributed `read` operation (e.g., using the `all` operator as in line 6 of Figure 5) and then dealing with faulty nodes. However, this may be inefficient. If the `all` operator is used again to retry the reads and possibly reset sensors, its effect is global instead of being limited to faulty sensors. On the other hand, if faulty sensors are singled out by using the aforementioned ability to access explicitly a given sensor based on identifier, a great deal of communication is generated because retries and reset operations need be invoked from the (remote) DADT instance. 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 an operation that is defined in the DADT but whose execution occurs local to the ADT instance on which it is invoked without requiring additional communication. Loosely speaking, actions enable the programmer to write DADT code that operates on ADT instances as if they were exporting a richer interface.

Figure 6 illustrates the concept. The action declaration is contained in lines 2–13, and is identical to the one for 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, given 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 of the ADT. These semantics can be visualized by considering actions as mobile code [4] 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 6 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 (line 14) using the notation described in Section 4.1. 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.

5 Restricting the Scope of Operations

Distributed sharing of ADTs as we have defined it thus far is a powerful concept. However, the invocation of an operation over *all* the instances in the member set may not always be desirable. For example, performance reasons may render it impractical or application needs may suggest policies to constrain an operation to a subset of the ADT instances.

Our reference example helps to clarify these concepts. In our scenario, different kinds of sensors (e.g., temperature, light, humidity) may be present, accounted for in Figure 1 with the attribute `sensorType`. As discussed so far, the sharing rules include all instances of a given ADT, therefore including multiple types of sensors in the same `average` computation, yielding a meaningless result. Instead, we would like to select a subset of the available sensors, namely 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 an abstract sensor. Unfortunately, subtyping is not possible for other characteristics of a sensor, such as those representing a portion its state, as opposed to a static characteristic like the sensor type. For instance, the laptop monitoring station in our example may need to reset all the sensors currently inactive or to compute the average over only sensors guaranteeing a given resolution. For these cases one would like to specify something such as

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

We provide this level of flexibility and expressiveness in DADTs through the notions of property and view. 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 boolean operations. Figure 7 shows the `DSensor` defined earlier augmented with properties. For instance, `isActive` returns true if the corresponding local attribute (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

```
1  datatype DSensor distributes Sensor with {
2    properties:
3      bool isActive();
4      bool isSensorType(int sensorType);
5      bool isPrecise(double resolution);
6    actions:
7      double[] reliableRead();
8    operations:
9      double average();
10     void resetAll();
11 }
12 bool DSensor::isActive() {
13   return local.isActive;
14 }
15 bool DSensor::isSensorType(int sensorType) {
16   return local.sensorType == sensorType;
17 }
18 bool DSensor::isPrecise(double resolution) {
19   return local.resolution >= resolution;
20 }
21 void DSensor::resetAll()  {/* ...as in Figure 5... */}
22 double DSensor::average() {/* ...as in Figure 5... */}
```

Fig. 7. Restricting the scope of operations over data.

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 dynamic projections over the member set, called DADT *views*. Views are defined by the DADT programmer using properties and dedicated constructs. For instance,

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

defines the view `active` as the subset of the member set of `DSensor` that contains only those `Sensor` instances for which the evaluation of the property `isActive` yields true. The DADT name refers 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 to restrict dynamically the scope of a DADT operation. For instance,

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

resets all sensors that are currently inactive. The execution semantics 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.

Just as we used negation in the above formula, we support the general ability to compose views by connecting properties with boolean operators. While the mechanics are simple, the feature itself is powerful as it allows one to express views using union and intersection. For instance,

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

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

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

Fig. 8. A space DADT providing distributed access to sites.

Additional features of our model enable the definition of *asymmetric* views. The content (i.e., ADT instances) of these views, unlike those defined thus far, depends on attributes of the specific DADT instance on which the operation restricted by the view is invoked. Although this feature increases the expressiveness of view definition, space limitations force us to refer the reader to [10] for further details.

If no view is specified at invocation time, the operation is performed on the whole member set, as discussed in Section 4.1. Indeed, the member set defines the most general view containing all 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 *inside* the definition of an operation (as specified by the DADT programmer), but also *outside* the DADT definition (as specified by the application programmer). For instance, the following program fragment reliably retrieves the values sensed by all the active temperature sensors using the action whose code[5] appeared in Figure 6, provided that at least three of them provide sufficient resolution:

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

All the considerations made thus far hold both for *data* and *space* views. For example, in our motivating scenario, the laptop requests an operation to be executed over nodes *within proximity*. This is accomplished in a two-step process, first defining the site, then the view. Figure 8 shows the definition of the `WirelessNetwork` space DADT. It defines a property `isReachable` that yields true if the target site is within a specified number of hops. A space view can represent proximity based on this property. This view can be used to restrain not only operations on a data DADT, but also on a space DADT, e.g., to reduce by 10% the communication range of nearby hosts as in:

```
spaceview proximity on WirelessNetwork as isReachable(2);
wn.modifyRange(-0.1) within proximity;
```

where `wn` is an instance of the `WirelessNetwork` DADT. Note how `proximity` is asymmetric, as it depends on the location of the DADT instance on which it is invoked.

One of the most powerful uses of views arises in the combination of data and space views, as in:

```
double v = ds.average() on temperature within proximity;
wn.modifyRange(-0.1) on temperature within proximity;
```

which combine the `temperature` data view with the `proximity` space view to return either the proximal average temperature with an operation over data, or reduce the communication range of proximal temperature sensors with an operation over space.

[5] In Figure 7 the action is declared in the interface of `DSensor`, contrary to Figure 6, and is therefore visible to and can be reused by any of the DADT operations and by the client objects calling them. The action code is encapsulated in the DADT, thus providing information hiding.

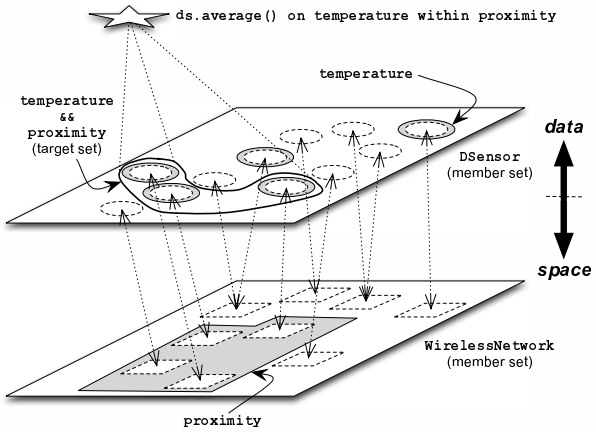

Fig. 9. DADT views.

The content of `targetset` inside the body of the invoked (data or space) DADT operation is computed as the overlap of the subsets defined by the two views. Figure 9 illustrates the concept w.r.t. the first of the two statements above. Different from Figure 4, data and space ADT instances are not shown—only their "projection" on the member sets and the place relation between data ADTs (circles) and sites (squares).

As 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 properties 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" (data or space) for accessing the distributed system. In our model, data and space become easily interchanged during the programming practice, with our model coherently maintaining their semantic interaction.

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 precompilation 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 runtime 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

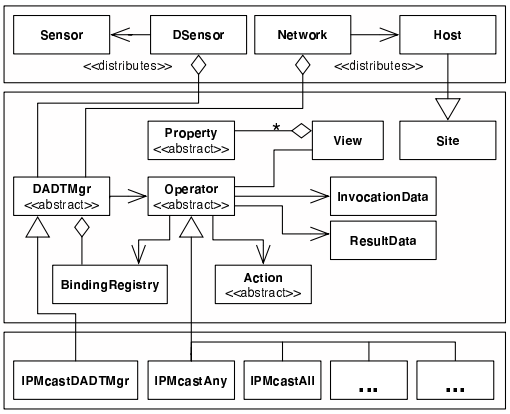

Fig. 10. The architecture of the $\mathcal{J}$DADT run-time.

grammar by Studman [12] 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 10. The top layer is composed of 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 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, to specify the `Site` instance abstracting the *node*[6] 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 `BindingRegistry` object associated to the `DADTMgr` contains information about which node's ADT is bound to which DADT, provides methods to determine the local ADT instances that satisfy a given view specification, and contains information about the available operators.

The abstract classes `Property` and `Action` represent the corresponding concepts. Both represent 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 7. The translator generates the 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;
  }
}
```

[6] The term *node* here refers to the computational environment where the run-time executes—a JVM in our implementation. This is not be confused with a *site*, which is a node's abstract representation as a space ADT.

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 (`Sensor` in our case) distributed by the DADT. This can be safely substituted for the keyword `local` used in Figure 7. A similar strategy is used for translating actions. `Property` and `Action` objects are created upon action invocation or view definition, serialized, and transmitted on the node of the target ADT instance. There, their `evaluate` method is executed by using this ADT instance as a parameter, to enable invocation of its operations. Note that here we assume that the DADT class is pre-deployed on the interested nodes. A more dynamic and open design is easily obtained with a "code on demand" paradigm [4], enabling class relocation only if and when needed.

A view (on data, space, or both) is represented by a `View` object containing the set of `Property` objects involved in the view definition, properly arranged in an abstract tree representing the view predicate. To determine whether a given ADT instance belongs to the view, the abstract tree is navigated from the leaves (the property objects) to the top (the view object), by invoking the `evaluate` method of each property and composing it with others according to the boolean operator in each abstract tree node. The process terminates at the root with the boolean result of the evaluation. Composition of views is easily achieved by composing their predicates, i.e., their abstract trees. Finally, a view object's `apply` method allows execution of an action on the ADT instances selected by an operator.

Operators, both built-in and programmer-defined, are subclasses of `Operator`. The corresponding implementation is retrieved at run-time from `DADTMgr`, using a Factory pattern. The operator behavior is specified by overriding only two methods. `evalRemote(View v)` is invoked on the caller node (e.g., where `ds.average` is called), and embeds the logic for distributing the information necessary to the collective evaluation of the operator on the view, and the retrieval of the results. Instead, `evalLocal(InvocationData d)` is performed on all the other nodes involved in the computation (i.e., those hosting the target ADT instances), and contains the logic for evaluating the operator local to a single node and sending the results back to the caller. The parameter contains the information representing an invocation, i.e., the view specification, the operator to be applied, the action to be executed, and the caller's identifier. Both methods above return results packed in a simple `ResultData` class. In essence, `Operator` subclasses embed all the distribution logic connecting the caller node and the ADT instances. In our prototype, the implementation of the `all` operator leverages off UDP unicast and multicast sockets, as shown in the bottom layer of Figure 10 containing the classes specializing our framework. Multicast is exploited for dispatching invocations to the ADT instances bound to a given DADT, while UDP is used for returning results. More details in [10].

To clarify how the various components cooperate, consider the following sample DADT program statements

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

using the fault-tolerant definition of `average` from Figure 6. The translator generates the `Property` subclasses based on the definition of the two views, creates the corresponding objects, and inserts 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, it transforms the signature of `average` into

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

In our case the target set is the conjunction of `preciseOn` and `proximity`, and is passed upon method invocation:

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

Note how a new `View` instance is generated on the fly to represent the clause `on...within...` of the invocation by merging the data view and the space view. In our implementation, both are represented by `View` objects, which are then composed like properties. Different constructors are provided to create a view out of its properties or based on already existing views. Moreover, the translator generates an `Action` subclass representing the action `reliableRead` in the figure, as we described earlier. Line 14 of Figure 6, containing the action invocation in conjunction with the `all` operator, becomes

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

The body of `apply` retrieves from the node's `DADTMgr` the proper implementation of the operator based on the name 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.evalRemote(
    new InvocationData(this,name,a,caller));
... }
```

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

The design just described is conceived as a *framework* in the object-oriented sense, and can be customized (e.g., to use different communication protocols) by redefining only two classes, i.e., `Operator` and `DADTMgr`. Moreover, although $\mathcal{J}$DADT is a proof-of-concept prototype, its realization is still non-trivial. It is fully decentralized and, although thus far we have tested it 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 network routing protocols (e.g., for MANET). Nevertheless, it can clearly be improved in many respects. Most notably, we are currently studying distributed algorithms for efficiently managing 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 [8] and EgoSpaces [7]), for which DADTs actually provide a generalization.

7 Related Work

The closest works come from the context of parallel systems. Shared ADTs (SADTs) [5] focuses on providing implementations of several standard data types, whose implementation is designed to scale well in the parallel environment. Concurrent Aggregates (CAs) [3] 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 do 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.

Recently, the Spatial Programming model [2] has been proposed for embedded systems. It makes an analogy between space and memory and exposes space to applications through spatial references, which enable the definition of regions including components interested in a given computation. Similar ideas have also been applied to mobile ad hoc networks in the SpatialViews language [9], where *virtual networks* can be defined depending on the physical location of a node and the services (expressed as Java interfaces) it provides. Common to our work is the notion of scoping virtual networks provide. However, DADTs provide a clean separation between data and space and the notion of data views, not supported by the aforementioned approaches, allows us to limit execution not only based on spatial constraints, but also on application ones, with the two nicely integrated in our model. Finally, unlike our approach, computation is distributed across nodes in a virtual network by migrating mobile agents from node to node, and this is an integral part of the model. Instead, in our model we provide simultaneous access to multiple instances through a single DADT reference, and mobile code is only one of the implementation options.

Some aspects of our work may look reminiscent of distributed object platforms (e.g., CORBA or Emerald [6]). 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.

Finally, the idea of sets as a programming abstraction was pioneered in the early 70's (e.g., [11]). However, the aim and results of these approaches are profoundly different from ours. Indeed, their goal was to use sets as the fundamental abstraction for *all* programming tasks. Instead, here we use a (much less powerful) set abstraction only to deal with distribution, and rely on standard imperative programming constructs for the dealing with the other concerns.

8 Conclusion

Developing distributed applications is a complex task, especially when the physical space must be taken into account, 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.

Acknowledgments. The authors wish to thank Domenico Bianculli for his work on the implementation of the translator. The work described in this paper was partially supported by the European Union under the project IST-004356-RUNES and by the Italian Ministry of Education, University, and Research (MIUR) under the VICOM project. Roman was supported in part by the US Office of Naval Research under ONR MURI research contract N00014-02-1-0715. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the sponsoring agencies.

References

1. ANTLR Web page. `www.antlr.org`.
2. C. Borcea, C. Intanagonwiwat, P. Kang, U. Kremer, and L. Iftode. Spatial programming using smart messages: Design and implementation. In *Proc. of the 24th Int. Conf. on Distributed Computing Systems (ICDCS)*, March 2004.
3. A.A. Chien and W.J. Dally. Concurrent Aggregates (CA). In *Proc. of the Symp. on Principles & practice of parallel programming*, pages 187–196, 1990.
4. A. Fuggetta, G.P. Picco, and G. Vigna. Understanding Code Mobility. *IEEE Trans. on Soft. Eng.*, 24(5), 1998.
5. D.M. Goodeve et al. Toward a model for shared data abstraction with performance. *J. of Parallel and Distributed Computing*, 49(1):156–167, 1998.
6. 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, February 1988.
7. C. Julien and G.-C. Roman. Active Coordination in Ad Hoc Networks. In *Proc. of COORDINATION*, 2004.
8. G.P. Picco, A.L. Murphy, and G.-C. Roman. LIME: Linda Meets Mobility. In *Proc. of the Int. Conf. on Software Engineering*, pages 368–377, May 1999.
9. Y. Ni, U. Kremer, A. Stere, and L. Iftode. Programming ad-hoc networks of mobile and resource-constrained devices. In *Proc. of the ACM SIGPLAN Conf. on Programming language design and implementation (PLDI)*, 2005.
10. G.P. Picco, M. Migliavacca, A.L. Murphy, and G.-C. Roman. Distributed Abstract Data Types. Technical report, Politecnico di Milano, 2004. Available at `www.elet.polimi.it/upload/~picco`.
11. J. T. Schwartz et al. *Programming with sets; an introduction to SETL*. Springer, 1986.
12. M. Studman et al. Java 1.5 Grammar. `www.antlr.org/grammar/1109874324096/java1.5.zip`.