Distributed Middleware Services Composition and Synthesis Technology

Miklós Maróti, Péter Völgyesi, Gyula Simon
Gábor Karsai and Ákos Lédeczi
Institute for Software Integrated Systems
Vanderbilt University
Box 1829, Station B
Nashville, TN 37235, USA
615-343-7472
{miklos.maroti, peter.volgyesi, gyula.simon,
gabor.karsai, akos.ledeczi}@vanderbilt.edu

Abstract— The highly distributed and resource constrained nature of computing in Networked Embedded Systems necessitates an application specific middleware—a kind of distributed operating system that provides global services for the application. We propose to automatically synthesize the middleware from abstract, platform-independent algorithm models. The modeling language captures the temporal and computational aspects of the distributed algorithms in a programming language-independent and platform-neutral way. It supports the specification, composition and verification of middleware components, and allows the integration of existing platform-specific components. We have implemented a proof-of-concept prototype modeling environment, and used it to model and generate the middleware for a structural vibration damping application running on an I/O automata-based Java simulator, and for a cooperative acoustic tracking application running on TinyOS. The proposed formalism allows the creation of a platform-independent library of middleware services that can be used to build and synthesize various application-specific middleware instances.

Table of Contents

1. Introduction

Networked Embedded Systems Technology (NEST) is becoming an increasingly important field of research as advances in digital circuitry and Micro ElectroMechanical Systems (MEMS) enable the design and large volume fabrication of remarkably compact autonomous nodes with computation and communication capabilities, one or more sensors and actuators, and a power supply. These systems are tightly coupled to physical processes, distributed across a large number of densely deployed processing nodes ($\sim$100,000s) that have limited resources and communication capabilities. Each node has one or more sensors and actuators directly attached to it that interact with the physical process at the node's location. Possible applications of NEST include large-scale active noise control, active flow control over aircraft wings, micro-satellite constellations, smart structures and others.

The Smart Dust project [8] at UC Berkeley aims to incorporate the requisite sensing, communication, and computing hardware, along with a power supply, in a volume no more than a few cubic millimeters. The missing ingredient is the middleware and applications layers needed to harness this revolutionary capability into a complete system [6].

The highly distributed nature of computing in a NEST application implies that each node should be equipped with sophisticated middleware -a kind of distributed operating system that provides global services for the applications (in addition to the local operating system that supports local resource management). This middleware layer is a key ingredient of NEST applications: it encapsulates services that are reusable across a number of specific problems, yet independent of the underlying hardware infrastructure (which is managed by the local OS).

The middleware is expected to support various coordination services beyond basic communication protocols. Coordination services range from simple event- and time-based coordination to complex algorithms for leader election, spanning tree formation, protocols for distributed consensus and mutual exclusion, distributed transactions, group communication services, clock synchronization and others. These services go beyond the

0-7803-7231-X/01/$10.00/© 2002 IEEE

usual capabilities provided by networking protocols. Additionally, because of the inherent unreliability of the nodes and communication links, aspects of fault tolerance must also be addressed by the middleware. Because of resource limitations, a complex, monolithic middleware layer that contains all services for all applications is not feasible. The middleware layer for NEST applications needs to be thin, application-specific and high-performance.

This paper describes the Distributed Services Composition and Synthesis Technology (DISSECT) environment, a proof-of-concept prototype of our middleware modeling, composition and automatic synthesis technology. It enables the programming language-neutral specification of individual middleware services at the algorithm level in a target platform-independent manner. It allows the construction of application-specific middleware assemblies by hooking up the individual services, while making sure that only services with compatible interfaces are used. Finally, it is able to automatically synthesize the middleware layer for two different target platforms: one is our own distributed system simulator with an Asynchronous I/O automaton-based middleware model implemented in Java; the other is a wireless networked sensor platform running TinyOS, a composable micro OS implemented in C and supplied by UCB. We have successfully demonstrated this technology using a structural vibration-damping problem with 50 nodes on our simulator and a cooperative acoustic tracking application running on the actual wireless sensor network using 10 real nodes. In both cases, the middleware layer was generated from high-level graphical specifications fully automatically. The performance was comparable to that of hand written code.

2. The notion of algorithm

There are different notions of *algorithm*. On the one hand, an algorithm is an intuitive idea that you have in your head before writing code. The code then implements the algorithm. The same algorithm may be codified in different programming languages, using different frameworks and libraries, and can run on different platforms. A *framework* is a set of constraints on components and their interactions, together with a set of benefits that derive from those constraints. We use different levels of abstraction to describe an algorithm, such as:

- machine executable code for a particular platform,
- soucre code written in high level programming languages using different frameworks and libraries,
- executable model under a particular model of computation,
- input-enabled I/O automaton with nondeterministic execution order,
- structural I/O automaton representation.

The undeniable success story of Computer Science is *compilation*, the generation of machine executable code from source code and libraries, proving that it is feasible to abstract the platform from the algorithm specification. However, the proliferation of programming languages and component/communication frameworks necessitates the repeated implementation of the same algorithms. This just proves that the specification of algorithms at the source code level is rigid and contains elements that are not part of the algorithm.

A new level of abstraction was introduced to separate the algorithm from the programming language: executable algorithm models that support simulation and/or code generation (see Ptolemy [9]). The models are constructed under a *model of computation*, a set of "laws of physics" that govern the interaction of components in the model. The timing and execution order of actions is handled by the model of computation. While Ptolemy allows the composition of components using different models of computation, the semantics of the resulting component is not well defined in some cases.

The modeling of asynchronous distributed algorithms by I/O automata was proposed by N. Lynch in [1]. I/O automata are *input-enabled*, meaning that an automaton is not able to somehow "prevent" input actions from occuring. Following the *optimistic* approach to composition [2], we believe that an automaton shall not only describe its behavior, but its expectation on the environment, as well. While the description of an automaton is abstract, the description of basic transitions is usually given in a high level programming language, and assumed to be executed atomically. The I/O automata methodology is well studied and a large collection of distributed algorithms are expressed and verified in it.

We aim to merge the formalisms of executable models (programming language-independence, implementability), interface automata (optimistic approach to composition) and I/O automata (nondeterministic execution order). Our vision is to abstractly define algorithms that are verifiable, reusable and effectively implementable in a platform-, programming language- and framework-independent way.

3. The I/O automata

Our modeling language is based on the definition of I/O automaton, which we will reproduce here.

Definition 1: An I/O automaton A consists of five components:

- $acts(A)$, a set of *actions* which is partitioned into three disjoint sets: $in(A)$, $out(A)$ and $int(A)$, the sets of *input, output* and *local* actions
- $states(A)$, a nonempty set of *states*

- $start(A)$, a nonempty subset of $states(A)$, known as the *start states*
- $trans(A)$, a *state-transition relation*, where $trans(A) \subseteq states(A) \times acts(A) \times states(A)$
- $tasks(A)$, a *task partition*, which is an equivalence relation on $out(A) \cup int(A)$.

Just like with Finite State Machines (FSM), I/O automata have a key weakness in their simple flat form: the number of states and state-transitions can get quite large even for a moderately complex systems. Such models quickly become chaotic and incomprehensible when one tries to understand abstract I/O automata. Indeed, I/O automata specified as in Definition 1 are not meant for human consumption. Some structure is lost which is needed to understand and implement the automaton. It is interesting to look at the current practice of specifying I/O automata: the informal specifications still have structure.

The set of states of an I/O automaton A is usually described in terms of a list of state variables and their initial values. If the domains of these state variables are $D_1, \ldots, D_n$, then

$$states(A) = D_1 \times \cdots \times D_n.$$

The set $acts(A)$ of actions is described in a similar way, although first it is grouped into logically coherent subsets. Specifically, $acts(A)$ is a disjoint union of action groups

$$acts(A) = G_1 \cup \cdots \cup G_m,$$

and each action group G_i is parameterized: described in terms of a list of action parameters. If the number of parameters of G_i is $p(i)$, and the domains of the action parameters are $D_{i,1}, \ldots, D_{i,p(i)}$, then

$$G_i = D_{i,1} \times \cdots \times D_{i,p(i)}.$$

The set of state transitions are also grouped into logically coherent subsets, that is, $trans(A)$ is a disjoint union of transition groups

$$trans(A) = T_1 \cup \cdots \cup T_l.$$

The transition groups are, in turn, described in a semi-formal programming language. It is not a big surprise that these program fragments read and write state variables and action parameters. Typical specifications use conditional and iterational statements, such as the $decide(v)_i$ transition of the $Process_i$ I/O automaton (pp. 205, [1]):

```
$decide(v)_i$
  Precondition:
    for all $j$, $1 \leq j \leq n$:
      $val(j) \neq null$
    $v = f(val(1), \ldots, val(n))$
  Effect:
    none
```

and the $receive(\text{“bcast”}, w)_{j,i}$ input transition of the $AsynchBcastAck_i$ I/O automaton (pp. 499, [1]):

```
$receive(\text{“bcast”}, w)_{j,i}$
  Effect:
    if $val = null$ then
      $val := w$
      $parent := j$
      for all $k \in nbrs - \{j\}$ do
        add (“bcast”, $w$) to $send(k)$
    else add “ack” to $send(j)$
```

We intend to formally define *structured sets*, the sets we use for describing $states(A)$ and $acts(A)$.

Definition 2: Structured sets are defined recursively:

- The domain of basic datatypes are structured sets.
- Finite products of structured sets are structured.
- Disjunct unions of structured sets are structured.
- Finite powers of structured sets are structured.
- The Kleene[1] star of structured sets is structured.

We classify the structured sets into five types, called *variables, products, unions, arrays* and *queues*, according to the above rules, respectively. These compositional operators are well known in both mathematics and computer science.

To achieve platform and programming language-independence, we limit the complexity of the specification of transition groups as much as possible. Typically, a specification uses only a limited number of state variables and action parameters. We took the following minimalist approach: We call a transition group specification *simple* if it is of the form

```
$transition(arg_1, \ldots, arg_n)$
  Precondition:
    $cond_1(par_1, \ldots, par_n, var_1, \ldots, var_m)$
      ...
    $cond_k(par_1, \ldots, par_n, var_1, \ldots, var_m)$
  Effect:
    $var_{m+1} = expr_1(par_1, \ldots, par_n, var_1, \ldots, var_m)$
      ...
    $var_{m+l} = expr_l(par_1, \ldots, par_n, var_1, \ldots, var_m)$
```

where $par_1, \ldots, par_n$ are action parameters, $var_1, \ldots, var_m$ are state variables, $cond_1, \ldots, cond_k$ are simple comparisons (the comparison relation is either $=$, $<$, $>$ or $<>$,

[1] the union of all finite powers of a set: $\{\emptyset\} \cup A \cup A^2 \cup \ldots$

and each side is either $par_1, \ldots, par_n$, $var_1, \ldots, var_m$ or a constant), and $expr_1, \ldots, expr_l$ are basic expressions using only basic arithmetic operators.

The use of simple conditions and basic expressions allows programming language-independence, but we still face the problem of accessing the state variables and action parameters in deeply nested structured sets. We use the notion of data ports to disassemble structured sets into less complicated structured sets:

Definition 3: Let S be a structured set. A *data port* P of S consists of four components:

- $domain(P)$, a structured set of *exported* elements
- $hidden(P)$, an unstructured set of *hidden* elements
- $select(P)$, an idempotent mapping of S into itself, that is, $select(P) \circ select(P) = select(P)$
- $decomp(P)$, a bijection between the range of $select(P)$ and $domain(P) \times hidden(P)$.

We say that a data port P can *read* an element $s \in S$, if $select(P)(s) = s$, that is, if s is in the range of $select(P)$. Reading the data port of a structured set S at an element s is equivalent to the transition:

```
$read_{P,s}(w), w \in domain(P)$
  Precondition:
    $select(P)(s) = s$
    $decomp(P)(s) \in \{w\} \times hidden(P)$.
```

Writing to the data port is always possible: it first brings the current state of the automaton into the range of $select(P)$, then calculates the hidden component of the state, and finally it composes the new state from the new domain value and the hidden state. More precisely, writing to the data port of a structured set S at an element s is equivalent to the transition:

```
$write_{P,s}(w), w \in domain(P)$
  Effect:
    $s := select(P)(s)$
    $(v, h) := map(P)(s)$
    $s := map(P)^{-1}(w, h)$.
```

As an example, consider the following I/O automaton A:

```
States:
  $v$, integer
  $h$, real number
  $c$, strign

Transitions:
$read(w)$
  Precondition:
    $c =$ "hello"
    $w = v$
  Effect:
    none

$write(w)$
  Effect:
    $c :=$ "hello"
    $v := w$
```

The $read(w)$ and $write(w)$ transitions correspond to $read_{P,(v,h,c)}(w)$ and $write_P(w, (v, h, c))$, respectively, P is the data port defined as

- $domain(P) = \{$ the set of all integers $\}$
- $hidden(P) = \{$ the set of all real numbers $\}$
- $select(P)$ maps $(v, h, c) \in states(A)$ to $(v, h,$ "hello"$)$
- $decomp(P)$ maps an element $(v, h,$ "hello"$)$ of the range of $select(P)$ to (v, h).

Now we can define simple state-transition relations. This definition differs from that of simple transition groups in that it reads and writes the action parameters and state variables through data ports.

Definition 4: Let A be an I/O automaton. We call a set $T \subseteq states(A) \times trans(A) \times states(A)$ a *simple state-transition relation*, if it is described by a simple transition group of the form:

```
$transition(action)$
  Precondition:
    $read_{P_1,action}(arg_1)$
      ...
    $read_{P_n,action}(arg_n)$
    $read_{P_{n+1},state}(var_1)$
      ...
    $read_{P_{n+m},state}(var_m)$
    $cond_1(par_1, \ldots, par_n, var_1, \ldots, var_m)$
      ...
    $cond_k(par_1, \ldots, par_n, var_1, \ldots, var_m)$
  Effect:
    $write_{P_{n+m+1},state}(expr_1(par_1, \ldots, par_n, var_1, \ldots, var_m))$
      ...
    $write_{P_{n+m+k},state}(expr_l(par_1, \ldots, par_n, var_1, \ldots, var_m))$
```

where $P_1, \ldots, P_n$ are dataports of $acts(A)$, $P_{n+1}, \ldots, P_{n+m+k}$ are data ports of $states(A)$, and $cond_1, \ldots, cond_k$ and $expr_1, \ldots, expr_l$ are simple conditions and basic expressions.

4. Structural model of I/O automata

We will introduce basic I/O automata serving as building blocks. Then define compositional rules to obtain complex automata. To support composition (through data ports), we need to extend the definition of I/O automata. We call the new concept structural I/O automaton.

Definition 5: A *structural I/O automaton* A consists of eight components:

- $acts(A)$, a structured set of *actions*
- $states(A)$, a structured set of *states*
- $start(A)$, a nonempty subset of $states(A)$, known as the *start states*
- $trans(A)$, a *state-transition relation*, where $trans(A) \subseteq states(A) \times acts(A) \times states(A)$
- $in(A)$, a set of data ports of the set $acts(A)$
- $out(A)$, a set of data ports of the set $acts(A)$
- $data(A)$, a set of data ports of the set $states(A)$
- $tasks(A)$, a *task partition* on $acts(A)$.

Notice, that we use structured sets for actions and states, the partition of actions into input and output actions is replaced with two list of data ports. The biggest change is that we have introduced a new kind of interaction mechanism: a list of data ports of $states(A)$ which allows direct manipulation of states of the automaton. This "feature" is needed by the most simple I/O automaton: the Variable.

The Variable

The variable is the most basic I/O automaton. It has no computational capabilities, it just stores a single value of a simple datatype T. Here is the definition:

- $acts(Variable_T) = \emptyset$
- $states(Variable_T) = T$
- $start(Variablt_T) = \{\text{an element of } T\}$
- $trans(Variable_T) = \emptyset$
- $in(Variable_T) = \{\}$
- $out(Variable_T) = \{\}$
- $data(Variable_T) = \{T\}$
- $tasks(Variable_T) = \{\}$.

The Activator

The activator operation allows the introduction of a new simple state-transition relation into a structured I/O automata. The new transitions must attach to an existing or a newly created action data port, and only to existing state data ports. The formal definition of the activated automata B of A is as follows:

- $acts(B) = acts(A) \cup N$
- $states(B) = states(A)$
- $start(B) = start(A)$
- $trans(B) = trans(A) \cup \{\text{simple transition relation}\}$
- $in(B) = in(A) \cup I$
- $out(B) = out(A) \cup O$
- $data(B) = data(A)$
- $tasks(B) = tasks(A) \cup \{\text{simple transition relation}\}$

where N is a new set of actions, P is the data port for N in $acts(B)$, $P_1, \ldots, P_n$ be data ports from $in(B) \cup out(B)$, $P_{n+1}, \ldots, P_{n+m+k}$ are data ports from $data(A)$, and the conditions of Definition 4 are statisfied. Depending whether we wish to export the new action through a data port, the pair (I,O) of sets is either $(\emptyset, \emptyset)$, $(\emptyset, P)$ or $(P, \emptyset)$.

The Product

The product operation allows the composition of a finite list of structured I/O automata into one. This corresponds to the usual composition of I/O automata in [1]. Let $A_1, \ldots, A_n$ be structural I/O automata, and R be a relation from the set $in(A_1) \cup \cdots \cup in(A_n)$ to $out(A_1) \cup \cdots \cup out(A_n)$. The *product* $A = A_1 \times \cdots \times A_n$ is a structural I/O automata defined as follows:

- $acts(A) = acts(A_1) \cup \cdots \cup acts(A_n)$
- $states(A) = states(A_1) \times \cdots \times states(A_n)$
- $start(A) = start(A_1) \times \cdots \times start(A_n)$
- $trans(A) = \bigcup\{Box(states(A), i, s) \times \{a\} \times Box(states(A), i, t) : (s, a, t) \in trans(A_i)\}$
- $in(A) = in(A_1) \cup \cdots \cup in(A_n) - \{\text{the domain of } R\}$
- $out(A) = out(A_1) \cup \cdots \cup out(A_n)$
- $data(A) = data(A_1) \cup \cdots \cup data(A_n)$
- $tasks(A) = \{tasks(A_1), \ldots, tasks(A_n)\}$

Where

$$\begin{aligned} &Box(X_1 \times \cdots \times X_n, i, c) = \\ &\quad \{\langle a_1, \ldots, a_{i-1}, c, a_{i+1}, \ldots, a_n \rangle : \\ &\quad a_1 \in X_1, \ldots, a_{i-1} \in X_{i-1}, a_{i+1} \in X_{i+1}, \ldots, a_n \in X_n\} \end{aligned}$$

We impose certain restrictions on the automata that may be composed using the product. The relation R specifies the interconnections from output data ports to input data ports. We require that for all pairs $(P, Q) \in R$ of data ports, $domain(P) = domain(Q)$.

The Union

The union operation allows the exclusive composition of a a finite list of structural I/O automata into one. Let $A_1, \ldots, A_n$ be structural I/O automata. The *union* $A = A_1 \cup \cdots \cup A_n$ is defined as follows:

- $acts(A) = acts(A_1) \cup \cdots \cup acts(A_n)$
- $states(A) = states(A_1) \cup \cdots \cup states(A_n)$
- $start(A) = start(A_1) \cup \cdots \cup start(A_n)$
- $trans(A) = trans(A_1) \cup \cdots \cup trans(A_n)$
- $in(A) = in(A_1) \cup \cdots \cup in(A_n)$
- $out(A) = out(A_1) \cup \cdots \cup out(A_n)$
- $data(A) = data(A_1) \cup \cdots \cup data(A_n)$
- $tasks(A) = \{tasks(A_1), \ldots, tasks(A_n)\}$

The Array

Conceptually, the array is a shorthand notation of the repeated use of product operation. However, there are substantial differences, because now it is possible to address individual automata of the array. Let A be a structural I/O automaton, and n be a natural number. The n-th power A^n of A is a structural I/O automaton defined as:

- $acts(A^n) = \{1, \ldots, n\} \times acts(A)$
- $states(A^n) = states(A)^n$
- $start(A^n) = start(A)^n$
- $trans(A^n) = \bigcup\{Box(states(A), i, s) \times \{a\} \times Box(states(A), i, t) : (s, a, t) \in trans(A)\}$
- $in(A^n) = \{1, \ldots, n\} \times in(A)$
- $out(A^n) = \{1, \ldots, n\} \times in(A)$
- $data(A^n) = data(A) \cup \{1, \ldots, n\} \times data(A)$
- $tasks(A^n) = \{1, \ldots, n\} \times tasks(A)$

Observe, that the number of input and output action data ports has been multiplied by n. This is because we can "send" an action to each of the n-many components. The state data ports are even more complex. Most notably, we allow the reading and writing of all embedded state data ports simultaneously. "Reading" means checking if all copies have the same state in the observable domain (see Definition 3), and then returning the common value. "Writing" means setting the state of each copies to a common value. One of the reasons behind the complexity of Definition 3 is to allow such constructs. This rule allows the generation of code for "embedded" transitions that use for-loops in their precondition and/or effect.

5. Case study

The Distributed Services Composition and Synthesis Technology (DISSECT) tool is a prototype design environment for middleware development. DISSECT is implemented using the Generic Modeling Environment (GME), a configurable toolkit for creating domain-specific modeling and program synthesis environments [Computer 2001 nov]. The configuration is accomplished through a metamodel specifying the modeling language of the application domain. It contains all the syntactic, semantic, and presentation information regarding the domain - which concepts will be used to construct models, what relationships may exist among those concepts, how the concepts may be organized and viewed by the modeler, and rules governing the construction of models. The modeling language defines the family of models that can be created using the resultant modeling environment.

We created the DISSECT metamodel that implements the concepts described earlier in the paper. The resulting environment will be now presented using an example application. We have designed and implemented a Cooperative Acoustic Tracking (CAT) applications running on the UC Berkeley mote platform and the TinyOS operating system. There are two kinds of motes in this system. A single mote acting as an active tag has a buzzer that can emit a 4KHz sound under program control. N additional motes act as the trackers. Each has a microphone and a 4KHz hardware bandpass filter attached to it. For simplicity, the tracker motes are arranged along a line simulating, for example, cooperative tracking on a road or a hallway inside a building.

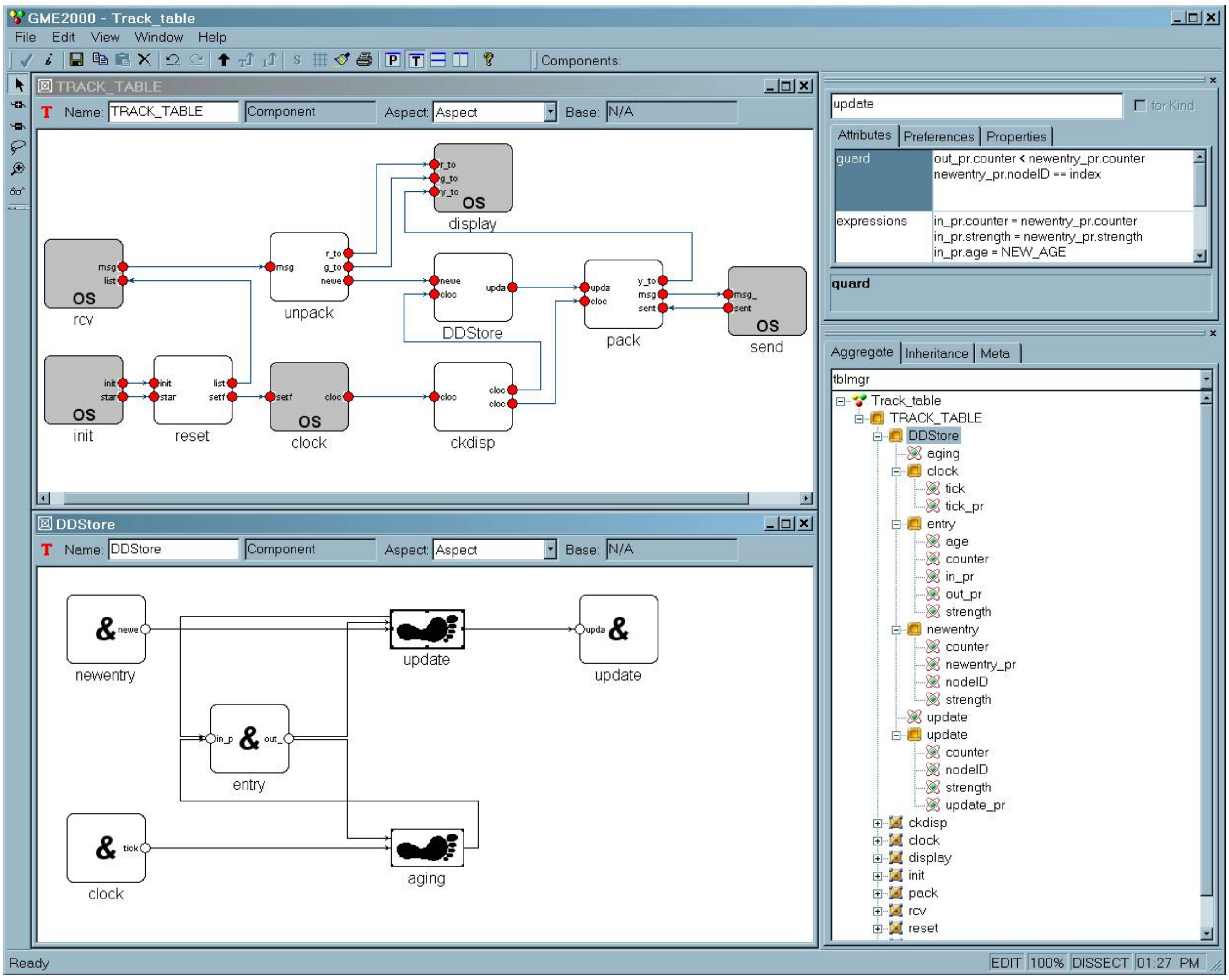
The active tag periodically broadcast a radio message announcing its intention to buzz. Immediately following the radio message, it buzzes for a half a second. The trackers who successfully received the broadcast start measuring the time of flight of the sound. Those motes whose microphones picked up the sound, compute the distance to the tag from the measured time. To decrease the effects of noise, jitter and other sources of error a simple averaging algorithm is also used.

The single middleware service in this simple application is called Track Table. Its task is to maintain a table at each of the trackers containing the latest measurement results of all the trackers. It is accomplished by listening to messages from neighboring nodes, updating their own table if the result is newer than what they already have and broadcasting their entire table periodically. The DISSECT model of the Track Table middleware service is shown in Figure 1.

The figure shows a screenshot of the DISSECT environment. The lower right hand window shows the hierarchical structure of the model in a tree-like fashion. The upper right hand window shows the textual attributes of the selected model element, in this case, the guard conditions and actions (expressions) of the update transition in the DDStore component (described later). The two main graphical windows show the top level view of the Track Table structural IO automaton (top window) and its most important component, the DDstore (for distributed datastore) that maintains the actual data structure of the measurement results (bottom window).

The Track Table service is initiated by receiving a message through the rcv component. (Note that gray boxes indicate existing operating systems components. Only the interfaces of these are modeled since no code generation is needed for them.) The message can be either from the actual local measurement or a neighbors table of results. The format of the two messages differ, so the unpack component extracts the information and provides it in a uniform format to the DDStore component.

The DDStore component is not a single component, it is replicated N times, where N is the number of nodes in



update

guard	out_pr.counter < newentry_pr.counter newentry_pr.nodeID == index

expressions	in_pr.counter = newentry_pr.counter
	in_pr.strength = newentry_pr.strength
	in_pr.age = NEW_AGE

guard

Figure 1. Track Table middleware service for acoustic tracking

the system. (For scalability, this could be reduced to the size of the neighborhood where the active tag is actually audible.) The inside of the DDStore component in the bottom window shows the newentry dataport. It is a structure of three data values: strength stores the actual measurement, counter is a timestamp, while nodeID is the id of the node where the measurement was made. This latter acts as the index in the DDStore; this is what determines which DDStore instance the given data belongs to.

The update transition is the heart of the DDStore component. Its guard conditions and actions (expression) are partially shown in the upper right hand window. The guard conditions check whether the new entry is indeed newer than the currently stored one and whether the index is matching. If these conditions are met and the transition gets executed, the entry state will store the new data. The assignment is done field by field. The update transition also sends the new data out for packing and broadcasting it to the neighbors. Aging and broadcasting are done with the help of the clock OS component. The Display component shows some debugging information on the three LEDs of the motes.

The code generator that was written for DISSECT targeting the motes and TinyOS takes these graphical models and generates the middleware layer automatically including interfacing with the operating system. Note that the same DISSECT environment is used to generate the middleware layer of SIESTA, our own simulator for distributed control applications for structural vibration damping (http://www.isis.vanderbilt.edu/projects/nest/downloads.asp). Of course, the code generator is different from the TinyOS one. SIESTA is implemented in Java and uses a simplified IO automaton-based middleware model. Middleware services captured and automatically synthesized for those kinds of applications include message routing and broadcast.

6. Acknowledgement

The DARPA IXO NEST program provided support for the work described in this paper.

7. Conclusions

We have presented a modeling language and a supporting graphical environment for the modeling and automatic synthesis of distributed middleware services. The same language and environment is capable of generating the middleware layer for two different hardware platforms running different models of computation implemented in two different programming languages. Furthermore, only assignments, i.e. actions, and conditional expressions, i.e. guard conditions, are captured textually. Finally, the environment is intuitive and relatively easy to use.

While this clearly demonstrates the promise of this technology, a lot of research is still to be done. Middleware service composition needs better support. Verification of interface models against the actual implementation in case of black box components (e.g. existing hand-written code) or against implementation models in case of white box components, are also open problems.

References

[1] N.A. Lynch, “Distributed Algorithms”, Morgan Kaufmann Publishers, 1996.

[2] L. de Alfaro, T.A. Henzinger, “Interface Automata”, *Proceedings of the Ninth Annual Symposium on Foundations of Software Engineering (FSE)*, ACM Press, 2001, pp. 109-120.

[3] T.A. Henzinger, “Design and Verification of Embedded Systems”, *Cray Distinguished Lecture*, University of Minnesota, September 2001.

[4] T.A. Henzinger, “From Models to Code: The Missing Link in Embedded Software”, *Fifth International Conference on Hybrid Systems: Computation and Control (HSCC)*, March 2002, Stanford, California.

[5] E.A. Lee, “What’s Ahead for Embedded Software?”, *IEEE Computer*, September 2000, pp. 18-26.

[6] J.M. Kahn, R.H. Katz and K.S.J. Pister, “Next Century Challenges: Mobile Networking for Smart Dust” *ACM/IEEE Intl. Conf. on Mobile Computing and Networking (MobiCom 99)*, Seattle, WA, August 17-19, 1999.

[7] A. Ledeczi, A. Bakay, M. Maroti, P. Volgyesi, G. Nordstrom, J. Sprinkle and G. Karsai, “Composing Domain-Specific Design Environments”, *Computer*, pp. 44-51, November, 2001.

[8] Smart Dust project, “Autonomous sensing and communication in a cubic millimeter” http://robotics.eecs.berkeley.edu/∼pister/SmartDust/

[9] Ptolemy project, “Heterogeneous modeling and design” http://ptolemy.eecs.berkeley.edu/