Algorithms for Generic Role Assignment in Wireless Sensor Networks

Christian Frank
Dept. of Computer Science
ETH Zurich, Switzerland
chfrank@inf.ethz.ch

Kay Römer
Dept. of Computer Science
ETH Zurich, Switzerland
roemer@inf.ethz.ch

Abstract. We consider configuration of wireless sensor networks, where certain functions must be automatically assigned to sensor nodes, such that the properties of a sensor node (e.g., remaining energy, network neighbors) match the requirements of the assigned function. Essentially, sensor nodes take on certain *roles* in the network as a result of configuration. To help developers with such configuration tasks for a variety of applications, we propose *generic role assignment* as a programming abstraction, where roles and rules for their assignment can be easily specified using a configuration language. We present such a role specification language and distributed algorithms for role assignment according to such specifications. We evaluate our approach and show that efficient and robust generic role assignment is practically feasible for wireless sensor networks.

Categories and Subject Descriptors: C.2.1 [Computer-Communication Networks]: Network Architecture and Design; D.1.m [Programming Techniques]: Miscellaneous

General Terms: Algorithms, Languages, Performance

Keywords: Sensor Networks, Programming, Configuration

1. INTRODUCTION

Wireless sensor networks consist of so-called sensor nodes – small untethered computing devices equipped with sensors, a wireless radio, a processor, and autonomous power supply. Large and dense networks of these devices can be deployed unobtrusively in the physical environment in order to monitor a wide variety of real-world phenomena with unprecedented quality and scale while only marginally disturbing the observed physical processes [9].

Many sensor-network applications require some form of self-configuration, where sensor nodes take on specific functions in the network. Configuration of a sensor network is particularly challenging, as the anticipated large number of sensor nodes participating in a network typically precludes manual configuration of individual nodes. Additionally, pre-deployment configuration is often infeasible because some configuration parameters such as node location and network neighborhood are typically unknown prior to deployment. Also, node parameters may change over time, necessitating dynamic re-configuration.

When sensor nodes join the network, they are in an initial, homogeneous software state. However, nodes may differ in their hardware capabilities and parameters such as their location or their network neighborhood. The goal of configuration is to break the initial symmetry and assign specific *roles* to individual sensor nodes based on their properties. As the network and node properties change over time, role assignments must be updated to reflect these changes. Based on the assigned roles, sensor nodes may adapt their behavior accordingly, establish cooperation with other nodes, or may even download specific code for the selected role.

A number of research projects related to sensor networks have stated the need for such role assignment (e.g., [6, 13, 20]) as a basic service to assign functions to sensor nodes. In addition, a number of "classical" network configuration problems can also be considered instances of role assignment, for example, *coverage*, *clustering*, and *in-network data aggregation*.

Coverage. A certain area is said to be covered if every physical spot falls within the observation range of at least one sensor node. In dense networks, each physical spot may be covered by many equivalent nodes. The lifetime of the sensor network can be extended by turning off these redundant nodes and by switching them on again when previously active nodes run out of battery power [27]. Essentially, this requires proper assignment of the roles `ON` and `OFF` to sensor nodes. □

Clustering. Clustering is a common technique to improve the efficiency of data delivery (e.g., flooding, routing) [11]. With clustering, one of the three roles `CLUSTERHEAD, GATEWAY, SLAVE` is assigned to each node. A clusterhead acts as a hub for slaves in its neighborhood such that slaves directly communicate with their clusterhead only. Gateways are slaves of more than one cluster and interconnect multiple clusters by forwarding messages between them. □

In-Network Aggregation. Due to the scarcity of energy and the high energy cost of wireless communication, reducing data communication is an important design goal in sensor networks. One common form of data reduction is *in-network data aggregation*, where certain nodes in the network aggregate sensory data from many sources [8]. For this, sensor nodes must be assigned the roles `SOURCE` (generate sensory data), `AGGREGATOR` (aggregate data), and `SINK` (consume aggregated data) roles. In order to achieve a significant network traffic reduction, aggregator nodes should be located close to the data sources they aggregate. □

While a number of specialized algorithms for these problems have been developed, these are typically hard to adapt to different applications, where varying criteria for assigning the above roles may have to be applied.

Driven by these observations, our aim is the provision of specification techniques, algorithms, and tools to support *generic* role

SenSys'05, November 2–4, 2005, San Diego, California, USA.

assignment that is applicable to a wide variety of role-assignment problems. Such a framework could make the use of the above network configuration techniques more flexible. Moreover, the results of our work may be integrated as a fundamental service into programming frameworks such as [6, 13, 20].

From another point of view, generic role assignment can be considered a *programming abstraction* that partially shields application developers from the complexity of programming sensor networks at the system level. Rather than implementing low-level protocols and node functions, the developer can now specify parts of the system behavior using a high-level configuration language. Such programming abstractions have recently gained significant attention (e.g., [1, 22]) and can be interpreted as a step towards making sensor networks more accessible for users that are not experienced system-level programmers (e.g., typical application-domain experts).

We have motivated the need for generic role assignment in an earlier position paper [15], where we also sketched possible directions for its realization – leaving open the question whether generic role assignment could be actually realized in practice. In the present work, we propose and evaluate concrete instances of a configuration language, a distributed role-assignment algorithm, and a role compiler. We also present a simulation-based tool that implements all these functions, allowing for large-scale planning and evaluation of role-assignment tasks in realistic network setups. We thereby support our earlier claim that generic role assignment is practically feasible both in terms of efficiency and robustness.

The remainder of this paper is structured as follows. In Section 2 we give an overview of the various components involved in generic role assignment. In Section 3 we introduce a language for specifying roles. Sections 4 and 5 present distributed algorithms for role assignment. A prototype implementation of these algorithms is discussed in Section 6, which was also used to perform the evaluation presented in Section 7. We discuss related work in Section 8 and application-specific role-assignment implementations in Section 9 before concluding the paper with Section 10.

2. OVERVIEW

Figure 1 gives a sketch of the envisioned use case and its core elements. To setup or reconfigure the sensor-network, the user/developer provides a *role specification* that defines possible roles and rules for how to assign roles to nodes. This specification is distributed to the whole network via a gateway – alternatively it could have already been pre-installed on the nodes. On the nodes, a *property directory* provides transparent access to node properties and capabilities. A distributed *role assignment algorithm* assigns roles to sensor nodes, taking into account role specifications and node properties. Finally, applications on the node access node properties (including the node's role), which may trigger execution of role-specific code, e.g., when the node has become a clusterhead an according routing component could be enabled.

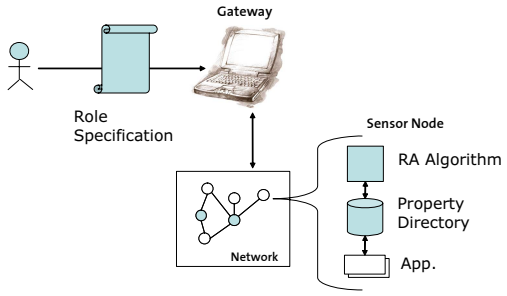


Figure 1: Core elements for generic role assignment

Property Directory. Properties of individual sensor nodes are available sensors (e.g., temperature) and their characteristics (e.g., resolution); other hardware features (e.g., memory size, processing power, communication bandwidth); remaining battery power; or physical location and orientation. Some properties are static, some may change over the lifetime of the network. However, we assume that properties are not subject to frequent significant changes. This reflects the understanding that a particular configuration is valid for a certain minimum amount of time. Depending on their nature, properties may be defined at production time, by hardware introspection, or by sensors. The property directory provides a unified interface for accessing property values. There is one such directory on each sensor node, which is independent of the directories on other nodes.

In our implementation, the property directory exports property values as a list of name-value pairs. Moreover, it can provide an asynchronous notification when a property value changes.

Sample contents of the property directory are shown on the right. The current *role* of the node, and other information acquired during the role-assignment process (e.g., topology information such as number of neighbors) are also treated as node properties. Applications can subscribe to role changes or to other properties of interest and react accordingly. Inversely, applications may update entries in the property directory, which would notify the role-assignment algorithm to adapt assigned roles accordingly.

Property	Value
battery	50%
pos	(12.3, 3.4)
temp-sensor	true
neighbors	7
role	ON

As described, the property directory is not specific to the role assignment task, but a general component facilitating cross-layer interaction among software components. For our purposes, the property directory supports numeric and Boolean types, node positions, and sets of node IDs.

Role Specification. In its basic form, a role is an identifier (e.g., `CH` for clusterhead, `GW` for gateway). The role specification is a list of role-rule pairs. For each possible role, the associated rule specifies the conditions for assigning this role. Rules are Boolean expressions that may contain predicates over the local properties of a sensor node and predicates over the properties of well-defined sets of nodes in the neighborhood of a sensor node. All nodes in the network have a copy of the same role specification. This reflects the understanding that all sensor nodes are in the same initial software state. Detailed role specification examples will be given in Section 3.

Role-Assignment Algorithm. The task of this component is to assign roles to sensor nodes, taking into account role specifications and sensor node properties. Depending on the specific problem instance, it might be useful to allow the assignment of multiple roles to one node. For example, a single sensor node might act both as a data source and as an aggregator. Property changes and node failures may necessitate re-assignment of roles.

A separate instance of the role-assignment algorithm is executing on each sensor node. Triggered by property and role changes on nodes in the neighborhood, the algorithm evaluates the rules contained in the role specification. If a rule evaluates to true, the associated role is assigned. We discuss such algorithms in Sections 4 and 5.

Role Compiler. The above algorithm can be considered a template that must be properly parameterized for a specific role assignment task. This parameterization is carried out by a compiler at the gateway, which reads a role specification and outputs appropriate parameters for the role-assignment algorithm. These parameters are

then encoded in a role specification message that is sent to all nodes in the network.

Basic Services. A number of basic services such as node localization, neighbor/topology discovery, or time synchronization may add valuable information to the property directory. The availability of such services could also be a node property. These services are decoupled from the rest of the system through the property directory.

In the next sections we focus on the role specification and the role-assignment algorithm.

3. ROLE SPECIFICATIONS

In this section we introduce the notation for *role specifications*. We first show how this approach can be used for a number of applications, later we will review the essential specification components in more detail.

3.1 Application Examples

Let us revise the examples sketched in the introduction into more formal specifications. Note that these role specifications will typically result in approximate solutions of the respective configuration problems.

Coverage. As mentioned earlier, nodes must be assigned `ON` and `OFF` roles. Requirements for the assignment of these roles are that the area of interest is covered by the sensors of `ON` nodes, and that `ON` nodes have sufficient remaining battery power. Assuming one is interested in coverage with temperature readings, one possible formulation could be:

```
1 ON :: {
2   temp-sensor == true &&
3   battery >= threshold &&
4   count(2 hops) {
5     role == ON &&
6     dist(super.pos, pos) <= sensing-range
7   } <= 1 }
8 OFF :: else
```

The rule in lines 1-7 specifies the conditions for a node to have `ON` status: it must have a temperature sensor and enough battery power (lines 2 and 3). As a third condition, we require that at most one other `ON` node should exist within this node's sensing range. This is specified by the `count` operator in line 4. It expects a hop-range as its first parameter and returns the number of nodes within this range for which the expression in curly braces evaluates to true. Here we request to evaluate nodes within 2 network hops. Note that the used property names (e.g., `role` in line 5, `pos` in line 6) in the nested expression refer to properties of the specified neighbor nodes. To access properties of the current node instead, the prefix `super` is used (e.g., `super.pos` in line 6). The `dist` operator used in line 6 returns the metric distance between two positions. In the example, it specifies that only nodes located within this node's sensing range should be counted.

In other settings it would be useful to retain state on network neighbors instead of just counting them. Clustering is such an example.

Clustering. A clustering approach needs to define assignment rules for clusterhead (`CH`), gateway (`GW`) and `SLAVE` roles. The assignment of these roles depends on a variety of parameters. Clusterheads should be more powerful devices (in terms of processing, memory, communication, and energy supply), because they act as hubs for many slaves. This may be easily formulated in terms of the property directory and is neglected here. For a role specification, consider the following basic scheme:

```
1 CH :: {
2   count(1 hop) {
3     role == CH
4   } == 0 }
5 GW :: {
6   clusterheads == retrieve(1 hop, 2) {
7     role == CH
8   } &&
9   count(2 hops) {
10    role == GW &&
11    clusterheads == super.clusterheads
12  } == 0 }
13 SLAVE :: else
```

A node that does not have any clusterhead among its neighbors declares itself clusterhead (`CH`, lines 1-4).

Nodes should be assigned the role gateway (`GW`) if they are neighbors to at least two clusterheads but are not aware of any other gateway nodes interconnecting the same two clusterheads.

To achieve this, we introduce the `retrieve` operator (line 6), which is similar to `count`, but returns a set of node identifiers instead of only counting the nodes. In this example, the retrieve operator is used to identify clusterheads in the 1-hop neighborhood of the node and to bind them to the local property `clusterheads` in line 6 (similar to binding of variables in declarative programming languages). Using the `clusterheads` property, we require in lines 9-12 that within 2 hops no other gateways should interconnect the same set of clusterheads.

The second parameter to `retrieve` in line 6 requests any *two* matching nodes. If not enough matching nodes exist, the `retrieve` expression evaluates to false. In this case, the `GW` role is not assigned, the property `clusterheads` remains undefined, and the evaluation of lines 9-12 can be omitted.

In-Network Aggregation. Similar rules can be designed for an exemplary application requiring in-network aggregation.

```
1 AGG2 :: { analogous to AGG1 }
2 AGG1 :: {
3   count(2 hops) {
4     role == SOURCE &&
5     dist(pos, sink-pos) >
6       dist(super.pos, sink-pos)
7   } >= 2 &&
8   count(2 hops) {
9     role == AGG1
10  } == 0 }
11 SOURCE :: { temp-sensor == true }
```

In this example, sensor nodes equipped with temperature sensors act as data sources (line 11). A single sink node with known position `sink-pos` consumes aggregated data. Aggregator nodes (`AGG1`) should be placed in the close neighborhood of many sources (line 4) compared to which the aggregator is closer to the sink (lines 5-6) because data flows from sources to the sink. We furthermore require that no other nodes with role `AGG1` should exist within two hops.

Note that we used a second role `AGG2` for aggregators of higher level which aggregate information from nodes with role `AGG1` instead of sources. `AGG2` nodes should be similarly placed between the `AGG1` nodes and the sink and no other `AGG2` nodes should exist in their 2-hop neighborhood.

3.2 Syntax and Semantics

Let us review the specification techniques introduced in the examples. A role specification consists of a list of *roles* and associated conditions involving the values of local properties of a sensor node or the properties of well-defined sets of nodes in the neighborhood of the node. The conditions for a role *k* are determined by an as-

sociated role predicate c^k. We assume c^k has been preprocessed by the role compiler and rearranged into its disjunctive normal form:

$$c^k = (c^k_{11} \wedge \cdots \wedge c^k_{1n_1}) \vee (c^k_{21} \wedge \cdots \wedge c^k_{2n_2}) \vee \ldots \quad (1)$$

Three types of *atomic predicates* c^k_{ij} are supported:

Simple predicates are essentially Boolean expressions formulated in terms of properties and constants, possibly involving basic arithmetic operations.

Count predicates of the form

`count(scope) { pred } rel const`

can be used to count nodes that match a nested predicate **`pred`** within a given number of hops **`scope`** around the current node and compare the result to a constant expression **`const`** using a given relation **`rel`**.

Retrieve predicates are similar, these have the form

`p == retrieve(scope,size) { pred }`

and can be used to bind the IDs of a set of nodes matching **`pred`** to a local property **`p`**. A parameter **`size`** specifies that at least **`size`** matching nodes must exist, otherwise the predicate evaluates to false. After evaluation, **`p`** contains the IDs of the matching nodes and can be used as a local property.

Within count and retrieve operators, the nested predicate **`pred`** specifies the conditions under which a remote node is counted or retrieved, respectively. These conditions are arranged in a disjunctive normal form in which, essentially, only *simple predicates* are allowed. Because the properties used in **`pred`** generally reference property values of remote nodes, it is furthermore possible to prepend `super` to property names to reference properties of the current node instead.

As mentioned earlier, the property directory supports numeric and Boolean types, node positions, sets of node IDs, and the enumeration `role`. When comparing node property values, *equality* is supported for all properties, while the usual ordering relations (such as $<$, $\leq$ etc.) are additionally available for all numeric properties.

Note that because retrieve predicates bind local properties which can be referenced by other *count* and *retrieve* statements, the former must be evaluated before predicates referencing the bound value. Moreover, the specification must not contain circular dependencies between any two retrieve statements that are part of the conditions of any role. This is checked by a compiler before sending the specification to the nodes. In Sections 4 and 5 we describe distributed algorithms that can be used to implement the semantics described above.

The presented specification language obviously cannot capture all thinkable role-assignment problems (see also Section 9 for this issue). However, from our experience it can be used to implement practical approximations of many configuration tasks. Moreover, our approach can be extended in two ways to be more powerful: Firstly, custom predicates (such as the `dist` operator mentioned earlier) can be implemented by the programmer to support complex role-assignment tasks. Secondly, applications may subscribe to certain role changes and changes of other properties. When notified of such a change, the application may perform any computations that cannot be expressed directly with the role specification language. In addition, the application may set values in the property directory, triggering the role-assignment algorithm to revise role assignments to take into account the modified properties.

Generally, the role-assignment abstraction enables the programmer to address configuration problems based on rather stable network properties (there is a limit on the frequency of changes of properties on which the configuration decisions are based). In order to perform configuration, any algorithm that implements role assignment will have to reach consensus on the current state of the network within local network neighborhoods around each node. The implementation of the abstraction is particularly optimized for *symmetric* problems where any two nodes within a local neighborhood both benefit from knowing each other's properties.

Crucial for the locality of any role-assignment algorithm is the given *scope* that is used in count and retrieve statements because it governs the degree of interdependence between nodes. We therefore define the *maximum scope* of a specification:

DEFINITION 3.1 (MAXIMUM SCOPE). *The* maximum scope *of a given specification is the highest hop-number used as a scope for count and retrieve statements.*

Similarly we introduce a term for the set of nodes that can influence the role selection of a given node:

DEFINITION 3.2 (CRITICAL AREA). *The* critical area *of a node u is the set of nodes v with*

$$0 < d(u,v) \leq \text{maximum scope}$$

where $d(u,v)$ is the hop-length of the shortest path between u and v.

4. ROLE-ASSIGNMENT ALGORITHM

In a previous position paper [15] we motivated generic role assignment and sketched a possible distributed algorithm that could be used to implement role assignment. This algorithm is based on a fixpoint iteration, where each node would repeatedly fetch the current values of all relevant remote properties in order to evaluate the role predicates, eventually deciding on a (preliminary) role for itself. These evaluation cycles would have to be properly sequentialized among neighboring nodes in order to ensure consistent role assignments. Assuming that there is a fixpoint configuration, each node would end up with a role that does not change in subsequent evaluation cycles.

While this approach works in principle, it turns out that the overhead for locking and unlocking neighbors for sequentialization is prohibitively high. Therefore, we have examined more efficient algorithms that *proactively* distribute property values to neighbors. This is based on the observation that a node can decide which of its property values are relevant to what neighbors, because each node uses the same role specification.

This proactive approach eliminates much of the overhead for "locking" nodes and concurrently introduces some redundancy that makes it more robust in the face of message loss. We present the basic algorithm below. Later in Section 5 we introduce two probabilistic initialization schemes that make it more efficient.

It must be emphasized that the algorithms are *heuristics* which may result in temporary inconsistencies due to the lack of sequentialization. We will employ randomized delays in order to avoid these inconsistencies. These temporary inconsistencies will be removed in subsequent evaluations at one of the affected nodes.

The algorithm operates on a given role specification that needs to be distributed to the nodes. Role specifications are encoded into byte-optimized messages at the sink using a syntax tree that is constructed from the given specification. We do not focus on the (reliable) distribution of the role specification in this paper. While we have implemented a flooding-based approach, several well-studied algorithms for code distribution exist [12]. In some contexts, the specification may also be loaded into the node offline before deployment.

4.1 Overview

The basic algorithm is built around local *cache tables* maintained at each node, which contain a collection of (local and remote) properties that are relevant for role assignment. Eventually, the node will refer to its cache table to assign its own role, based on the information it has learned about its neighbors up to that time. We will refer to this as *local rule evaluation* as the node does not involve any additional remote data apart from its own cache.

In the upcoming subsections we will discuss how to *initialize* the node's cache table after the specification is received, how to *propagate* node properties that have changed, how to perform the above-mentioned *local rule evaluation*, how to adapt to *changing network properties*, and finally, how to detect *termination* of the algorithm.

4.2 Initialization

The initialization of the local cache table is performed upon receiving a new role specification. As all nodes share the same specification, they will essentially setup their tables in the same way.

In a first step, the node extracts the set of relevant properties from the specification. For each property it infers from the specification the farthest distance (in hops) each property needs to be propagated at most, which we will refer to as the property's *max* value.

If a property is only used in simple predicates formulated from local properties, its *max* value is 0 and the property will never be propagated to other nodes. The node's role is also a property and must be propagated as far away as the maximum scope of all count and retrieve operators it occurs in. Note that the information on property *max* values is constant for a given specification and their maximum corresponds to the specification's *maximum scope*.

Using the above information, a given node A initializes its cache table as shown in Figure 2. To illustrate the algorithm, we use a simplified coverage specification – as depicted – where we assume that the node's sensing range is equal to its transmission range and we specify that no two neighbors are allowed to be `ON` concurrently.

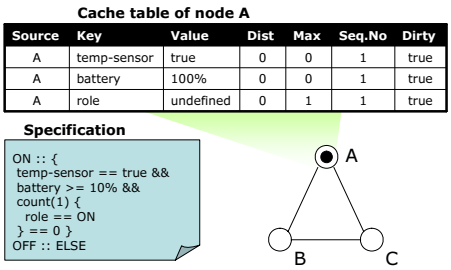


Figure 2: Node A after initialization

The table is initialized with a row for each property used in the specification. Each row contains a field denoting the source node (which is A in this case as information stems from A's local properties), the property key, and its current value. Furthermore, a *dist* field denotes the hop-distance between the source node and A. At initialization this value is always 0 because all information is local. The table also holds the *max* value described earlier to indicate how far a property must be propagated.

Each row also contains a *sequence number*, which every node maintains for its local entries. It is initialized with 1 and increased every time a local property (with *dist=0*) is updated. Finally, a *dirty* bit specifies whether this information was already propagated to neighbors (false) or not (true). Initially, this value is true.

The cache table – initialized with local properties as described here – contains redundant information that is already present in the property directory. When optimizing for memory, these entries could be generated just-in-time from the local property directory instead. Similarly, the *max* values could be stored only once for each property and *dist* values once for each node.

Once the cache table has been initialized, a node will schedule the execution of three procedures that we will describe in the following subsections (note that all random delays are uniformly distributed in the given intervals).

Property Propagation after a random delay $t_{\text{prop}} \sim (0, T_{\text{prop}})$. The delay helps aggregating newly arriving information into a single message and at the same time spreads traffic over a longer time interval. See Section 4.3 for details.

Local Rule Evaluation is scheduled after an initial delay t_{init}. This delay is chosen to allow adequate property propagation from all nodes in the current node's *critical area*. It is computed from the maximum scope *s* using an additional random offset $t_{\text{eval}} \sim (0, T_{\text{eval}})$ to reduce the chance of simultaneous role evaluations:

$$t_{\text{init}} = sT_{\text{prop}} + t_{\text{eval}}$$

Failure Detection at regular intervals $T_{\text{heartbeat}}$.

4.3 Property Propagation

To transmit properties to its neighbors, a node creates an *update* message, essentially containing a list of cache table rows.

The message is composed from all rows with *dirty* = *true* and *dist* < *max*. Entries with *dist* = *max* have reached the their maximal scope and need not be propagated any further (e.g., the local properties *battery* and *temp-sensor* in the coverage example). Essentially, the message contains a copy of these rows with the *dist* field increased by 1 and *max* and *dirty* fields left out (*dirty* is true anyway and *max* values can be derived from the specification). Furthermore, the node resets all dirty bits of its table to false. The resulting *update* message is broadcast to all neighbors. Note that property keys can be efficiently encoded as an integer index, because all nodes use the same specification.

The receivers of such *update* messages enter the contained information into their local tables. If entries of the incoming message and the local table refer to the same property of the same source node, the information with the larger sequence number is retained (note that sequence numbers are increased by the source node only, while other nodes forward them unmodified). If the information has taken a shorter path (source and sequence number are the same, but *dist* field is smaller) the *dist* field is set to the smaller value.

On the first incoming *update* message, receivers schedule

Property Propagation after a random delay $t_{\text{prop}} \sim (0, T_{\text{prop}})$ in order to forward new information with *dist* < *max*.

Local Rule Evaluation after a random delay $t_{\text{eval}} \sim (0, T_{\text{eval}})$. Generally, T_{eval} is chosen to be larger than T_{prop}, see Section 4.4 for details.

The property propagation delay t_{prop} fulfills two functions at different layers: Firstly, it is used to smooth out traffic bursts that would occur after a property change that is forwarded over multiple hops. When using a contention-based MAC layer, this would additionally reduce collisions, as transmit attempts are spread over a longer period of time. Secondly, it reduces the number of update messages by collecting many "dirty" table rows into a single message as shown in Figure 3.

Our propagation procedure adds some redundancy when information is forwarded over multiple hops because one bit of information is typically forwarded over multiple paths. Yet this redundancy adds significantly to the robustness of the algorithm in face of packet loss. We will examine robustness in Section 7.

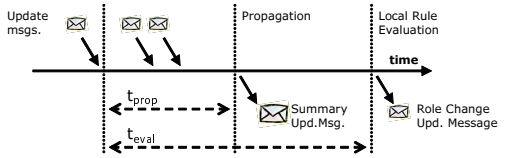


Figure 3: Property propagation and local rule evaluation

When the algorithm is used with low-duty-cycle MAC layers, one must ensure that the symmetry-breaking nature of the randomized timers is preserved. This can be achieved either by making active periods sufficiently long or by suspending algorithm timers in sleep periods (and thus stretch the algorithm execution time). What is essential is that the MAC layer does not synchronize property propagation to such an extent that it forces neighboring nodes to propagate past role changes simultaneously.

In our example from Figure 2, assume node A has recently updated its *role* property to `ON` and thus also increased the row's sequence number. In A's table, the only row with $dist < max$ and $dirty = true$ is the *role* property. A therefore broadcasts an update message to its neighbors containing its *role* property entry only. Assume that, similarly, node C has sent an *update* message, after it had picked the role `OFF`. Node B's cache table, after receiving update messages from A and C, is shown in Figure 4. In the example, no information needs to be forwarded further: All rows either have $dist = max$ or $dirty = false$.

Cache table of node B

Src	Key	Value	Dist	Max	Seq.No	Dirty
B	temp-sensor	true	0	0	1	false
B	battery	100%	0	0	1	false
B	role	*undef.*	0	1	1	false
C	role	OFF	1	1	2	true
A	role	ON	1	1	2	true

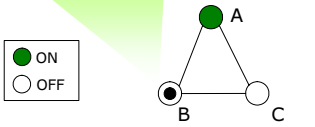


Figure 4: Sample cache table at node B

4.4 Local Rule Evaluation

As mentioned, local rule evaluation is either triggered by a new role specification, in this case delayed by t_{init}, or by the first update message following a previous local rule evaluation, in this case delayed by $t_{\text{eval}} \sim (0, T_{\text{eval}})$. The timer t_{eval} has two functions: Firstly, it helps avoid simultaneous role evaluations, for which we chose T_{eval} to be relatively large compared to T_{prop} (see Section 6 for timer settings). Secondly, it also helps avoid unnecessary transient roles by making a late but informed decision rather than performing many re-evaluations (e.g., after each incoming update message). A typical outcome where t_{eval} turns out larger than t_{prop} is shown in Figure 3.

On expiration of t_{eval} (or initially t_{init}), a node evaluates the role specification using its local cache table only. For this, it evaluates the role predicates of its specification sequentially and will assume the first role, for which the corresponding predicate matches. Each role predicate c^k is assumed to be in disjunctive normal form given in (1). Its *atomic predicates* c^k_{ij} can be evaluated sequentially.

Simple predicates are evaluated using property values from the local cache table.

Count predicates of the form

`count(`**`scope`**`) { `**`pred`**` } `**`rel const`**

consider cache table rows with $1 \leq dist \leq$ **`scope`**. The nested predicate **`pred`** is then applied to the source nodes of all these rows. The number of matching nodes is then compared to **`const`** using the given relation **`rel`**.

Retrieve predicates of the form

`p` `== retrieve(`**`scope,size`**`) { `**`pred`**` }`

are evaluated similarly. However, instead of only counting matching nodes, a set **`S`** of matching nodes is computed. Note that a retrieve statement c binds **`p`** only while evaluating its enclosing conjunction $c^k_x = (c_1 \wedge \cdots \wedge c_{n_x})$. If $|\mathbf{S}| <$ **`size`**, c^k_x evaluates to false. Otherwise, the remainder of c^k_x is evaluated for all $\mathbf{p} \subset \mathbf{S}$ with $|\mathbf{p}| =$ **`size`** until the enclosing predicate c^k_x becomes true. If no such **`p`** exists, the evaluation of c^k_x returns false.

If the node's role has changed, **property propagation** is triggered without delay.

In our example, node B performs local rule evaluation. B's cache table is shown in Figure 4. In the given example, B first checks the conditions for `ON`. Assuming that the node is equipped with a temperature sensor and a full battery, the first two conditions of the coverage specification evaluate to true. Evaluating the count statement requires counting table rows with property *role* and *dist* $= 1$. As the result of the count expression is 1, the predicate for role `ON` does not match. Node B continues by checking the conditions for `OFF`. As the "else" statement imposes no constraints, node B is assigned the role `OFF`.

4.5 Property and Network Dynamics

So far, we have described role assignment in a static network. In this section we consider the implications of changes in the network and of node configuration. For this, we distinguish three classes of changes.

The first class are *property changes*, e.g., the battery level has changed. The property directory is configured to notify the role assignment system of such changes. If a change occurred, the node updates the cache table and re-examines its chosen role via the **local rule evaluation** procedure. *Only* if the changed property value affects other (local or remote) properties by changing the evaluation result of a predicate in the specification, an update message is composed and broadcast to all neighbors. Note that the number of update messages generated through property changes is limited by the minimal interval T_{min} between two *update* messages that are induced by property changes. The programmer may also specify a tolerance interval Δ_p for a property p, such that reconfiguration is only triggered if p's value changes by more than Δ_p.

A second class of changes is when nodes *join* the network. For example, it is imaginable that additional nodes are deployed into a given area. When such a new node overhears any protocol message, it first requests a copy of the role specification from its network neighbors. After it received a reply, it uses the specification to initialize as described earlier and broadcast an update message to inform neighbors of its relevant properties.

The third class are *node failures*. To detect such failures, nodes send heartbeats every $T_{\text{heartbeat}}$. A heartbeat is essentially an *update* message, containing the key of the property that needs to be propagated farthest in the specification (e.g., the *role* property in our examples) with an empty property value. The sequence number is incremented at each heartbeat. At receiving nodes, the heartbeat message will update the sequence number field of the corresponding row in the cache table and set it to *dirty*. Forwarding nodes will thus include heartbeats in their *update* messages, provided the *maximum scope* has not been reached yet and that no other properties of the originating node where already forwarded within the node's *critical area* within the last $T_{\text{heartbeat}}$ seconds.

After sending (or suppressing) its own heartbeat, a node verifies whether all nodes contained in its cache table have sent information during the last $N_{\text{heartbeat}} \times T_{\text{heartbeat}}$ seconds. We use the factor $N_{\text{heartbeat}}$ to allow lost heartbeat messages before node failure is assumed and thus accomodate unstable links. $N_{heartbeat}$ is currently set to 3. Entries referring to nodes that have not sent any information by that time are deleted, followed by local rule evaluation and subsequent propagation if a role or property has changed.

4.6 Termination

Role assignment is continuously executed during the lifetime of a sensor network to adapt role assignments to changes in the network. Hence, role assignment itself does not terminate.

However, role assignment is typically executed in phases which start either after receiving a new role specification or on a property or neighborhood change (cf. above Section 4.5). In a role-assignment phase, the above algorithm will perform a fixpoint iteration, in which a node typically iterates through a sequence of different roles.

If a fixpoint configuration exists, each node will eventually assume a stable role. Termination refers to detecting when such a stable role has been assumed. In our approach we assume a role stable if it did not change for $T_{\text{termination}}$ seconds. Note that applications are notified of such stable roles, only. Changes of local or neighborhood properties may later trigger re-assignment of roles. If such changes occur within a role-assignment phase, these will be incorporated in the result of the current phase. However, changes must be infrequent enough to let such role-assignment phases terminate. Therefore, properties that change very fast (e.g., a light sensor) are typically not useful as input for network configuration.

In our experiments stable configurations are reached after only very few role-changes at each node (see Section 7 for details). However, there are role specifications where no such fixpoint configuration exists and no stable roles can be assumed, for example:

```
1 RED   :: { count(1) { role == GREEN } >  1 }
2 GREEN :: { count(1) { role == RED   } == 0 }
```

Here, a node requires the absence of `RED` neighbors to become `GREEN`, yet its neighbors become `RED` as a direct consequence of its own role having become `GREEN`.

We consider such specifications erroneous and provide two approaches to help a developer detect such faulty specifications. Firstly, we provide a comprehensive simulation tool to evaluate role specifications in realistic network setups which will be described in Section 7. Secondly, we use heuristics to detect potentially non-terminating specifications in a real network. If a node goes through the same cycle of roles over and over again, non-termination is assumed. Another, simpler heuristic is to assume non-termination if a node exceeds a given number of role changes without reaching a stable phase. Further analysis of the class of non-terminating specifications (as the example above) is part of current work.

5. PROBABILISTIC INITIALIZATION

In the above algorithm, all nodes start with the initial role `undefined`. In this section, we examine two approaches to initialize nodes probabilistically. The above role-assignment algorithm would then only *repair* inconsistencies of an initial configuration.

5.1 Drawing Roles

A first approach is to use a given specification together with an estimated average network density to compute a probability for the selection of each role. Assume the role specification contains specifications for roles $\{1,\dots,q\}$. We will compute a set of probabilities $\{p_1,\dots,p_q\}$ such that each node will initially assume role k with probability p_k. Note that we will use these probabilities for initializing *every* node in the network, not incorporating any additional information about a given node or its neighborhood. Thus, $\{p_1,\dots,p_q\}$ could even be pre-computed *offline* by a role compiler and disseminated to the nodes together with the role specification.

We will map the role specification to a system of q equations with q unknowns, namely the probabilities $p_1,\dots,p_q$. We will now delineate how we translate different parts of the specification into this equation system.

For this transformation, let us assume the role specification contains role predicates of the general form of Equation (1):

$$c^k = (c^k_{11} \wedge \dots \wedge c^k_{1n_1}) \vee (c^k_{21} \wedge \dots \wedge c^k_{2n_2}) \vee \dots$$

Let us assume that the values of c^k_{ij} are independent of each other. Let $P(c^k_{ij})$ denote the probability that c^k_{ij} is true. Assuming that probabilities $P(c^k_{ij})$ are known for all atomic predicates, we can derive the probability p_k of the role k:

$$p_k = P(c^k) = \sum_i \prod_j P(c^k_{ij}) \tag{2}$$

while we set the probability of a possible else role to

$$p_{\text{else}} = 1 - \sum_i p_i \tag{3}$$

We will now show how the probabilities $P(c^k_{ij})$ can be obtained for all types of *atomic predicates* c.

Let us first consider the case, where c is a *simple predicate*. If c is of the form `role==r` then $P(c) = p_r$. In all other cases, we require the programmer to explicitly specify $P(c)$. This is often possible, as we are only interested in the probabilities *at startup*. For example, the probability for the predicate `battery>10%` can be approximated with 1 at deployment time. Otherwise, an educated guess may be applied. The respective programmer-specified $P(c^k_{ij})$ are distributed along with the specification.

For *count* and *retrieve* predicates, we assume that the network density is known and that a function $E(h)$ for the expected number of nodes within h hops (the so-called h-neighborhood) can be estimated from deployment parameters[1].

Now consider a *count predicate* c of the form:

`count(`**`scope`**`) { `**`pred`**` } `**`rel lim`**

where the nested predicate **`pred`** is in disjunctive normal form and contains only *simple* sub-predicates c^k_{ij}. That is, we can compute $p_{\textbf{pred}} = P(\textbf{pred})$ according to Equation 2. Let us now consider the case where **`rel`** is "`<=`", the other cases can be solved in similar ways.

Estimating the number of nodes n to be expected within **`scope`** by $n = E(\textbf{scope})$, we can now formulate the probability that x out of n nodes match **`pred`** using the binomial distribution:

$$P(\mathbf{x} \text{ of } \mathbf{n} \text{ nodes match}) = \binom{n}{x} p^x_{\textbf{pred}} (1 - p_{\textbf{pred}})^{n-x} \tag{4}$$

Thus, the probability that less or equal **`lim`** nodes match **`pred`** (and thus the above count predicate c is true) corresponds to the sum the above probabilities for all $x \leq$ **`lim`**:

$$P(\texttt{count} = \texttt{true}) = \sum_{x=0}^{\textbf{lim}} \binom{n}{x} p^x_{\textbf{pred}} (1 - p_{\textbf{pred}})^{n-x} \tag{5}$$

[1] In our implementation, we estimated $E(h)$ for unit disk graphs and random uniform node deployment on a plane. By fitting results of simulations we found that for any fixed node density $E(h) \sim h^2$.

Finally, a *retrieve predicate c* of the form

```
p == retrieve(scope,size) { pred }
```

requires that at least **size** nodes exist within **scope** that match the given nested predicate **pred**. For calculating role probabilities, we therefore consider an equivalent count statement:

```
count(scope) { pred } >= size
```

So far, we have derived q equations with q unknowns by substituting $P(c_{ij}^k)$ in Equation (2) for $1 \leq k \leq q$. We use a fixpoint iteration to solve this equation system. Assume a set of probabilities for each role $p_k(t)$ are known at a given step t. Substituting these into the right side of equation (2), we can compute a set of new probabilities $\hat{p}_k(t+1)$. To avoid oscillations in this series, we add a memory term $p_k(t)$ that averages the old values into the newly chosen ones:

$$p_k(t+1) := \frac{1}{2}\hat{p}_k(t+1) + \frac{1}{2}p_k(t) \tag{6}$$

Note that at each step, we also normalize the $p_k(t+1)$ such that $\sum_k p_k(t+1) = 1$. We initialize this series with equal probabilities for each role $p_k(0) := 1/q$ and iterate until the series converges to a fixpoint. We would like to emphasize that this computation is done *offline* by the role compiler as all information needed is the specification and the mentioned estimation of the neighborhood size $E(h)$. The resulting probabilities are then flooded along with the specification upon which each node draws role k with probability p_k and then starts initialization as described in Section 4.2.

We will show in Section 7 how the probabilistic initialization can provide significant improvements over the baseline algorithm where all nodes start with the role `undefined`.

5.2 One Wave

In the above approach we approximate retrieve statements with count operators ignoring the fact that retrieved node IDs affect other predicates. Hence, sub-predicates are not independent of each other (as we assumed). Furthermore, other nodes depend on retrieved node IDs. The initialization of retrieved node IDs cannot be performed probabilistically (at least it is highly improbable that the initialization is done correctly), and therefore additional interaction to adequately initialize the local properties is required.

In this regard, partial (deterministic) information is helpful and can be used to properly initialize the bound properties, while other decisions can still be performed probabilistically. In this section we describe how we can use conditional probabilities to improve the stability of probabilistic decisions.

The basic idea for this approach is to leverage an existing network flood (e.g., for delivering the role specification) and execute the algorithm of Section 4 while forwarding this "propagation wave".

We generate such a propagation wave by scheduling the property propagation procedure of Section 4.3 differently: The sink is the only node that initially sends an *update* message (other nodes refrain from sending *update* messages before they received one). A node that received an update message, awaits a random propagation delay $t_{\text{waveprop}} \sim (0, T_{\text{waveprop}})$, then choses its role (see below) and includes its own chosen role into the forwarded *update* message. Note that all information of these update messages can be piggybacked onto flooded role specification messages.

Let us consider the evaluation of the count predicate used as an example in the previous Section 5.1:

```
count(scope) { pred } <= lim
```

Figure 5 depicts the situation, where the propagation wave reaches a given node A. Let us assume that A expects n nodes in **scope** and m of these are behind the wave front (to the left of A). These have already chosen their roles and included all relevant information (including the selected role) in previous update messages. That is, A has received update messages from all nodes behind the wave (on the left). A will now decide probabilistically what it expects from the $n-m$ nodes on the right.

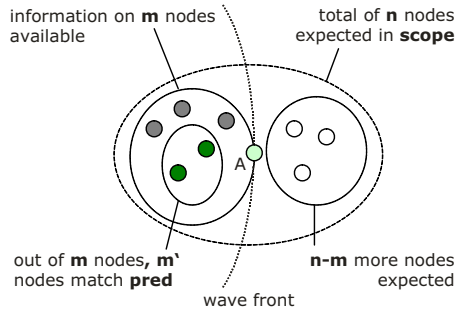


Figure 5: Propagation wave

For this, let us assume that out of the m known nodes on the left, m' match the nested predicate **pred**. We can now reformulate the probability that the count predicate is satisfied as the probability that **lim**$-m'$ "more" nodes match (out of the $n-m$ expected nodes on the right). The conditional probability that the count predicate c is true can be expressed as follows:

$$P'(c) = \sum_{x=0}^{\mathbf{lim}-\mathbf{m}'} \binom{\mathbf{n}-\mathbf{m}}{x} p_{\mathbf{pred}}^x (1-p_{\mathbf{pred}})^{\mathbf{n}-\mathbf{m}-x} \tag{7}$$

Note that count statements using other relations (greater or equal to a constant) can be treated analogously, and also the aspect that retrieve statements require a minimum number of matching nodes.

When evaluating a retrieve predicate, the node additionally binds the local properties to a set of nodes that are known to match the nested predicate, if such a set of the required size exists. If such a set exists, we consistently reflect the dependency between the local node and remote nodes (in our case the remote nodes behind/left of the wave).

For sub-predicates c in **pred** of the form `role==r` we still assign the corresponding pre-computed probability $P(c) = p_r$ of Section 5.1 as no additional information is available on the $n-m$ remote nodes that are in front of the wave.

This way, we derive q equations for the conditional probabilities p'_k, one for each role similar to Equation (2), but formulated in terms of the previously computed p_k.

$$p'_k = P'(c^k) = \sum_i \prod_j P'(c_{ij}^k) \tag{8}$$

What we changed is the way we obtain probabilities P' for the atomic predicates c_{ij}^k of the equation. On the nodes, we compute the values of p'_k in one step (without the need for fixpoint iteration) using the $\{p_k\}$ sent along with the specification and chose role k with the respective probability p'_k.

The advantage of the above algorithm is that role assignment can be performed almost entirely within *one* network-wide flood and that only few role assignments have to be "fixed" later on (using the baseline algorithm). Moreover, this approach is able to capture interdependencies between atomic predicates better. We will analyze the performance of this algorithm in Section 7.

6. IMPLEMENTATION

Generic role assignment is implemented within a simulation tool that enables the programmer to test various specifications in different kinds of network environments. It consists of a compiler for role specifications, a network setup and configuration component, a visualization tool for introspecting algorithm execution and results, and finally a discrete event simulator back-end that executes the distributed algorithms in a network environment using additive interference models and realistic wireless parameters.

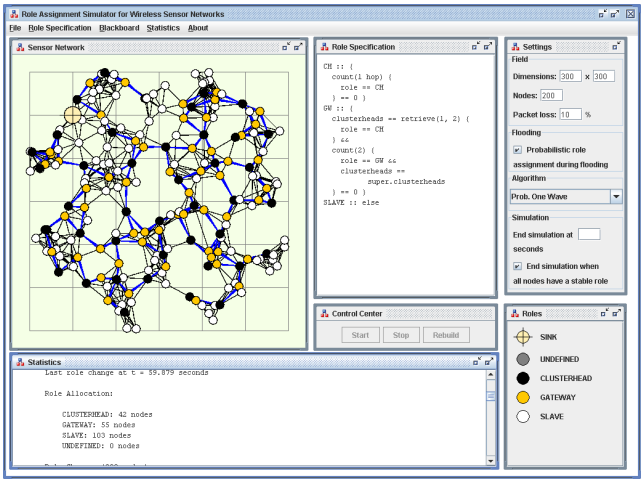


Figure 6: Role-assignment simulation tool

The user interface is shown in Figure 6. It enables the user to set up a network topology, define property directory contents (i.e., the actual node parameters) and select the algorithms for execution on the nodes. The role compiler then translates a given specification – in this case the clustering specification of Section 3.1 – into the corresponding syntax tree, performs necessary pre-processing and context checking (i.e., ensures that there are no circular dependencies and rearranges the role predicates into disjunctive normal form) and computes a number of additional parameters (such as role probabilities). Syntax tree and parameters are then encoded into a role specification message, which is then interpreted on the simulated nodes. Finally, the user can trigger delivery of the role specification message through a flood initiated at the sink, upon which the network nodes initialize and execute the role-assignment algorithm.

The visualization tool enables qualitative assessment of the algorithm execution and of role-assignment results. Figure 6 shows exemplary results when using the *one-wave* variant of the cache table algorithm to implement the clustering specification introduced in Section 3.1. The highlighted edges are drawn between a gateway and the clusterheads it interconnects. Note that in the shown simulation, apart from messages lost due to collisions, an additional per-packet loss of 10% was enabled, which might have caused some – but only few – inconsistent role assignments. We will discuss quantitative results in Section 7.

We use the discrete event simulator JiST/SWANS [3] as a simulation back-end. We adopted wireless parameters from the CC1000 radio [5] that is in use on the BTnode [29] platform, on Mica motes and on many other platforms, please consider Figure 7(a) for details. The physical layer supports additive interference and two-ray fading. Using these parameters we obtain a maximal transmission range of about 33m.

We use a variant of the CSMA MAC described in [26] with timers and delays adapted for 38.4 kbit/s. Only the broadcast service is used (i.e., no channel reservations are performed). The MAC

Parameter	Value	
bandwidth	38.4	kbit/s
transmit power	5	dBm
sensitivity	−96	dBm
rcv. threshold	−84	dBm
interference limit	−96	dBm
frequency	868	Mhz
antenna gain	0	dB
node height	5	cm
max. xmit range	33.5	m

(a) Wireless parameters

Parameter	Value
T_{prop}	3s
T_{eval}	10s
$T_{\text{heartbeat}}$	60s
$N_{\text{heartbeat}}$	3
$T_{\text{termination}}$	60s
T_{waveprop}	5s
T_{min}	1s

(b) Algorithm timers

Figure 7: Simulation parameters

does not perform collision detection, or any other means to improve robustness. We deliberately chose such a simplistic MAC to study the robustness of our algorithm separately from "tricks" performed by more advanced MAC layers. The obtained robustness results can be considered "worst case" and better values can be expected when using more sophisticated MAC layers (e.g., [14, 21, 28]).

For further analysis we have developed a second "back-end" for role assignment which derives an integer program (IP) formulation from a role specification and a given network topology. The IP can provide insights into whether a specification is infeasible, that is, whether a valid assignment of roles to nodes exists at all. Furthermore, it helps assess the quality of distributed heuristics (part of current work) that aim at assigning certain roles "optimally" (e.g., in the coverage example chose fewer ON nodes but still cover the respective area).

7. EVALUATION

An important requirement on programming abstractions is that the induced overhead should be proportional to the complexity of the specified problem. To gain an insight into whether our algorithms are adaptive in the above sense, we evaluated three specifications of increasing difficulty: The *coverage* example that we used to illustrate the *caching* algorithm in Section 4 can be considered a baseline case. We will later use this example to measure the effect of increasing the *scope* of the count statement on algorithm overhead in the corresponding Section *Scalability* below. We furthermore examine *aggregation* and *clustering* as described in Section 3.1. These both require propagation of additional properties over two hops. The *clustering* specification is especially challenging through use of the *retrieve* operator that makes role-assignment decisions dependent on the *identities* of nodes in the neighborhood.

We will evaluate all three presented algorithms, the baseline cache table algorithm from Section 4 and its combination with the two probabilistic initialization schemes described in Sections 5.1 and 5.2, to which we will refer as *cachetable*, *probabilistic* and *wave*, respectively. We set algorithm parameters according to Figure 7(b). To evaluate the protocols, we let the three algorithms assign roles to a previously uninitialized network and stop the simulation if all nodes were stable for $T_{\text{termination}}$ seconds (each node decides termination locally).

We performed experiments on connected (w.h.p.) random topologies. If there is no multi-hop path from the sink to a node, this node simply does not participate in the experiment. In our evaluations we place a variable number of nodes in a 300x300m square field.

For measuring *overhead*, we consider the total number of messages that are sent by each node and – as messages can be of variable size – the total payload sent by each node in Section 7. Note that the number of messages includes (one) specification message per node, which is the initial flood.

To quantify *correctness* of the outcome, we consider the (theoretical) network topology where two nodes have a direct network link if their distance is less than the maximum transmission range of 33.5m. An assigned role is considered correct if the respective role predicate matches for the set of neighbors obtained from the above theoretical topology.

Later on we will consider the *number of role changes* that occur after initialization to study *convergence* of our algorithms . For the *probabilistic* and *wave* algorithms, we do not count the initial role change that is induced by the probabilistic decision, as we are interested in the remaining inconsistency that has to be repaired. For each data point, we indicate 95% confidence levels obtained from repeated simulations on independently drawn random topologies.

Overhead. We study the communication effort spent by the three presented algorithms. We vary the number of nodes in the confined area to see how increasing node density affects the performance of our algorithms. The average number of messages sent per node using each specification are shown in Figure 8.

The results of Figure 8(a) show that the simplified *coverage* specification can be implemented effortlessly by all three algorithms. The maximum of *three* messages includes the specification flood, the later propagation wave for the *wave* algorithm (we implemented propagation and role specification waves separately), and finally at most one more message that is used to check whether repairs are needed.

In the aggregation example of Figure 8(b), the *probabilistic* and *wave* variants outperform the baseline algorithm. Note that the *wave* algorithm does not perform better than *probabilistic*. This is due to the *aggregation* specification (cf. Section 3.1): The *wave* algorithm can improve performance only if it can exploit knowledge about nodes *behind* the wave front. For the first count statement of role `AGG1`, this would require that `SOURCE` nodes are behind the wave front. However, the specification further requires that `SOURCE` nodes are farther away from the sink than aggregators. This is unlikely to happen, because the wave propagates from the sink outwards. Therefore, no aggregator roles are assigned during the first wave.

For clustering in Figure 8(c), *probabilistic* performs slightly worse than *caching*. This can be explained by the fact that the choice of roles here depends very strongly on the *identities* of nodes in the neighborhood and not only on their roles. Hence it is very unlikely that a probabilistic decision "guesses" the right node identities, this has been one motivation for the design of the *wave* variant, which exhibits better performance. Nevertheless, it is notable that *probabilistic* still performs comparably to *caching*.

The relative performance of the algorithms with respect to payload size per node is similar. With the *wave* algorithm, the total payload size per node in the clustering example is 179 bytes for 200 nodes and 388 bytes for 600 nodes, resulting in avg. message sizes of 26 to 39 bytes, respectively. Results are similar in the *aggregation* case, while *coverage* only requires transmitting at most 40 bytes per node as a maximum over all algorithms and node densities. Note that the maximal node degree we use is quite high: 600 nodes in the given area yield an average degree of around 20.

Correctness. We show the results when measuring the ratio of nodes with an *incorrect* role assignment to the total number of nodes in the network in Figure 9. The baseline *coverage* case does not exhibit any significant incorrectness. This is due to the simplicity of the specification, where the count operator considers the 1-hop neighborhood only. Essentially, all nodes send one message to announce their role. If inconsistencies occur, these are repaired with the second message.

For *aggregation* and *clustering* specifications, the *caching* and *wave* algorithms perform best, while the *probabilistic* variant suffers from its deficiencies when used with retrieve operators. Note that even though we omitted any means for reliable message transfer, the wave algorithm achieves very low incorrectness numbers.

Note that shown incorrectness is due to unreliable message delivery, only. If message delivery were reliable, improbable yet possible simultaneous role evaluations would not contribute to incorrectness, as these would be repaired in subsequent algorithm iterations. Nodes that end up with incorrect roles have not learned from each other's properties properly (i.e., at least one message must have been lost). A reliable MAC layer would theoretically incur zero *incorrectness* but (possibly) worse *convergence* results.

Robustness. To examine robustness in the face of message loss, we introduced an additional packet-loss probability. We measured again the ratio of incorrect role assignments to the total number of nodes for our three examples. Results are shown in Figure 10. The x-axis denotes the probability used for dropping a message.

On this scale, the algorithms do not exhibit significant differences. Note that clustering is less robust than *coverage* and *aggregation*. This is due to the fact that the dependency of the predicates on neighbor nodes is much stronger in clustering as explained in *overhead* considerations above: If, caused by a lost message from a clusterhead neighbor, a node inconsistently becomes clusterhead, many neighbors of the two will (incorrectly) become gateways, which they will have to correct later, through additional communication effort. Therefore, errors tend to amplify. Nevertheless, with all three specifications, high packet loss still yields acceptable correctness levels.

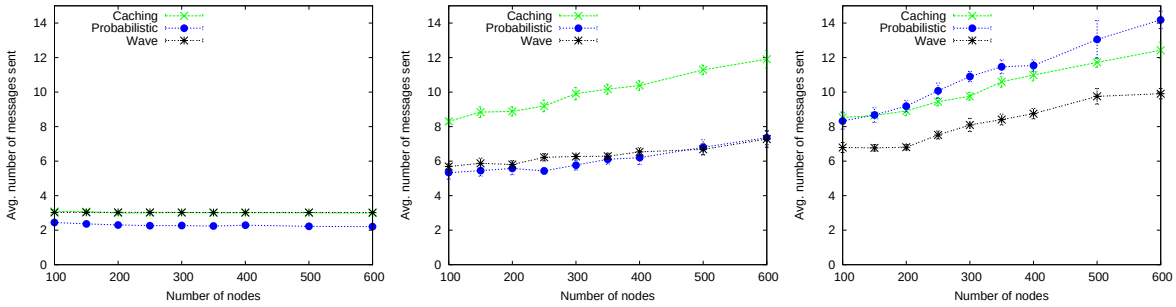
Note that message loss causes a dynamic neighborhood relation. Hence, the results depicted in Figure 10 can be interpreted as how well the algorithms can deal with dynamic neighborhoods. We plan further evaluation using bursts of lost messages that affect the perceived neighborhood more severely.

Convergence. In Figure 11, we quantify the number of role changes required after initialization until a node assumes a stable role. It shows the total number of role changes per node *after initialization*. Note that the initial role change for *probability* and *wave* is not counted here as we were interested in the repairs needed after initialization and whether initialization could improve convergence. The avg. number of role changes per node is less than 2 for all three specifications. That is, all communication effort is invested into property propagation, and almost no "undesired" repair iterations occur.

Note that the values shown for the *wave* algorithm are very low. After the initial wave, only few repairs are needed. The number of repairs needed in the *coverage* case is effectively zero, while with other specifications, at most one role change for every two nodes is required.

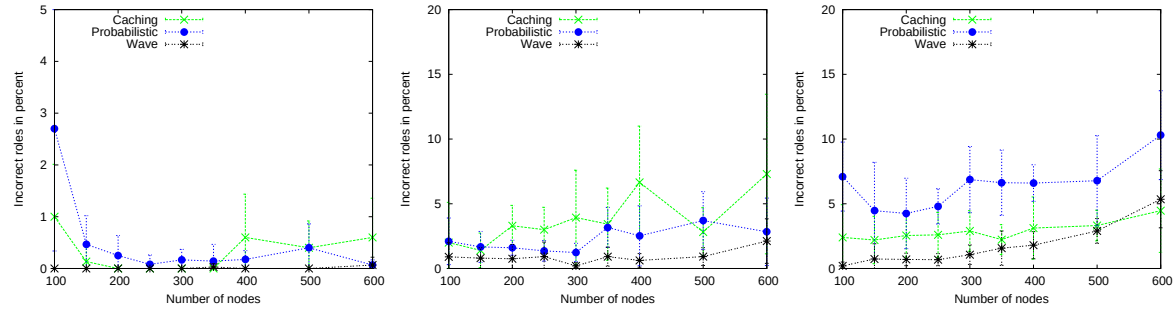
Scalability. A parameter that is crucial for the required algorithm overhead is the *maximum scope* that the programmer specified. To assess how the maximum scope affects overhead, we measured data exchanges for the simplified *coverage* example when varying the scope of its count operator from 1 to 4 in a setting with a constant number of 200 nodes.

The results depicted in Figure 12 demonstrate that information is indeed combined into few update messages. With an average node degree of about 7 and a maximum scope of 4 hops, the employed number of messages still remains low. However, the total payload increases, e.g., from 21 bytes (for scope 1) to 400 bytes per node (scope 4) for the best performing *probabilistic* algorithm. The average message size is at most 48 bytes with a scope of 4.



(a) Coverage (b) Aggregation (c) Clustering

Figure 8: Sent messages per node with increasing density



(a) Coverage (b) Aggregation (c) Clustering

Figure 9: Percentage of incorrect assignments from total nodes with increasing density

Summing up, the evaluation results indicate that the role initializations of *probabilistic* and *wave* can provide improvements over the baseline algorithm. However, we have also seen that there is no single best algorithm, although *wave* outperforms the other algorithms in most settings. With aggregation we found a case, where the *wave* algorithm is "tricked" by the specification. Apart from such special cases, a rule of thumb is that *wave* performs better than *probabilistic* for specifications that make use of `retrieve`. An interesting direction for future work would be the development of further heuristics to automatically derive from a given specification which algorithm can be expected to give the best results.

8. RELATED WORK

Self-configuration in ad hoc and sensor networks has been an active research topic in the recent past. Various other approaches for solving *specific* self-configuration problems have been devised. Examples include coverage [16]; aggregator placement [4]; clustering, routing and addressing [10, 17, 18]. [10] uses a fixed set of roles to build a network-wide backbone infrastructure. However, none of these approaches are *generic* frameworks that support the assignment of user-defined roles in an application-specific manner.

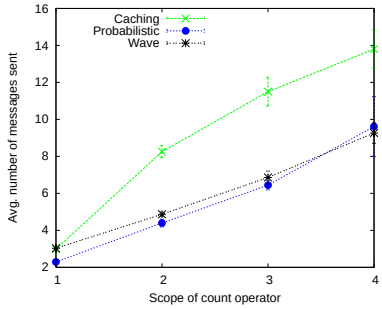


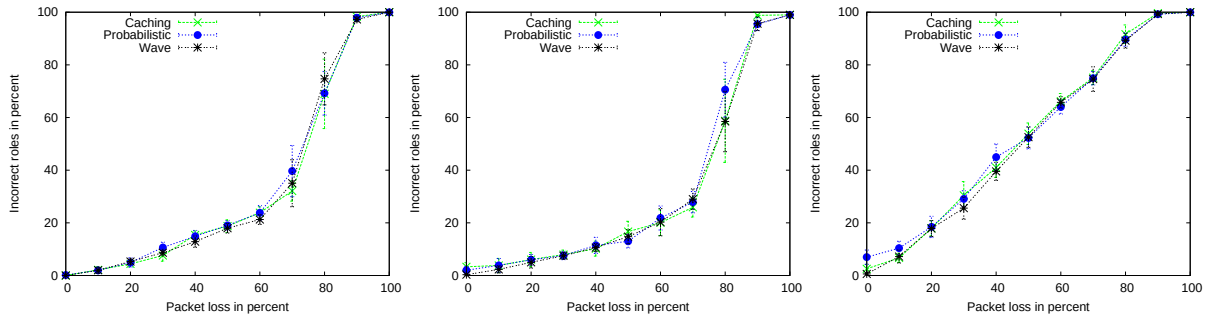
Figure 12: Coverage application with increasing scope

The concept of role assignment has been mentioned in various research projects related to wireless sensor networks. In [6], a middleware called MiLAN is outlined that controls the allocation of functions to sensor nodes in order to meet certain quality-of-service requirements specified by the user. In [13], a cross-layer framework called TinyCubus is presented that uses the notion of roles to implement efficient code deployment. In [20], a high-level programming approach for sensor networks is presented, where a high-level task specification is compiled into a set of node-level programs that must be properly allocated to sensor nodes taking into account the node capabilities.

Recently, neighborhood programming abstractions [22, 24] have been proposed, where network neighbors can easily share variables among each other. These abstractions pose an interesting opportunity for implementing our role assignment approach. In particular, each node in the network could set up a "sharing region" that equals its *critical area* in order to exchange property values among nodes.

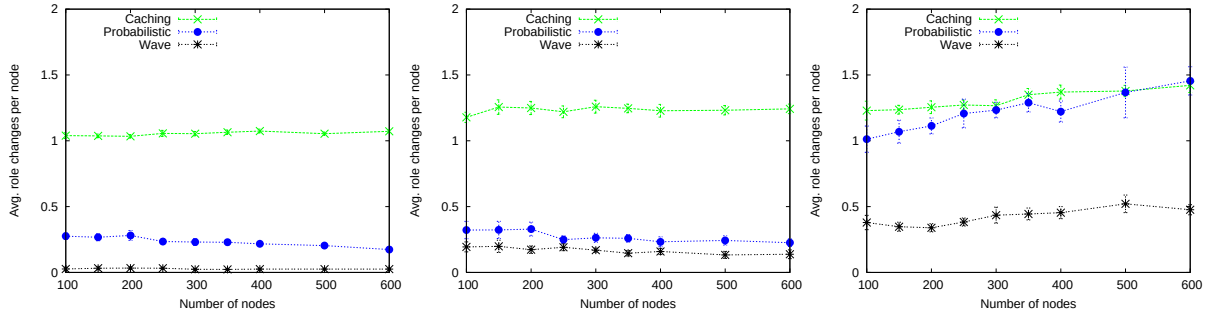
Inspired by cellular cooperation in biological organisms, Amorphous Computing [2] explores ways to program smart matter – very densely deployed collections of indistinguishable smart particles. In contrast, our approach is based on the observation that sensor nodes may significantly differ in their properties, may rely on a number of basic services (e.g., localization), and are less densely deployed. Also, we focus on the configuration of sensor networks, the actual "programming" (i.e., distributed data processing etc.) is not part of our work, although roles and other property values derived during role assignment may provide valuable input.

Our scheme for role assignment is somewhat similar to cellular automata [25], where the state of a particle in a regular arrangement is completely defined by the previous values of a neighborhood of particles around it. Note that a classification of a subclass of cellular automata in [25] indicates that a large group of automata converges to well-defined states. Major differences of our approach are that state updates are not synchronous, sensor nodes are not in a regular arrangement, and sensor nodes differ in their properties.



(a) Coverage (b) Aggregation (c) Clustering

Figure 10: Robustness in face of dropped packets



(a) Coverage (b) Aggregation (c) Clustering

Figure 11: Total role changes per node with increasing density

9. QUALITATIVE COMPARISON

Throughout the paper, we used specific instances of coverage, clustering, and aggregation problems to illustrate and evaluate generic role assignment. In the literature, numerous specialized algorithms for solving various instances of these problems have been proposed. In this section we want to convey an idea of how these algorithms compare to our implementations that are based on generic role assignment. A more extensive, quantitative comparison is part of future work.

For each of the problems, we selected a representative instance and algorithm from literature and discuss differences to our implementation with respect to semantics and efficiency.

Coverage. A common semantic for coverage is that all of a node's sensing range should be covered by other sensors before the node can turn off [19]. In our version of coverage we approximate this semantic by turning a node off if there are at least N (we use $N = 1$ in the example) other on nodes within sensing range. For a more direct implementation of the semantics in [19], a node would first `retrieve` all neighbors with overlapping sensing ranges (i.e., which are at most twice the sensing range away) and then use a custom predicate to decide whether all of the node's sensing range is covered by these neighbors. This specification would require the role-assignment algorithm to first propagate *position* and *role* properties across the required number of hops and then perform local rule evaluation after random delays. Chosen roles would be immediately propagated to affected neighbors. Our algorithm would (quite closely) match the implementation of [19] in the variant where sensing ranges are assumed equal at all nodes. □

Clustering. Our version of clustering was inspired by (but is not identical to) passive clustering (PC) [11]. With PC, a node declares itself clusterhead if none of its neighbors have done so before. A node becomes a gateway if $\alpha CH + \beta > GW$ where CH and GW are the numbers of clusterhead and gateway neighbors, resepctively, and $\alpha, \beta > 0$ are algorithm parameters that control the desired number of gateways in the system. This approach could be directly implemented by generic role assignment with an overhead comparable to the implementation in [11], where only the `role` property is shared among neighbors.

With PC, protocol information is piggybacked on existing network traffic. In contrast, generic role assignment uses a separate protocol, resulting in some traffic overhead when compared to PC. An interesting direction for future work would be *passive* generic role assignment, where protocol information would be piggybacked as with PC. □

Aggregation. A common optimization criterion for aggregator placement is to minimize the total network traffic, which is identical to solving the Steiner-tree problem. Various heuristics are used in the literature to approximate this NP-hard problem. One commonly used heuristic is called *center at nearest source* [7], where among a number of neighboring source nodes, the one closest to the sink is selected to act as an aggregator. This approach can be implemented with generic role assignment, where a node would use `count` to decide if there is another source in the neighborhood that is closer to the sink. The scope of `count` (i.e., size of the considered neighborhood) would be tuned to achieve a reasonable tradeoff between algorithm overhead and optimality of aggregator selection.

If geographic positions are used, the implementation of *closer* is straightforward by propagating the position property of a node within the scope. In the original algorithm in [7], this is done by a network-wide flood. If hop-distance to the sink is used instead of geographic positions, an additional role specification would be needed to obtain the hop-distance of each node from the sink. For this, each node would have a property `dist` that specifies its distance from the sink. A node would then `retrieve` among its neigbors the node with the smallest `dist` value and set its `dist` property to the retrieved value plus one. This would require a

simple extension of the `retrieve` operator to sort the retrieved nodes according to some user-specified criterion (e.g., minimum of a property value). Note that other, custom-implemented, components that determine the `dist` property could be integrated with role-assignment via the property directory. □

Apart from the above specific examples, we can make some more general observations about the efficiency of generic role assignment when compared to specialized algorithms. With generic role assignment, the value of a property is *always* propagated to *all* nodes in the scope of this property. In some cases, however, it may not be necessary to propagate certain property values. In other cases, certain nodes may not need to propagate a property at all or only to certain nodes within a scope. As part of future work we will investigate whether such optimizations can also be supported by generic role assignment.

10. CONCLUSION AND OUTLOOK

In this paper we have investigated a novel programming abstraction called generic role assignment, which allows the automatic assignment of roles to sensor nodes based on properties of sensor nodes and their respective network neighborhood. We have presented a distributed algorithm for role assignment and two variations that perform probabilistic initialization. Through an extensive quantitative evaluation we have shown that role assignment is not only a powerful tool, but can also be implemented in an efficient and robust way.

Generic role assignment can be regarded as a versatile tool for sensor-network configuration. Previous work on (self-)configuration mainly focused on rather specific problems – generic frameworks were missing so far.

A particularly noteworthy application of our approach is rapid prototyping for sensor networks. Currently, the deployment of sensor networks often involves a trial-and-error phase, where algorithms and protocols are tested in different configurations in real-world settings (e.g., [23]). With generic role assignment, these different configurations could be easily generated and changed. While all components under test could be loaded onto the nodes before deployment, these could be started and stopped through the assignment of certain roles and initialize their respective configuration parameters from the property directory.

Currently, we are implementing the proposed approach on the BTnode platform [29]. Future work includes the application of role assignment to more heterogeneous types of sensor networks which are interconnected with various kinds of smart appliances. We also plan to increase the expressiveness of generic role assignment by introducing more powerful data types (sets, enumerations, etc.), by providing more general ways to specify scopes of count and retrieve, and by supporting the assignment of multiple roles to a single node.

11. ACKNOWLEDGMENTS

We gratefully acknowledge the work of our students Oliver Bieri and André Bayer on algorithm design and implementation. We would like to thank our shepherd and our anonymous reviewers, and as well Holger Karl and Pedro José Marron for valuable comments and suggestions on draft versions of this paper.

The work presented in this paper was supported (in part) by the National Competence Center in Research on Mobile Information and Communication Systems (NCCR-MICS), a center supported by the Swiss National Science Foundation under grant number 5005-67322.

12. REFERENCES

[1] T. Abdelzaher et al. EnviroTrack: Towards an Environmental Computing Paradigm for Distributed Sensor Networks. In *ICDCS'04*, Tokyo, Japan, March 2004.

[2] H. Abelson et al. Amorphous Computing. *Communications of the ACM*, 43(5):74–82, May 2000.

[3] R. Barr. *An efficient, unifying approach to simulation using virtual machines*. PhD thesis, Cornell University, May 2004.

[4] B. Bonfils and P. Bonnet. Adaptive and decentralized operator placement for in-network query processing. In *IPSN'04*, Berkeley, USA, April 2003.

[5] SmartRF CC1000 Datasheet (rev. 2.2). Chipcon AS, April 2004. `www.chipcon.com/files/CC1000_Data_Sheet_2_2.pdf`.

[6] W. B. Heinzelman, A. L. Murphy, H. S. Carvalho, and M. A. Perillo. Middleware to Support Sensor Network Applications. *IEEE Network*, 18(1):6–14, Jan-Feb 2004.

[7] C. Intanagonwiwat, D. Estrin, R. Govindan, and J. Heidemann. Impact of network density on data aggregation in wireless sensor networks. In *ICDCS'02*, Vienna, Austria, July 2002.

[8] C. Intanagonwiwat, R. Govindan, and D. Estrin. Directed Diffusion: A Scalable and Robust Communication Paradigm for Sensor Networks. In *MobiCom'00*, Boston, USA, August 2000.

[9] H. Karl and A. Willig. *Protocols and Architectures for Wireless Sensor Networks*. John Wiley & Sons, May 2005.

[10] M. Kochhal, L. Schwiebert, and S. Gupta. Role-based hierarchical self organization for wireless ad hoc sensor networks. In *WSNA'03*, San Diego, USA, September 2003.

[11] T. J. Kwon and M. Gerla. Efficient Flooding with Passive Clustering (PC) in Ad Hoc Networks. *Computer Communication Review*, 32(1):44–56, January 2002.

[12] P. Levis, N. Patel, D. Culler, and S. Shenker. Trickle: A self-regulating algorithm for code propagation and maintenance in wireless sensor networks. In *NSDI'04*, San Francisco, USA, March 2004.

[13] P. J. Marrón, A. Lachenmann, D. Minder, J. Hähner, R. Sauter, and K. Rothermel. Tinycubus: A flexible and adaptive framework for sensor networks. In *EWSN'05*, Istanbul, Turkey, January 2005.

[14] J. Polastre, J. Hill, and D. Culler. Versatile low power media access for wireless sensor networks. In *SenSys'04*, pages 95–107, Baltimore, USA, November 2004.

[15] K. Römer, C. Frank, P. J. Marrón, and C. Becker. Generic role assignment for wireless sensor networks. In *11th ACM SIGOPS European Workshop*, pages 7–12, Leuven, Belgium, September 2004.

[16] S. Slijepcevic and M. Potkonjak. Power efficient organization of wireless sensor networks. In *ICC'01*, Helsinki, Finland, June 2001.

[17] K. Sohrabi, V. Ailawadhi, J. Gao, and G. Pottie. Protocols for Self Organization of a Wireless Sensor Network. *Personal Communication Magazine*, 7:16–27, 2000.

[18] L. Subramanian and R. H.Katz. An architecture for building self-configurable systems. In *MobiHoc'00*, Boston, USA, August 2000.

[19] D. Tian and N. D. Georganas. A node scheduling scheme for energy conservation in large wireless sensor networks. *Wireless Communications and Mobile Computing*, 3(2):271–290, 2003.

[20] A. Ulbrich, T. Weis, G. Mühl, and K. Geihs. Application Development for Actuator and Sensor Networks. In *GI Workshop on Sensor Networks*, ETH Zurich, Switzerland, March 2005.

[21] T. van Dam and K. Langendoen. An adaptive energy-efficient mac protocol for wireless sensor networks. In *SenSys'03*, pages 171–180, Los Angeles, USA, November 2003.

[22] M. Welsh and G. Mainland. Programming sensor networks using abstract regions. In *NSDI'04*, San Francisco, USA, March 2004.

[23] G. Werner-Allen, J. Johnson, M. Ruiz, J. Lees, and M. Welsh. Monitoring volcanic eruptions with a wireless sensor network. In *EWSN'05*, Istanbul, Turkey, January 2005.

[24] K. Whitehouse, C. Sharp, E. Brewer, and D. Culler. Hood: A neighborhood abstraction for sensor networks. In *MobiSys'04*, Boston, USA, June 2004.

[25] S. Wolfram. *Cellular Automata and Complexity*. Addison-Wesley, 1994.

[26] A. Woo and D. E. Culler. A transmission control scheme for media access in sensor networks. In *MobiCom'01*, Rome, Italy, July 2001.

[27] Y. Xu, J. Heidemann, and D. Estrin. Geography-Informed Energy Conservation for Ad-Hoc Routing. In *MobiCom'01*, Rome, Italy, July 2001.

[28] W. Ye, J. Heidemann, and D. Estrin. An energy-efficient mac protocol for wireless sensor networks. In *INFOCOM'02*, New York, USA, June 2002.

[29] BTnodes. `www.btnode.ethz.ch`.