Solving Generic Role Assignment Exactly *

Christian Frank, Kay Römer
Department of Computer Science
ETH Zurich, Switzerland
{chfrank,roemer}@inf.ethz.ch

Abstract

Generic role assignment is a programming abstraction that supports the assignment of user-defined roles *to sensor nodes such that certain conditions are met. Many common network configuration problems such as coverage (assign roles ON and OFF to sensor nodes such that ON nodes cover a physical area with their sensors), clustering, or in-network data aggregation can be formulated as role assignment problems. Building on our previous work in this area, we propose an extended role specification language that supports the minimization or maximization of the use of a given role. Moreover, we provide a mapping of this language to integer linear programs and implement this mapping. We show how the resulting tool can be used analyze aspects of role specifications such as feasibility and optimality.*

1. Introduction

In previous work [5, 14] we have introduced *generic role assignment* as a programming abstraction for wireless sensor networks to support the developer with a variety of different network configuration problems that commonly arise in sensor network applications. Essentially, generic role assignment allows the definition of a set of *roles* that should be assigned to sensor nodes such that a given set of *rules* is satisfied. These rules can refer to properties (e.g., location, battery level) of a sensor node and to properties of nodes in a neighborhood.

For illustration of this concept, consider the *coverage problem* [18], where two roles `ON` and `OFF` are defined that should be assigned to sensor nodes such that each physical spot is within the sensing range of at least one sensor node with role `ON`. The rationale behind this is that nodes with role `OFF` do not contribute to sensing coverage and can thus be switched to a power-saving sleep mode. If a node with role `ON` fails (e.g., due to depleted batteries), roles must be re-assigned to ensure continuous coverage. Other role assignment problems are clustering (using the three roles clusterhead, gateway, and slave) and in-network data aggregation (using roles source, aggregator, and sink).

In [5] we introduced a language to specify such role assignment problems, proposed parameterizable distributed algorithms to find role assignments that match a given specification, and provided a mapping of language specifications to algorithm parameters. Through experiments, we could show that the proposed system performs well for a number of different role assignment problems.

However, we did also mention a number of open issues with this work. The first of these issues is *termination*. The distributed algorithms we have proposed are based on a distributed fixpoint iteration, where roles are iteratively reassigned to nodes until a global configuration is achieved that satisfies all rules. While we could show that the examined role assignment specifications converge within very few iterations, there are also *infeasible* specifications for which no assignment of roles to nodes exists that satisfies all rules. For such specifications, our distributed algorithms do not converge. Mechanisms are therefore needed to detect such faulty specifications to assist a developer. The second open issue is that many practical role assignment problems include some global *optimization criteria*, such as the minimization of the number of `ON` nodes in the above coverage problem. Such problems cannot be expressed with our previous proposal.

The contribution of this paper is three-fold. Firstly, we introduce a simple extension to our role assignment language to express global optimization criteria. Secondly, we provide a mapping of the modified language to Integer Linear Programs (ILP). This mapping is implemented in an existing tool using the CPLEX solver. Using this tool, we, thirdly, re-examine some of the role assignment problems presented in [5]. In particular, we show how infeasible (non-terminating) specifications can be detected, and

*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.

1-4244-0054-6/06/$20.00 ©2006 IEEE

quantify improvements that can be achieved by using global optimization criteria.

Primarily, the contribution of the present work can be characterized as a tool to analyze certain properties of role specifications. The proposed approach could also be used to actually implement role assignment in a sensor network (as a replacement of our earlier distributed algorithms). For this, however, a global view on the network topology and node properties must be collected to solve the centralized ILP, and computed role assignments must then be delivered to the network. This is only sensible in networks with static properties.

In the remainder of this paper, we propose an extended specification language in Section 2 and describe the mapping of this language to integer linear programs in Section 3. Results that have been obtained with an implementation of this mapping for existing role specifications are discussed in Section 4.

2. Role Specifications

In this section we briefly review the role specification language that has been introduced in [5] and describe an extension to specify optimization criteria.

2.1. Syntax and Semantics

A role specification consists of a list of role definitions, where each role definition consists of a role identifier and an associated predicate (i.e., a rule). The latter is a Boolean expression formulating conditions on the local properties of a sensor node and on the properties of well-defined sets of nodes in the neighborhood of the node.

Let us first consider a simple example before giving a more formal definition of the language elements. Figure 1 (a) shows the code for a simplified variant of the coverage problem discussed in the introduction, where the sensing and communication ranges are assumed to be identical. Two role definitions are shown for role `ON` (lines 1-5) and for role `OFF` (line 6). The specification requests that role `ON` is assigned to a node which has a battery level above a certain threshold (line 2), and where at most one other node (line 5) with role `ON` (line 4) is within communication and sensing range (lines 3-5). Role `OFF` is assigned if and only if the conditions for role `ON` are not met (line 6).

More formally, a language specification is a list of pairs (k, c^k) with role identifiers k and predicates c^k. Without loss of generality, we assume that c^k is given in 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:

2.1.1 Simple predicates

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

```
battery >= threshold
```

2.1.2 Count predicates

Count predicates have the form

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

and 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`**, for example:

```
count(1 hop) {
  role == CH
} <= 1
```

2.1.3 Retrieve predicates

Retrieve predicates are similar to count predicates, these have the form

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

and can be used to bind the identities of a set of nodes matching **`pred`** to a local property variable **`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. Example:

```
clusterheads == retrieve(1 hop, 2) {
  role == CH
}
```

In 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 original node instead:

```
count(2 hops) {
  super.battery < battery
}
```

Property values such as `battery` in the above example are stored in a so-called property directory on each node. This directory supports entries of numeric and Boolean types (e.g., `battery`, `temp-sensor`), further node positions, sets of node IDs (e.g., `clusterheads`), and the enumeration `role` (i.e., the role that has been assigned to the node). 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 retrieve predicates bind local properties that can be referenced by other *count* and *retrieve* statements. Hence, predicates must be evaluated in an order that ensures that properties are set before they are used. This also implies that there must not be circular dependencies between any two retrieve statements that are part of the conditions of any role.

In addition to these role definitions which have been introduced in previous work [5], we allow the specification of global optimization criteria of the form `min` **`role`** or `max` **`role`**, where **`role`** can be any of the roles that have been defined, for example:

```
min ON
```

We allow multiple such specifications to demand minimization or maximization of the respective set of roles.

Finally, we note that in our previous distributed implementations of role assignment [5], roles defined earlier in the specification are given priority over later ones. That is, if the predicates of multiple roles match, the role that is defined first is assigned. This relieves the programmer from (the often non-trivial) task of ensuring that all role predicates are logically disjoint. The mapping of role specifications to ILP described later will implement the same semantics.

2.2. Example Specifications

The above specification language can be used to describe a variety of different role assignment problems, please refer to [5] for details. However, to make the paper self-contained, we will briefly discuss the specification of a clustering problem in addition to the coverage specification which has already been introduced in the previous section.

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

```
1 ON :: {
2   battery >= threshold &&
3   count(1 hop) {
4     role == ON
5   } <= 1 }
6 OFF :: else
```

(a) Simplified coverage.

```
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
```

(b) Clustering.

Figure 1. Sample role specifications.

Consider Figure 1 (b), which shows a role specification implementing clustering. 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. For this, 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. 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.

Only if a node is neither a gateway nor a clusterhead, it is assigned role `SLAVE` in line 13.

3. Mapping Role Specifications to Integer Programs

In this section we provide a mapping of role specifications to integer linear programs. An instance of a role assignment problem consists of a role specification with m roles using the language described in Section 2, all property values of all nodes that are referenced by the role specifica-

tion, and a sensor network graph $G = (V, E)$ with $n = |V|$ participating nodes.

G is used to define the elements of the h-hop neighborhood matrices $\mathbf{A}^{(h)}$ as follows:

$$A_{ij}^{(h)} = \begin{cases} 1 & \exists \text{ path from node } i \text{ to node } j \\ & \text{with length } \le h \\ 0 & \text{otherwise} \end{cases}$$

A total of $h = 1 \dots S$ such matrices will be generated, where S denotes the maximum scope that occurs in all *count* and *retrieve* statements of a specification. Note that these matrices can be readily computed from the adjacency matrix $\mathbf{A}^{(1)}$. We will use the above notation $A_{ij}^{(h)}$ to formulate the ILP constraints of *count* and *retrieve* operators in Section 3.3 below.

In the next section, we introduce the variables we use to encode the outcome of the role assignment algorithm. Unless otherwise noted, all variables are binary with values $\in \{0, 1\}$.

3.1. Role Assignment Variables

We use a set of binary variables x_{ik} to encode whether a node i satisfies the predicate c^k of a given role k:

$$x_{ik} = \begin{cases} 1 & \text{if node } i \text{ satisfies } c^k \\ 0 & \text{otherwise} \end{cases} \quad (2)$$

By c^k we refer to the Boolean predicate describing the conditions for assuming role k as discussed in Section 2.1. We will ensure that (2) holds by translating c^k into a set of equivalent constraints on x_{ik} in Section 3.3.

Furthermore, we require that at least one role predicate must match for every node, otherwise, the role assignment is not feasible, giving rise to the constraint:

$$\sum_k x_{ik} \ge 1 \quad \forall k, i \quad (3)$$

Note that to avoid infeasibility, the programmer can always specify an `else` role q which does not imply any constraints as it does not have any sub-predicates (see Section 2). Therefore, the predicate of an `else` role would always be satisfied, and, according to the above definition (2), a solution would set x_{iq} to 1.

This points to a different property of the variables x_{ik}: The properties and network neighborhood of a given node i could satisfy the conditions for more than one role, as we do not require the programmer to specify disjunct conditions in the predicates c^k. Therefore, more than one c^k can be satisfied, requiring (because of (2)) that more than one x_{ik} is set to 1 for a given node i. Thus, x_{ik} cannot be used as a result of role assignment according to the semantics defined in Section 2, which imply that the *first* matching role with $x_{ik} = 1$ is assumed at a node i.

We implement these *first role matches* semantics using an additional set of binary result variables y_{ik} which we require to be 1 *only* if the conditions for roles $1, ..., k-1$ (i.e., roles specified *prior* to k) are not satisfied. Here we assume that k denotes role k's position in the list of roles specified by the programmer (we can achieve this by simply renumbering the roles). The following constraint formulates the necessity that $y_{ik} = 1$ if the given x_{ik} is 1 while all x_{il} with $l < k$ are 0:

$$x_{ik} - \sum_{l=1}^{k-1} x_{il} \le y_{ik} \quad \forall k, i \quad (4)$$

Moreover, we also require in (5) that role k can only be assigned *if* the corresponding predicate matches, i.e., $x_{ik} = 1$, and further in (6) that exactly one role should be assigned.

$$y_{ik} \le \quad x_{ik} \quad \forall k, i \quad (5)$$

$$\sum_k y_{ik} = \quad 1 \quad \forall i \quad (6)$$

Finally, we introduce binary variables for node sets that are bound to the results of retrieve predicates, e.g., `clusterheads` for the role *gateway* of the clustering example. For each such set p bound by a retrieve predicate, we include an additional set of variables $q_i^p(j)$ with $j \in V$ with the following interpretation:

$$q_i^p(j) = \begin{cases} 1 & \text{if the set } p \text{ at node } i \text{ contains node } j \\ 0 & \text{otherwise} \end{cases} \quad (7)$$

To preserve a readable notation, we will sometimes omit the index p and just write $q_i(j)$ implicitly referring to the respective property p.

3.2. Objective Function

We can now reformulate the optimization criteria described in Section 2.1 into an objective function in terms of the variables y_{ik}. Assume that among a set of roles R, for a given problem instance, the user would like to minimize the set of roles $m \subset R$ and maximize the set of roles $M \subset R$ in the network. The corresponding objective function is:

$$\min \sum_{i \in V} \left(\sum_{k \in m} y_{ik} - \sum_{k \in M} y_{ik} \right)$$

Note that it would be easily possible to formulate more complex optimization criteria using the y_{ik}, such as maintaining certain ratios between `ON` and `OFF` in the coverage example.

3.3. Translating Predicates

In this section we will show the mapping of role predicates c^k to respective ILP constraints on x_{ik}. Note that the constraints will depend on i because the network neighborhood is a function of the node identity i.

Consider a role predicate given in the normal form of Section 2.

$$c^k = \underbrace{(c^k_{11} \wedge \cdots \wedge c^k_{1n_1})}_{a^k_1} \vee \underbrace{(c^k_{21} \wedge \cdots \wedge c^k_{2n_2})}_{a^k_2} \vee \dots \quad (8)$$

As a first step, we translate the Boolean operations to ILP constraints. This is done by means of a (standard) ILP modeling technique, which uses additional indicator variables $u_i(c)$, similar to x_{ik}, for each atomic predicate c occurring in (8) that indicate whether c is satisfied at a given node i.

Disjunctions and conjunctions can be expressed in terms of ILP constraints as follows. Assume a conjunctive term of the form ($c = c_1 \wedge \cdots \wedge c_p$), consisting of p terms, and let $u_i(c_1) \dots u_i(c_p)$ be the respective indicator variables for a given node i. We will use an additional variable $u_i(c)$ to indicate whether the whole conjunction c is true. We require therefore that $\sum u_i(c_q) \geq p$ if and only if $u_i(c) = 1$. The constraints modeling necessity and sufficiency for every node i are:

$$\left.\begin{array}{rcl} \sum_{q=1}^{p} u_i(c_q) & \leq & u_i(c) + p - 1 \\ \sum_{q=1}^{p} u_i(c_q) & \geq & p \times u_i(c) \end{array}\right\} \forall i \quad (9)$$

Similarly, for an analogous disjunction of the form $a = a_1 \vee \cdots \vee a_p$, we require that at least one of the indicator variables $u_i(a_q)$ of a node i is 1, thus $\sum u_i(a_q) \geq 1$ if and only if $u_i(a) = 1$, for all nodes i. The constraints modeling necessity and sufficiency are:

$$\left.\begin{array}{rcl} \sum_{q=1}^{p} u_i(a_q) & \leq & p \times u_i(a) \\ \sum_{q=1}^{p} u_i(a_q) & \geq & u_i(a) \end{array}\right\} \forall i \quad (10)$$

We will use (9) and (10) on several occasions when we need to model conjunctions or alternatives.

In the following, we show how the atomic predicates c^k_{qr} of (8) are mapped to constraints on the indicator variables $u_i(c_{qr})$.

3.3.1 Simple predicates

We begin with simple predicates c that are *local* in that they refer only to the properties of a given single node. These can be formulated in terms of known property values of a node – essentially constants – and can be evaluated before generating the ILP. Hence, the indicator variable $u_i(c)$ would be replaced by either 0 or 1 in this case.

Simple predicates that are *nested* in *count* or *retrieve* statements are special cases that will be considered in Section *Nested Predicates* below.

3.3.2 Count predicates

Consider a count predicate c of the form:

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

In the following we formulate equivalence between $u_i(c) = 1$ and c. Let $u_j^{\texttt{(pred)}}$ be the variable indicating whether **pred** is true at a node j. We constrain $u_i(c)$ as follows to formulate the necessity that $(u_i(c) = 1) \rightarrow c$:

$$\sum_{j \neq i} A_{ij}^{\texttt{(scope)}} u_j^{\texttt{(pred)}} \quad \leq \quad (1 - u_i(c))M + \texttt{lim}$$

We use M to denote a constant value greater than n. Note that the above reduces to either (11) or (12) if the indicator $u_i(c)$ is 1 or 0, respectively:

$$\sum_{j \neq i} A_{ij}^{\texttt{(scope)}} u_j^{\texttt{(pred)}} \quad \leq \quad \texttt{lim} \quad (11)$$

$$\sum_{j \neq i} A_{ij}^{\texttt{(scope)}} u_j^{\texttt{(pred)}} \quad \leq \quad M + \texttt{lim} \quad (12)$$

The former case (11) exactly formulates the semantics of the above count predicate c, namely that the number of nodes j within **scope** that match **pred** should be less or equal to **lim**. In the latter case (12), the constraint is nullified by M, as $M > n$ and the left-hand sum will never be larger than the number of nodes.

In a second step we formulate the constraints for sufficiency, namely that $c \rightarrow (u_i(c) = 1)$, i.e., either $\neg c$ or $u_i(c) = 1$:

$$\sum_{j \neq i} A_{ij}^{\texttt{(scope)}} u_j^{\texttt{(pred)}} \quad \geq \quad (\texttt{lim} + 1)(1 - u_i(c))$$

Note that count operators using relations other than $\leq$ can be treated analogously.

3.3.3 Retrieve predicates

A retrieve predicate c of the form

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

can only be true if the following three requirements are met.

Firstly, at least **size** nodes must exist within **scope** that match the given nested predicate **pred**. We model this requirement as a count statement of the form:

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

Secondly, we must ensure that every retrieved node j that is in the set **p** of node i really is in **scope** and also matches the predicate **pred**:

$$q_i^{\mathbf{p}}(j) \leq A_{ij}^{(\texttt{scope})} u_i^{(\texttt{pred})} \tag{13}$$

Thirdly, we require that the number of elements in the set **p** is **size**:

$$\sum_{j=1}^{n} q_i^{(\mathbf{p})}(j) = \texttt{size} \tag{14}$$

And finally, we can formulate that $u(c) = 1$ if and only if all of the above requirements hold at the same time using the approach in (9).

3.3.4 Nested predicates

In the following consider predicates that occur nested in *count* or *retrieve* statements with a given **scope**.

For a nested predicate c that refers to the *role* of the node, i.e., `role==k`, we set $u_j(c)$ to the value of the respective variable indicating whether the role of a node j is k, namely y_{jk}. In fact, a separate respective indicator variable $u_j(c)$ is not needed in this case, as $u_j(c)$ can be replaced with y_{jk}.

A special case are simple predicates c that check equality between two properties that represent node sets such as

```
clusterheads == super.clusterheads
```

in the clustering example.

Here we introduce additional indicator variables Q_{ij} that are set to 1 iff the set **clusterheads** of node i is equal to the set **super.clusterheads** at node j. The interesting case is when the two sets are bound by retrieve predicates (as in the above clustering example), and thus constitute variables of the algorithm. In these cases, the parameter **size** of the respective retrieve predicates indicates the size of the set (we assume the sets are of equal size, otherwise the compiler can already set $u(c)$ to 0). In all other cases Q_{ij} can be evaluated before generating the ILP and simply noted as 1 or 0, respectively.

In the following, we describe the constraints on Q_{ij}. These are formulated in terms of the variables $q_i(k)$ representing the set of node i, where $q_i(k) = 1$ iff the element $k \in \{1, ..., n\}$ is contained in the set at node i and 0 otherwise, as defined in (7). Likewise, respective variables $q_j(k)$ for node j are used. We first compute a set of n helper variables $q_{ij}(k) = q_i(k) \wedge q_j(k)$ using Equation (9) as shown in Figure 2 for two exemplary sets of size 2, where a black circle indicates $q_*(k) = 1$.

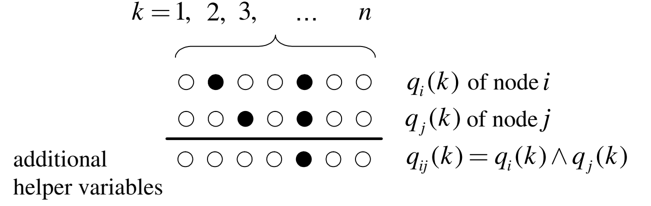


Figure 2. Set equality.

Using $q_{ij}(k)$, we can formulate that variables Q_{ij} should be 1 if the intersection of the sets at node i and j contains at least **size** elements (i.e., if the sets are equal):

$$\sum_{k=1}^{n} q_{ij}(k) \leq Q_{ij} - 1 + \texttt{size} \tag{15}$$

Otherwise, Q_{ij} should be 0:

$$\sum_{k=1}^{n} q_{ij}(k) \geq \texttt{size} \times Q_{ij} \tag{16}$$

Finally, we can replace $u_j(c)$ that occur nested within a `count` or `retrieve` statement of a given node i by the corresponding Q_{ij}.

3.4. Complexity

Depending on the complexity of the role specifications, the resulting integer programs are solvable in reasonable time for up to 1000 nodes using the CPLEX [4] standard solver. However, *retrieve* predicates introduce considerable complexity due to the large number of indicator variables. Note that the above mapping could be improved to reduce the number of variables, which has been omitted for ease of exposition. Below we will analyze the number of variables used in the ILP mapping as an indicator for the complexity of the generated ILP.

In the above Section 3.1 we used a total number of $n \times m$ variables to encode the basic role-assignment decision, where m denotes the number of specified roles and n denotes the number of nodes in the network. Further, we added another set of n variables for each node, to encode local properties containing node sets that are bound by retrieve statements, resulting in another $n^2 \times b$ variables, where b stands for the total number of local properties bound by retrieve statements. In the mapping of predicates in Section 3.3, we introduced a number of additional indicator variables. A total of $n \times a$ variables are used to encode atomic predicates at each node, where a is the number of atomic predicates in the specification. The most complex

atomic predicate – set equality – required additional n indicator variables for each *pair* of nodes, summing up to n^3 variables.

However, it would be straightforward to reduce the number of variables by exploiting the locality (i.e., limited scope of count and retrieve predicates) inherent in the role specification language. Note that the language was originally aimed at supporting the user/programmer with a flexible means to describe *local* configuration decisions taken at each node. In previous work [5], we described a number of distributed role assignment algorithms that exploited the locality in the specification, such that the communication overhead is strictly dependent on the scope used in count and retrieve predicates.

Similarly, the minimal number of variables needed for the ILP also depends on the size of the scope used in count and retrieve statements. Consider a retrieve statement binding a local property **p**. Due to the limited scope, rather than n variables, each node i will only need k_i variables to encode **p**, where k_i corresponds to the number of nodes in the *scope*-hop neighborhood of i as illustrated in Figure 3. Moreover, the set of nodes that needs to be encoded in the helper variables q_{ij} is even smaller, as $q_{ij}(p)$ must only be provided for nodes p which are located in the intersection of the scopes of nodes i and j.

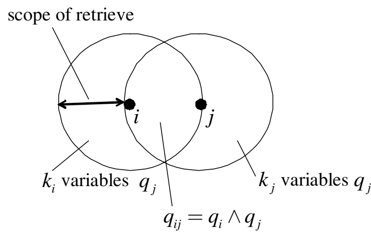


Figure 3. The minimal number of variables needed to represent retrieved node sets and their equality is relatively small and dependent on the given scopes.

With these improvements, we obtain a much smaller number of variables and coefficients. For example, for a 100 node network with nodes being placed randomly in a 300m by 300m area, the ILP implementing the clustering specification shown in Figure 1 (b) contains only 2428 variables (rather than 100^3) and is solvable within seconds using CPLEX [4].

4. Results

We extended the existing role assignment simulation tool [5] with the capability of generating an ILP representing the execution of a given specification on a given topology. The generated ILP is formulated using the ZIMPL [7] modelling language, which we use as a front end for the CPLEX [4] standard solver. Finally, the simulation tool visualizes the ILP's results on the respective network topology.

Using the above ILP mapping of role specifications, we can re-examine the role-assignment specifications described in [5] regarding *optimality*, *feasibility* and *termination*, which were open problems so far.

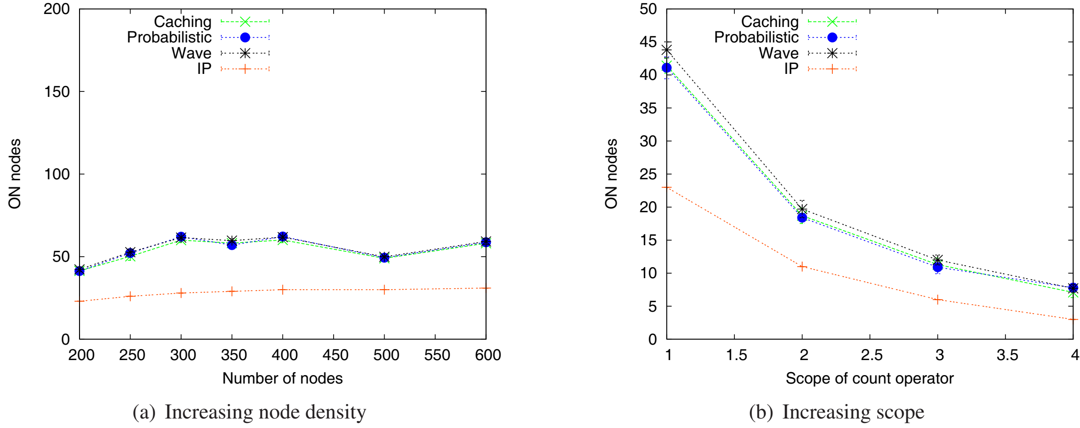
One new aspect of the generated ILP is that it enables the programmer to express a desired *optimality* goal, i.e., to minimize or maximize the number of nodes that are assigned a certain role. While the existing specifications were not written with a possible optimization in mind – rather, these were designed as input for distributed algorithms that assign roles in a greedy fashion –, it is nevertheless interesting how the *optimal* results compare to the ones found using the distributed algorithms described in [5].

As illustrated in Figure 4, we examined the minimal number of `ON` nodes required to ensure coverage. Using the specification in Figure 1 (a), we could easily solve the resulting ILP for random networks with up to 600 nodes using CPLEX. The nodes with a communication range of 33.5m were randomly distributed in a square of 300m by 300m.

Previously, we had studied the same specification with several algorithms (*caching*, *probabilistic* and *wave*) in terms of their efficiency regarding communication traffic, but did not employ any strategy to minimize the number of `ON` nodes. It is therefore not surprising that the ILP approach can yield solutions with about half as many `ON` nodes as the distributed algorithms. While we cannot expect to achieve the same results with a optimizing distributed algorithm (part of current work), the optimal solution obtained via the ILP indicates the possible space for optimization.

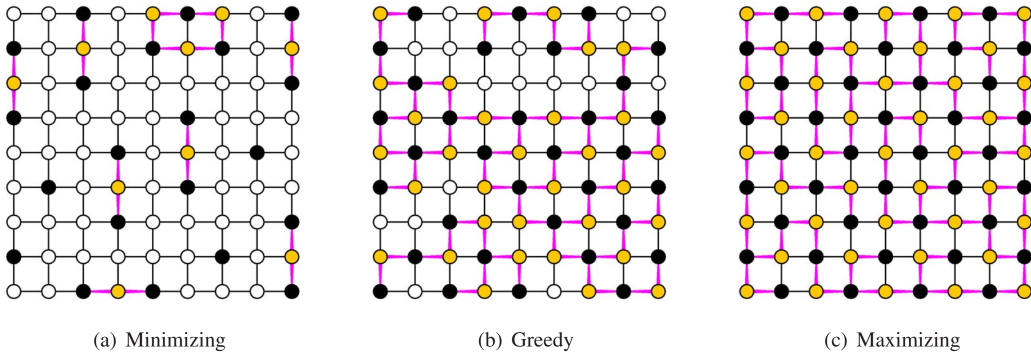
Moreover, optimization can provide interesting insights into undesirable effects of a role specification. While the clustering specification in Figure 1 (b) results in a connected backbone network of clusterheads and gateways with the distributed algorithms as depicted in Figure 5(b), minimizing the `CH` and `GW` roles results in a set of isolated clusters as depicted in Figure 5(a), which may not be desirable for practical applications. Here, the solution places clusterheads exactly three hops away from each other, thus ensuring that each slave is a neighbor of at least one clusterhead, while at the same time avoiding assignment of the gateway role (as hardly any node has more than one clusterhead neighbor). This hints at a weakness of the specification, namely that it does not enforce that clusterhead and gateway nodes should provide a *connected* backbone.

For the case that clusterhead and gateway roles are maximized, an exemplary solution is shown in Figure 5(c). Here,



(a) Increasing node density (b) Increasing scope

Figure 4. Number of ON nodes in simplified coverage example.



(a) Minimizing (b) Greedy (c) Maximizing

Figure 5. Clustering results on a 9x9 grid, minimizing vs. maximizing clusterheads and gateways (clusterheads are black, gateways are yellow, slaves are white, and edges between gateways and their clusterheads are emphasized).

clusterhead nodes are exactly two hops away and arranged in an ordered fashion that allows many gateway nodes. Essentially, Figures 5(a) and 5(c) show two extremal outcomes that may occur with the current clustering specification – hinting the developer that there is a potential problem with this specification.

Enabling the programmer to formulate an additional constraint, namely that a set of roles should be *connected*, would be an interesting research direction, which would provide a complementary extension to related work [12] that studies various criteria that determine whether a set of roles is connected in the average case, as connected topologies are needed for a code deployment algorithm.

Finally, the ILP translation helps to better understand erroneous specifications that do not *terminate* when using the current distributed algorithms. Consider the following toy example from previous work [5]:

```
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. The distributed algorithms we developed in previous work would change back and forth between the roles RED and GREEN. Using the ILP mapping of the above specification we could show that problem instances used in previous evaluations – a varying number of 100–600 nodes randomly distributed on a 300m by 300m area – were in fact *infeasible* since the ILP did not have a solution.

While this approach shows infeasibility for a given network graph, we can also reformulate the ILP to check whether there exists *any* network graph for which the problem has a solution. Doing so for the above specification, we obtained graphs of single isolated GREEN nodes, as the only feasible combination of network topology and role specification. When adding the (practical) requirement that every node should have at least one neighbor, no solution can be

found, indicating infeasibility of the specification for *any* possible network topology.

For illustration of this result, consider the simplest graph with two nodes u, v and an edge (u, v) as shown in Figure 6. Looking carefully, we can see that, out of the 4 assignments that are possible in this graph, none is feasible. While the last example (both nodes green) seems feasible at first, it is not: As both nodes also match the predicate for RED, the assignment does not comply with the requirement that the *first* matching role should be assigned.

This example suggests that there is a close relationship between specifications that do not terminate when executed by the distributed algorithms and specifications that result in infeasible ILPs. At the least, infeasible specifications are highly likely to be non-terminating. However, this is a conjecture which needs to be confirmed as part of future work.

Figure 6. RED / GREEN example: Out of the 4 possible role assignments, none satisfies the role specification.

5. Related Work

Numerous approaches for solving *specific* role assignment problems have been devised. Examples include coverage [15]; aggregator placement [3] and data fusion [10]; clustering, routing and addressing [8, 16, 17]. [8] uses a fixed set of roles to build a network-wide backbone infrastructure.

The need for generic role assignment has been expressed in previous work, among others, in [6] as part of the MiLAN project, in [19] to allocate tasks to sensor nodes, and in [12] in the context of role-based code updates.

Integer program formulations of network configuration problems have also been applied in various settings. In [2], the authors derive upper bounds on the lifetime of data-gathering networks by computing optimal configurations consisting of the roles *sensor*, *relay*, and *aggregator*. Moreover, integer program formulations have been used as a starting point for developing distributed approximation algorithms which solve problems that are related to our specifications, most prominently the minimum dominating set [9] and, recently, the facility location problem [13].

However, none of the approaches above are *generic* frameworks that support the assignment of user-defined roles in an application-specific manner.

6. Conclusion and Outlook

In this paper, we described an extension to our role assignment system that maps a given role assignment instance – consisting of a network topology, a set of node properties, and a role assignment specification – to an integer linear program formulation.

We showed that this mapping, together with a simple extension to the role specification language that allows specifying global optimization criteria, provides the user with a valuable tool for analyzing the feasibility, optimality, and termination of role assignment specifications, aspects which have been open questions so far. Most prominently, we were able to show that a known non-terminating specification is infeasible unless applied to a trivial network graph consisting of isolated nodes.

Future work includes adding support for requesting connectivity properties for certain roles [1], and finding ways to identify non-terminating role specifications.

References

[1] A. Ahuja, T. L. Magnanti, and J. B. Orlin. *Network Flows: Theory, Algorithms, and Applications*, chapter 13, page 530ff. Prentice-Hall, 1993.

[2] M. Bhardwaj and A. Chandrakasan. Bounding the lifetime of sensor networks via optimal role assignments. In *Proceedings of the 21st Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM'02)*, New York, NY, USA, 2002.

[3] B. Bonfils and P. Bonnet. Adaptive and decentralized operator placement for in-network query processing. In *Proceedings of the 2nd International Workshop on Information Processing in Sensor Networks (IPSN'03)*, Palo Alto, CA, USA, Apr. 2003.

[4] CPLEX ILP solver. http://www.ilog.com/products/cplex/.

[5] C. Frank and K. Römer. Algorithms for generic role assignment in wireless sensor networks. In *Proceedings of the 3rd International Conference on Embedded Networked Sensor Systems (SENSYS'05)*, San Diego, CA, USA, Nov. 2005.

[6] W. B. Heinzelman, A. L. Murphy, H. S. Carvalho, and M. A. Perillo. Middleware to support sensor network applications. *IEEE Network*, pages 6–14, Jan. 2004.

[7] T. Koch. ZIMPL *(Zuse Institute Mathematical Programming Language) User Guide*, Feb. 2005.

[8] M. Kochhal, L. Schwiebert, and S. Gupta. Role-based hierarchical self organization for wireless ad hoc sensor networks. In *Proceedings of the 2nd ACM International Workshop on Wireless Sensor Networks and Applications (WSNA'03)*, San Diego, CA, USA, 2003.

[9] F. Kuhn and R. Wattenhofer. Constant-time distributed dominating set approximation. In *Proceedings of the 22nd ACM Symposium on Principles of Distributed Computing (PODC'03)*, July 2003.

[10] R. Kumar, M. Wolenetz, B. Agarwalla, J. Shin, P. Hutto, A. Paul, and U. Ramachandran. DFuse: A framework for

distributed data fusion. In *Proceedings of the 1st International Conference on Embedded Networked Sensor Systems (SENSYS'03)*, Los Angeles, CA, USA, Nov. 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, Jan. 2002.

[12] P. J. Marron, A. Lachenmann, D. Minder, J. Hähner, R. Sauter, and K. Rothermel. TinyCubus: A flexible and adaptive framework for sensor networks. In *Proceedings of the 2nd European Workshop on Wireless Sensor Networks (EWSN'05)*, pages 278–289, Istanbul, Turkey, Jan. 2005.

[13] T. Moscibroda and R. Wattenhofer. Facility location: Distributed approximation. In *Proceedings of the 22nd ACM Symposium on Principles of Distributed Computing (PODC'03)*, pages 108–117, 2005.

[14] K. Römer, C. Frank, P. J. Marron, and C. Becker. Generic role assignment for wireless sensor networks. In *Proceedings of the 11th ACM SIGOPS European Workshop*, Leuven, Belgium, Sept. 2004.

[15] S. Slijepcevic and M. Potkonjak. Power efficient organization of wireless sensor networks. In *Proceedings of the IEEE International Conference on Communications (ICC'01)*, Helsinki, Finland, June 2001.

[16] 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.

[17] L. Subramanian and R. H.Katz. An architecture for building self-configurable systems. In *Proceedings of the 1st ACM International Symposium on Mobile Ad Hoc Networking and Computing (MOBIHOC'01)*, Boston, MA, USA, Aug. 2000.

[18] 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.

[19] 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, Mar. 2005.