Building Application Stack (BAS)

Andrew Krioukov, Gabe Fierro, Nikita Kitaev, David Culler
Computer Science Department
University of California, Berkeley

krioukov@cs.berkeley.edu, gt.fierro@berkeley.edu,
kitaev@berkeley.edu, culler@cs.berkeley.edu

Abstract

Many commercial buildings have digital controls and extensive sensor networks that can be used to develop novel applications for saving energy, detecting faults, improving comfort, etc. However, buildings are custom designed, leading to differences in functionality, connectivity, controls and operation. As a result today's building applications are hard to write and non-portable. What is required is a form of mass customization that allows applications to automatically adapt to differences in buildings.

We present BAS, an application programming interface and runtime for portable building applications. BAS provides a fuzzy query interface allowing application authors to describe the building components they require in terms of functional and spatial relationships. The resulting queries implicitly handle multiple building designs. BAS also incorporates a hierarchical driver model, exposing common functions of building components through standard interfaces.

We demonstrate and evaluate BAS by implementing two novel applications – an occupant HVAC control app and a ventilation optimization app – on two different buildings using raw building control protocols and then again using BAS. We show that the BAS code is much shorter, easier to understand and does not change for each building.

Categories and Subject Descriptors

C.3 [**Special-Purpose and Application-Based Systems**]: Process Control Systems

General Terms

Design, Experimentation, Management

Keywords

Building Applications, Controls, Energy Efficiency

1 Introduction

Commercial buildings are one of the largest consumers of energy in the United States, accounting for 19% of delivered energy [18]. Nearly half of this energy (42%) is consumed in buildings with existing digital control systems [9] comprising some of the largest deployed sensor networks and often containing thousands of sensors and actuators per building. This existing infrastructure is a potential goldmine, enabling new analyses and cyber-physical applications that can be implemented in software alone. These applications have the potential to drastically improve building energy use, occupant comfort, reliability and maintenance (e.g. [10, 12, 17]).

However, the widespread development and use of these apps is inhibited by *lack of portability*. Buildings are custom designed, using similar templates but each with a unique shape, layout, heating, ventilation, air conditioning, lighting and electrical system. Today, this necessitates writing custom applications for each building, requiring a deep understanding of each building's architecture, connectivity, and control operation – effectively, the building equivalent of programming in assembly. Instead, what is required is a form of *mass-customization*, a way of writing applications to automatically adapt to a wide range of buildings.

BAS is an application programming interface and runtime that enables writing portable code by providing methods to explicitly and implicitly handle differences in building designs. A key insight of BAS is the use of fuzzy, relativistic queries to allow authors to express their high-level intent in a way that is inherently portable, e.g. "turn off the lights for top floor cubicles near windows," as well as supporting programmatic exploration of a building's specific components, allowing applications to explicitly handle building differences. Thus programmers can alternate between macro and micro level views of the building (e.g. "lights on the top floor" vs. "Light Relay 1023") to express both general intentions and specific actions.

Existing protocols such as BACnet [1], OPC [14], and LonTalk [7] provide discovery, networking and read/write access to sensors and actuators. This functionality is crucial, but alone is insufficient for portability. Applications must know *which* components to access and how they relate to each other. This requires key metadata: how a component is functionally linked to others, where a component is physically located and how it can be addressed on the network. BAS provides a way to express this information as a graph of objects and provides a query API for applications to efficiently refer to the desired components in a portable way.

Of course, this metadata can also be hardcoded in lookup tables or configuration files, as it is often done in today's building apps, but this ad-hoc approach is error-prone and inefficient. Having each application create its own view of the building requires all authors to have a deep understanding of each building's design and requires all applications to be updated if a change is made or error corrected in the building model. Moreover, creating an easy to use interface once benefits all applications and lowers the bar for new development.

We make the following contributions: (1) we propose a fuzzy query API and graph representation of building metadata (2) we construct a hierarchical driver model for building components, minimizing building specific code and (3) we evaluate BAS by implementing two control applications, executed on two real buildings on campus, first using only the existing building control protocol and then again using BAS.

2 Architecture

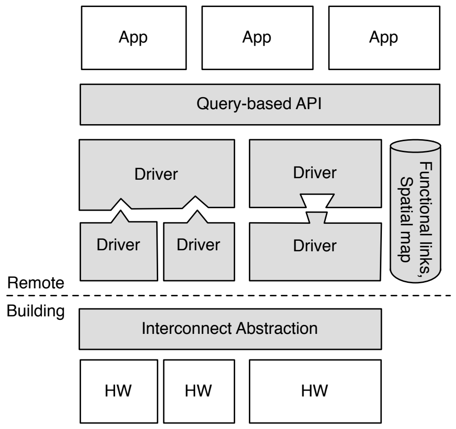

Figure 1. BAS layered architecture.

The goal of BAS is to provide an API and runtime that allow applications to run on a wide range of buildings. At the same time, BAS aims to minimize the effort required to set up the system on a new building by factoring out common components and partially automating metadata collection.

The BAS architecture is shown in Figure 2. Buildings with digital controls contain sensors and actuators wired to a number of controller panels: embedded computers with analog input and output ports. In turn, these controllers are networked with serial or ethernet links and optional routers and gateways, connecting to a computer with a graphical interface for monitoring the building. Buildings often have multiple independent control networks, such as a lighting control system and an HVAC controller.

At the lowest level, BAS interfaces with the building hardware through a control protocol such as BACnet or OPC. The protocol exposes a network view of the building components consisting of a point name, control panel identifier and optional description string for each sensor output and actuator. For example, a point named "SDH.AH1A.SF_VFD:INPUT REF 1" on panel "SDH.MBC-01" represents the speed input to the variable-frequency drive controlling the supply fan in one of our buildings. This point takes a value between 0 and 100 corresponding to a scale of off to maximum speed for the fan. The BAS interconnect abstraction layer acts as a hardware interface, exposing all points from different control networks with different protocols in a uniform way by using the sMAP architecture, as discussed in Section 5.

While having access to building sensor and actuator points is sufficient for implementing applications, point names and functionality are very building-specific, thus programming at this level leads to non-portable code. The BAS driver layer abstracts groups of points into functional objects with standardized methods. Each driver type must expose at least a minimum predefined interface. For example, fan drivers expose "get_speed" and "set_speed" methods. These methods apply to many different fan designs, e.g. variable speed, 3-speed, or on/off fans. Drivers can be built up hierarchically so that common functionality is implemented once, as shown in Figure 4.

The objects exposed by drivers correspond to physical components within the building, e.g. chillers, light banks, electrical circuits, etc. Drivers abstract the inner workings of these devices, but since buildings are all constructed differently the functional and spatial relationships between these objects are crucial. Being able to answer *which* air handler serves a particular room or *which* circuit powers a light bank is key to expressing control actions in a general way. BAS captures functional relationships in a directed graph of "supplies" relations, effectively capturing airflow loops, water loops and electrical trees. In addition, BAS incorporates spatial tags stored in a GIS database, allowing us to answer queries such as, "select the lights for all rooms near windows" or "select the air handler that supplies room 123."

At the highest level, BAS provides a query-based API for writing applications. The API consists of a selector query for choosing a set of driver objects corresponding to building components based on object name, type, attributes, functional relationships and spatial relationships. Each object type has a predefined minimal set of methods, i.e. sensor readings and actuation capabilities. The application can make use of these guaranteed methods or discover all available methods. Similarly, objects can be selected with a precise query, e.g. "the thermostat for room 123," or by exploring the graph of functional relationships, e.g. "all temperature sensors upstream of room 123." Applications access the BAS API through a web service and can be written in any language.

3 Query Interface

The BAS query interface is designed to allow application authors to select objects based on type, attributes and functional or spatial relationships. This allows authors to *describe* the particular sensor or actuator that the application requires rather than hardcoding a name or tag that may not apply in differently designed buildings.

Queries are expressed in terms of names indicated with a $ prefix, tags starting with # and relationships indicated by $<$ and $>$ operators with $A > B$ meaning that A supplies or feeds into B. For example, an air handler might supply variable air volume (VAV) boxes that supply rooms; a whole building

Table 1. Example BAS queries showing fuzzy and relative lookups. $A > B$ means A that feed into or supplies B; $X < Y$ means the X that is fed by or supplied by Y. Operators are right associative.

`#FLOOR`	All floor areas
`#LIGHT > $Floor 1`	All lights on the first floor
`#AREA < $Lightbank 1`	Zones served by light bank 1
`#AH > $Room 123`	All air handlers that serve Room 123
`#AREA < #AH > $Room 234`	Areas served by the same air handler as Room 234
`#ELECMETER > #AREA < $Air Handler 1`	Power meters for zones served by AH1

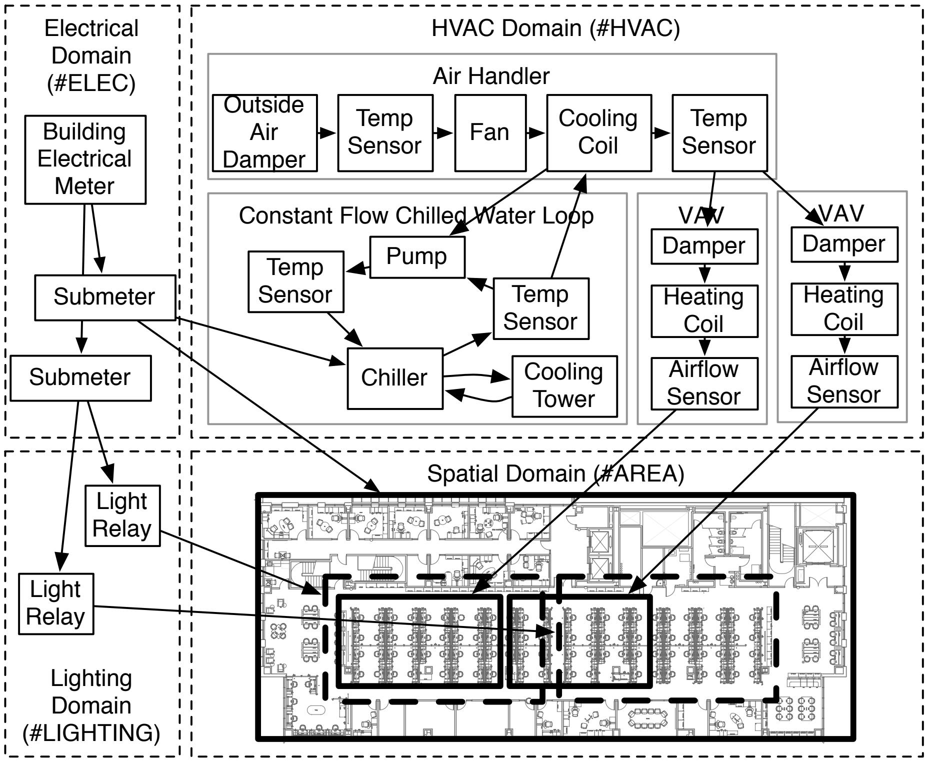

Figure 2. Partial functional and spatial representation of our test building. Directed edges indicate a supplies or feeds into relationship. BAS queries are executed by searching the graph.

power meter may feed into multiple breaker panels that supply different floors. Query strings are evaluated right to left (right associativity) with the left-most operand indicating the object to return. Table 1 lists example queries.

Internally, BAS stores a directed graph of objects to answer queries. Figure 3 shows a partial rendering of functional and spatial graph for one of our two test buildings. Objects are exposed by drivers and can be low-level (e.g. damper, sensor, fan) or high-level (e.g. air handler, chilled water loop). Directed edges indicate the flow of air, water, electricity, etc. Edges are purposefully unnamed to allow for buildings with many different designs to be represented without introducing new names. Tags are automatically applied based on the type of driver instantiated and the interface it implements. Tags describe the object type and functionality. They are intentionally low-level and are not meant to uniquely identify objects. The intent is to use relationships to select objects in a general way. For example, a supply air temperature sensor can be selected as `#TEMP < #COOL < $Air Handler 1`, that is, the temperature sensor down stream of the cooling element in Air Handler 1. This allows the application to run with both a standard temperature sensor built into the air handler as well as a less common building design with a temperature sensor installed in the duct work or at a VAV input. A partial list of tags is show in Table 2.

In addition to functional relationships, BAS also stores spatial data. Spatial areas are defined as polygons on floor maps of the building and stored in a GIS database [15]. Areas are defined for the regions served by a given duct or VAV box and for lighting zones. These functional areas do not always align with logical spaces such as hallways, offices or the cubicles assigned to a research group. BAS allows logical areas to be defined for use in the query system. By default the query system treats all intersecting areas as logically linked. If a region named `$BAS Cubicle` was defined, then the query `#LIGHT > $BAS Cubicle` would return all light zones that overlap with the BAS cubicle area. Spatial areas can be easily added or modified by updating the underlaying GIS database.

Table 2. Partial list of object tags. Tags describe the types of primitive objects. Queries use tags and functional or spatial relationships to find objects.

#ELEC	All objects in electrical domain
#HVAC	All objects in the HVAC domain
#AREA	All areas in the spatial domain
#LIGHT	All objects in lighting domain
#FLOOR	All floors
#SEN	All sensors
#ACT	All actuators
#RELAY	Relays
#DMP	Dampers
#VLV	Valves
#ELECMETER	Electrical meters
#AH	Air handlers

Queries are evaluated with a graph traversal algorithm. Starting at the right-most operand, all objects matching the tags or object names are added to the search list. Next, we search all incoming edges or all outgoing edges based on the operator (< or >) for objects that match the left hand operand. Each search is preformed recursively in a breadth-first way. Visited objects are tracked to prevent infinite loops. Searches are limited to objects in the domains of the two operands, to search across a third domain it must be specified explicitly. For example, to find lights for zones served by Air Handler 1 the query is `#LIGHT > #AREA < $Air Handler 1`; `#AREA` must be specified explicitly to search through the spatial domain. Intersecting spatial areas are treated as bidirectionally linked for the purposes of query execution.

4 Drivers

In order to provide a fuzzily-searchable query interface for device discovery, measurement and actuation, BAS must be able to provide a sufficiently concrete method of defining a building that does not drastically limit the portability or generalizability of any of its components. BAS exploits the high-level functional similarities between buildings to define a set of high and low-level driver interfaces that encompass the range of possible internal components in a given building. The goal of this structure is to facilitate the adherence of a building to a common, exposed API on top of which portable applications can be built.

BAS driver interfaces define common APIs for high-level objects whose functionality depends on and consists of a certain set of lower-level components, e.g. a certain type of air handler may contain an outside air damper, a set of temperature sensors, a cooling coil valve, etc, but it retains a level of functional congruity with different types of air handlers that may contain different components. The low-level BAS driver interfaces define these fundamental components in terms of their basic type: sensors, dampers, valves, fans and so forth.

Each of these interfaces is then implemented by one or more classes whose code handles the combination of device-specific functionality to provide the logic for the interface. The ability of the drivers to support multiple classes for a given interface means that BAS remains extensible enough to

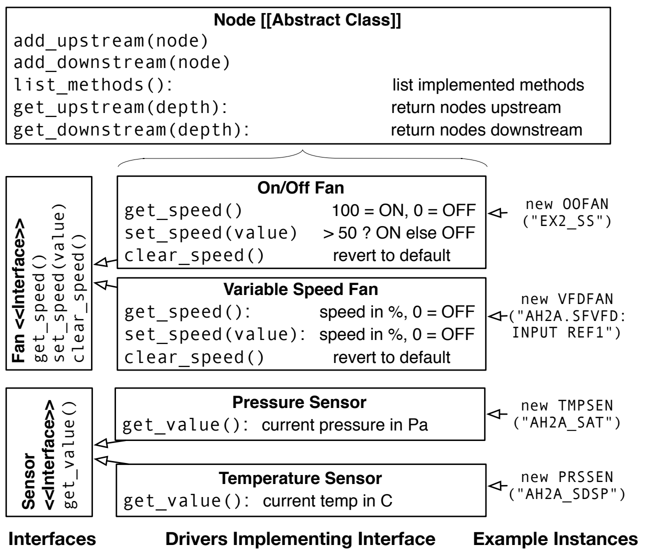
Node [[Abstract Class]]
add_upstream(node)
add_downstream(node)
list_methods(): list implemented methods
get_upstream(depth): return nodes upstream
get_downstream(depth): return nodes downstream
Fan <<Interface>>
get_speed()
set_speed(value)
clear_speed()
On/Off Fan
get_speed() 100 = ON, 0 = OFF
set_speed(value) > 50 ? ON else OFF
clear_speed() revert to default
new OOFAN ("EX2_SS")
Variable Speed Fan
get_speed(): speed in %, 0 = OFF
set_speed(value): speed in %, 0 = OFF
clear_speed() revert to default
new VFDFAN ("AH2A.SFVFD: INPUT REF1")
Sensor <<Interface>>
get_value()
Pressure Sensor
get_value(): current pressure in Pa
new TMPSEN ("AH2A_SAT")
Temperature Sensor
get_value(): current temp in C
new PRSSEN ("AH2A_SDSP")
Interfaces
Drivers Implementing Interface
Example Instances

Figure 3. Interfaces for lower-level devices specify methods for general equipment types to be implemented in equipment-specific classes. Classes are instantiated with references to the actual hardware.

integrate any custom interface for any esoterically-designed building component – provided that the logic exposes the expected API – as well as promoting reusability and portability. As seen in Figure 4, each of the driver implementation classes contains custom logic that adapts the specifics of a type of equipment to the expected interface.

For the high-level drivers, the interface specifies the expected components in terms of descriptive tags; any instance of that driver must provide instances of the requisite low-level driver classes to populate that instance's functionality. These descriptive tags distinguish between the functional roles of the objects and classify them in a manner that enables the query interface to discover and traverse them in terms of their functional relationships to other objects. Tags are predefined and documented by BAS to ensure portability. Objects are described by lists of these tags, joined by underscores (see example below). Basic functionality for these objects is provided by the Node abstract class which endows objects with an awareness of the network graph, granting them the ability to establish themselves in relation to objects around them.

Once the objects for a building are fully populated and instantiated, BAS creates the network graph for the purpose of the query interface. The high-level objects use their sets of expected lower-level objects in order to pre-construct internal network graphs, which are incorporated into the comprehensive graph by linking lower-level objects to each other by calling `object.add_upstream(target_object)` and `object.add_downstream(target_object)`.

4.1 Example: Simplified Air Handler

In this simplified case, the type of air handlers found in the BACnet-based building contain an outside air damper, a mixed air temperature sensor, a cooling coil valve, a supply fan and a supply air temperature sensor. In order to create an

```
#instantiate air handler object
ahu1 = ahu('Air Handler 1', {
    'OUT_AIR_DMP': Damper('SDH.PXCM-01 SDH.AH1A_OAD'),
    'MIX_AIR_TMP_SEN': TmpSen('SDH.PXCM-08 SDH.AH1A_MAT'),
    'COOL_VLV': Valve('SDH.PXCM-01 SDH.AH1A_CCV'),
    'SUP_FAN': VfdFan('SDH.PXCM-01 SDH.AH1A.SF_VFD'),
    'SUP_AIR_TMP_SEN': TmpSen('SDH.PXCM-01 SDH.AH1A_SAT')
    } )
#link air handler object into the network graph
ahu1['COOL_VLV'].add_upstream(cold_water_loop['PUMP'])
ahu1['SUP_FAN'].add_downstream(vav1['DMP'])
ahu1['SUP_FAN'].add_downstream(vav2['DMP'])
...
```

Figure 4. Instantiating a simplified air handler

instance of this type of air handler, it is first necessary to create instances of these expected components. Once these are provided in the instantiation of the air handler, the particular BAS air handler class knows how to incorporate the API for each component into the expected logic for the general air handler API.

BAS defines low-level driver interfaces for dampers, sensors, valves and fans, and the BAS implementation for the building contains interface-compliant classes that provide the necessary BACnet-specific logic for reading from and writing to the necessary points for each of these drivers. The exact point names for each object are provided upon instantiation of that object.

5 Interconnect Abstraction

The interconnect layer provides a RESTful interface that routes sequences of read and write requests to the underlying building control protocols. It addresses three main challenges in processing requests from the driver and application levels: interfacing with different building control protocols, handling network throughput, and maintaining a consistent building state. By using sMAP [5] to provide this interface, BAS is able to implicitly handle distributed setup as well as data archiving.

A building may contain any one of a number of building control protocols – such as BACnet, OPC, LONtalk, ALC SOAP, and ModBus – each of which may define or provide their own communication protocol, software library, etc. for the purpose of reading and writing to a set of points corresponding to aspects of the building. It is the job of the interconnect layer to handle the implementation details to sufficiently abstract the reading and writing of points for the driver and application levels. For example, when interfacing with BACnet, the interconnect layer helpfully removes the need to know the unusual or specific numeric constants required to correctly formulate a BACnet packet.

The devices inside buildings often communicate over a specialized physical layer that may not be immediately compatible with TCP/IP. A building's control protocol will usually provide some mechanism for communicating with these devices, but the implied intricacies of such a mechanism are often too complex to be accounted for at a higher level in BAS. By handling the construction of packets and managing network throughput at the interconnect layer, the driver layer does not require extensive knowledge of various network protocols.

Furthermore, the interconnect layer handles all end-to-end reliability on behalf of the drivers. The utility of the higher abstraction layers is made with the guarantee of a consistent building state, so despite the multi-tiered indirection between an application and its points of actuation, the end-view of a building will be correct.

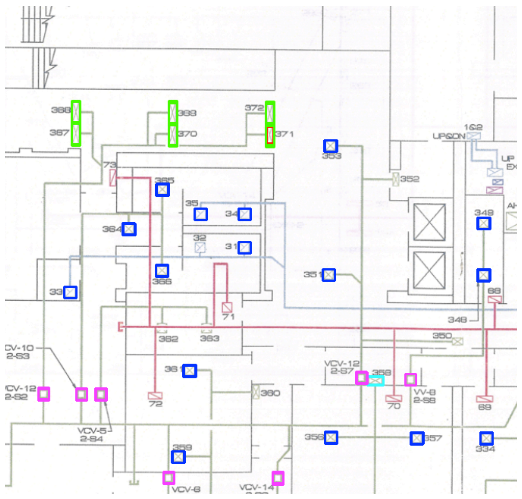

Figure 5. Example of parsing an image to identify air handlers, VAV boxes, and dampers

6 Autopopulation

In order to port applications between buildings, BAS must be correctly configured for each building, requiring some amount of work to be done to catalog or discover exactly what devices are in the building, how they are logically and functionally connected, and how these logical pieces affect the spatial aspects of the building. Automating this process is an area of ongoing work. Our initial approach is to use existing interfaces for object discovery combined with drivers we wrote for different building components and an image recognition technique for importing functional and spatial relationships from existing documents.

Most building control protocols provide some method for device discovery. This usually takes the form of providing a list of all devices that can be read from and/or written to over the internal network. Unfortunately, for some protocols such as BACnet, device discovery takes the form of a `whois` packet broadcast over the local subnet, meaning that in the absence of having a machine on the building subnet, the discovery method is unable to return any results. It is possible, however, to construct packets that take into account the internal structure of the building network, and thus conduct a device discovery query remotely. We scan a given IP for valid gateway devices and then search for internal sub-networks by examining the error codes returned by BACnet.

Once we have a list of devices on the internal building network, we can use the building's naming schema to identify objects by type, name, and location. The reality is that most building control protocols do not enforce the inclusion of much of this metadata, leading to incomplete and sometimes inconsistent catalogs of building devices. Despite the wide variability in the quality of existing building metadata, BAS can automatically construct at least a partial representation of the building, drastically reducing the overhead for initializing a building, depending on the structure of the internal network and the amount of metadata provided.

In some cases it is possible to extract data from the existing building management system to construct the functional and spatial relationships. For example, the ALC WebCTRL [2] automation system provides a SOAP interface that contains an internal geographically organized tree of all the building's devices, helpfully combining the processes of discovery and geo-tagging.

Other representations of the building such as IFCs [11] or EnergyPlus models [8] could be used to automatically load building metadata. However, these are not often available. As a fallback method we use computer vision techniques on architectural diagrams to recognize various types of objects on a floor plan, noting their location and relationship.

The example in Figure 5 uses only a few lines of the OpenCV Python library to extract VAV objects from a ductwork diagram by nature of their shape using the built-in template matching functionality. Combining this with OCR, it is possible to extract the geospatial coordinates, names and functional relationship of VAVs.

7 Applications and Evaluation

The power of BAS lies in its ability to be easily adapted from one building to another, bridging the common functionality between different hardware implementations, communication protocols, and building control systems. Here, we will examine the deployment of two applications on two buildings with different control systems, implemented with and without BAS.

The first building is approximately 100k sq. feet and has 7 floors. It contains large open cubicle areas for graduate students and private offices for faculty and administrators. It also contains a cafe, auditorium and several classrooms. It runs a Siemens control system with over 8000 available sensor and actuator points exposed over BACnet. The second building is approximately 75k sq. feet in size and has 6 floors with office areas and a library area. The control system is ALC, which supports both low-level BACnet access and a higher-level SOAP interface, exposing over 5000 points.

7.1 Occupant HVAC Control App

The occupant HVAC control application allows occupants to temporarily blow hot or cold air into a room. The logic is simple. For a given room, the corresponding damper and, in the case of a request for heating, heating element are opened for a limited time. Then, the damper and heating defaults are restored.

Figure 7.1 shows the implementation in Python using a BACnet library. BACnet commands require cumbersome low-level arguments: device name, object instance number, property and data type; this information is not necessarily easy to access. The application must contain an explicit building-specific mapping of which dampers and heating valves control each room, along with which internal BACnet object and value type those objects are. BACnet points have cryptic names and do not contain any spatial awareness of the building itself. This code could be additionally augmented with consistency checking that verifies that the writes were received correctly by the appropriate devices, as BACnet uses UDP and thus is inherently unreliable.

If this application were to be deployed to any other building, the devices and objects would have to be rewritten to point to the new building's BACnet IP router. The instance numbers as well as the property, value and object types would have to be double checked to ensure correctness, and the point names would have to be changed.

Figure 7.1 shows the functionally equivalent application written using BAS. The increased readability is immediately obvious. BAS encodes spatial relationships, so finding VAVs corresponding to a space is trivial. BAS drivers expose convenient methods for setting airflow and heating in a standard way that applies to all VAVs. The application makes no reference to the specific communication protocol nor to the specific point names. This app could be ported to a completely different building and run without further configuration of

```
#Using direct BACnet
import bacnet
import time
dampers = {
            'Room 444': ('SDH.PXCM-11','SDH.S4-03:DMPR POS', 'SDH.S4-03:VLV POS'),
            'Room 446': ('SDH.PXCM-11','SDH.S4-05:DMPR POS', 'SDH.S4-05:VLV POS'),
            'Room 448': ('SDH.PXCM-11','SDH.S4-09:DMPR POS', 'SDH.S4-09:VLV POS'),
            ...,
            }
def cool_room(room_number):
  damper = dampers[room_number]
  device = bacnet.find(name=damper[0]) #bacnet.find('SDH.PXCM-11')
  object = bacnet.find(name=damper[1]) #bacnet.find('SDH.S4-05:DMPR POS')
  bacnet.write_prop(device, object_type=bacnet.OBJECT_ANALOG_OUTPUT, \
          instance_number=object.instance_number, property=bacnet.PROP_PRESENT_VALUE, \
          value=100, value_type=bacnet.BACNET_APPLICATION_TAG_REAL)
  time.sleep(1000)
  bacnet.write_prop(device, object_type=bacnet.OBJECT_ANALOG_OUTPUT, \
          instance_number=object.instance_number, property=bacnet.PROP_PRESENT_VALUE, \
          value=1, value_type=bacnet.BACNET_APPLICATION_TAG_NULL)

def warm_room(room_number):
  damper = dampers[room_number]
  device = bacnet.find(name=damper[0]) #bacnet.find('SDH.PXCM-11')
  object = bacnet.find(name=damper[1]) #bacnet.find('SDH.S4-05:DMPR POS')
  heating = bacnet.find(name=damper[2]) #bacnet.find('SDH.S4-05:VLV POS')
  bacnet.write_prop(device, object_type=bacnet.OBJECT_ANALOG_OUTPUT, \
          instance_number=heating.instance_number, property=bacnet.PROP_PRESENT_VALUE, \
          value=100, value_type=bacnet.BACNET_APPLICATION_TAG_REAL)
  time.sleep(60)
  bacnet.write_prop(device, object_type=bacnet.OBJECT_ANALOG_OUTPUT, \
          instance_number=object.instance_number, property=bacnet.PROP_PRESENT_VALUE, \
          value=100, value_type=bacnet.BACNET_APPLICATION_TAG_REAL)
  time.sleep(1000)
  bacnet.write_prop(device, object_type=bacnet.OBJECT_ANALOG_OUTPUT, \
          instance_number=object.instance_number, property=bacnet.PROP_PRESENT_VALUE, \
          value=1, value_type=bacnet.BACNET_APPLICATION_TAG_NULL)
  bacnet.write_prop(device, object_type=bacnet.OBJECT_ANALOG_OUTPUT, \
          instance_number=heating.instance_number, property=bacnet.PROP_PRESENT_VALUE, \
          value=1, value_type=bacnet.BACNET_APPLICATION_TAG_NULL)
```

Figure 6. BACnet-specific occupant controlled HVAC application code.

```
#Using BAS
import appstack
import time
api = appstack.Appstack()

def cool_room(room_number):
  vav = api('#VAV > $%s' % room_number)
  vav.set_airflow(100)
  time.sleep(1000)
  vav.clear_airflow()

def warm_room(room_number):
  vav = api('#VAV > $%s' % room_number)
  if 'set_heat' in vav.list_methods():
    vav.set_heat(100)
    time.sleep(60)
    vav.set_airflow(100)
    time.sleep(1000)
    vav.clear_heat()
    vav.clear_airflow()
```

Figure 7. BAS implementation of occupant HVAC controls.

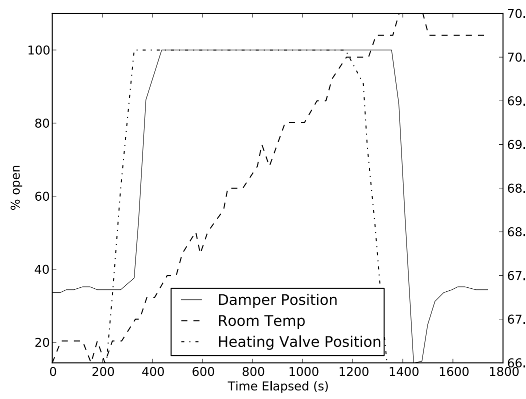

Figure 8. Trace of the occupant HVAC control app running on one of our buildings. The heating coil and airflow increase to maximum, rapidly warming the room.

the application itself, provided that BAS is correctly installed on the given building. Figure 7.1 shows this application running on our building with Siemens controls. The app warms an enclosed office area in response to an occupant request.

7.2 Ventilation Optimization App

The ventilation optimization application is based on the observation that many buildings are over-ventilated; California Title 24 [4] requires 15 CFM of *fresh air* per person in the building, but the fraction of fresh air taken into the building varies from 30% to 100% of the supply air depending on the economizer (outside air damper) position. As a result, airflow rates are often set assuming minimum fresh air intake. The ventilation optimization app adjusts the minimum airflow rates for all VAV boxes in the building based on the fraction of fresh air their corresponding air handler brings in.

It is immediately evident in the writing of this application that some knowledge of the layout of the HVAC system is required in order to adjust each VAV by the correct amount. Figures 7.2 and 7.2 show this application implemented on our Siemens and ALC buildings respectively. Note the hard-coding of associations between VAVs and air handlers, the cryptic point names and the control protocol specific actuation code. Porting the application from one building to another required a complete rewrite of the read/write logic, not just the point names.

The same application implemented using BAS is shown in 10. Using relative queries simplifies the code by dynamically finding the corresponding economizers, VAV boxes and dampers; thus, porting the application from our first building to the second required no change in the actual code.

7.3 Porting BAS

In configuring BAS to run on two buildings as different as the BACnet-based and ALC-based buildings mentioned above, there is a surprisingly small amount of work to be done considering the differences between them. The most accessible programmatic interface to the first is through BACnet; the particular implementation allows for direct access to many individual components within the building, but lacks much of the location-aware metadata that is useful to BAS. Conversely, the ALC-based building provides a

```
#Using direct BACnet
import bacnet
#damper setpoints for each outside air damper
oad_to_dmp_stpts = {
  'SDH.PXCM-01 SDH.AH1A_OAD': [
    'SDH.PXCM-04 SDH.S1-20:CTL FLOW MIN',
    'SDH.PXCM-04 SDH.S1-19:CTL FLOW MIN',
    ...],
  'SDH.PXCM-01 SDH.AH1B_OAD': [
    'SDH.PXCM-11 SDH S2-04:CTL FLOW MIN',
    ...],
  'SDH.PXCM-08 SDH.AH2A_OAD':[
    'SDH.PXCM-11 SDH.S4-03:CTL FLOW MIN',
    ...]
  }
for oad in oad_to_dmp_stpts.keys():
  device = bacnet.find(name=oad)
  oad_airflow = bacnet.read_prop(device, object_type=bacnet.OBJECT_ANALOG_OUTPUT, \
    instance_number=device.instance_number, property=bacnet.PROP_PRESENT_VALUE)
  for dmp in oad_to_dmp_stpts[oad]:
    damper = bacnet.find(name=dmp)
    old_setpoint = bacnet.read_prop(device, object_type=bacnet.OBJECT_ANALOG_OUTPUT, \
      instance_number=damper.instance_number, property=bacnet.PROP_PRESENT_VALUE)
    new_setpoint = old_setpoint / oad_airflow
    bacnet.write_prop(device, object_type=bacnet.OBJECT_ANALOG_OUTPUT,\
      instance_number=damper.instance_number, property=bacnet.PROP_PRESENT_VALUE, \
      value=new_setpoint, value_type=bacnet.BACNET_APPLICATION_TAG_REAL)
```

Figure 9. Ventilation optimization app implemented on our first building using BACnet.

```
#Using ALC SOAP
import suds
url_to_wsdl = 'http://......'
client = suds.client.Client(url)
air_handlers = {
  '#doe_basement/#doe_base_equipment/#doe_ah-c': [
    '#doe_vav_c-2-13/air_flow/flow_tab',
    '#doe_vav_c-2-14/air_flow/flow_tab',
    ...]
  '#doe_penthouse/#doe_ah-b1_ah-b2_rf-1': [
    '#doe_vav_b-5-01/air_flow_b1/flow_tab',
    ...]
  }
for ahu in air_handlers.keys():
  oad_airflow = client.getValue(ahu+'/oa_damper')
  for vav in air_handlers[ahu]:
    old_damper_airflow = client.getValue(vav+'/m269')
    new_damper_airflow = old_damper_airflow / oad_airflow
    client.setValue(vav+'/m269', new_damper_airflow)
```

Figure 10. Ventilation optimization app implemented in BAS and executable on both buildings.

```
#Using BAS
import appstack
api = appstack.Appstack()
ah_dampers = api('#OUT_AIR_DMP > #AH')
for dmp in ah_dampers:
  for vav in api('#VAV < $%s' % dmp.name):
    vav.set_min_airflow(vav.min_fresh_air() / dmp.get_percent_open())
```

Figure 11. Ventilation optimization app implemented on our second building using the ALC SOAP interface.

SOAP (Simple Object Access Protocol) interface over BACnet, which allows BAS to glean a considerable degree of location-aware metadata, but does not allow as fine-grained control over individual devices.

In practice, configuring BAS to run the application in Figure 10 above required only some adjustment of the damper and heating valve drivers to communicate with the specific control protocol of the building in question, which was greatly simplified by the abstractions already provided by the interconnect layer (about 5 LOC per driver). While it could be argued that the BACnet specific code in Figure 7.2 could be configured to run on a different (yet still BACnet controlled) building with changes on the order of 5 LOC, this configuration would have to be done on an application-by-application basis, whereas BAS only requires configuration once per building. The hierarchical nature of BAS drivers means that only low-level drivers (e.g. damper, valve, fan, etc.) need to be ported, while high-level drivers (e.g. air handler) generally remain unchanged.

8 Related Work

Several approaches have been proposed for organizing building metadata. Tree structures are most commonly used today [2, 6]. In this approach, related sensor and actuator points are grouped hierarchically. For example, in one of our buildings HVAC equipment is grouped by location and function: `/basement/hot_water_plant/boiler/steam_flow` and power meters are grouped by panel and breaker: `/main_breaker/panel_41/floor4_lighting/real_power`. Hierarchies provide a single, "precomputed" way to explore the data. This works well for answering some queries, e.g. "what loads are on panel 41," but cannot handle more complex queries, e.g. "which circuits power room 123's outlets and lights."

The next approach is to use predefined classes or entity-relationship models for common building components. Industry Foundation Classes [11] specify models for structural, mechanical and electrical aspects of buildings. IFCs are intended to describe building design and facilitate sharing of information among design and construction teams. IFC includes classes for HVAC equipment and a connectivity model for building a directed graph of objects [3]. BAS models the active sensor and actuator components, while IFC focuses on the design specification without linking this to the running controls. BAS uses a similar connectivity model, but eliminates the need for rigid predefined classes by using tags and a driver model that only requires implementing basic read/actuate methods for each driver type.

Project Haystack [16] uses a list of tags and rules about their use to describe building components. The tagging approach is very flexible and overcomes the rigid structure of fixed object classes or hierarchies. However, tagging schemes like Haystack cannot encode the full range of functional and spatial relationships. Queries are limited to the relationships that are manually tagged. In BAS, relationships are queried by traversing a graph structure.

All of these efforts focus on describing building data, while BAS crucially provides methods for actively controlling the building and raises the level of abstraction to allow control to be implemented portably. We share the same vision as [13] of enabling building applications. [13] focuses on integrating existing metadata from multiple sources and devising a common data representation for use by applications. BAS is complementary and is focused on how applications can conveniently make use of available building controls portably and at a higher level of abstraction (driver API), how to make applications automatically adjust to different building designs (fuzzy queries and exploration) and how to execute applications on a new building (interconnect and runtime platform).

9 Conclusion

Modern commercial buildings contain some of largest deployed sensor networks, often consisting of thousands of sensors and actuators. This infrastructure has the potential to enable a wealth of applications that change the way we interact with buildings, reduce energy consumption, support grid operation, improve reliability and maintenance.

The main challenge is to make applications portable and easy to develop. All buildings are designed differently and building controls are inconsistent in naming, function and available features. Today this requires developers to have detailed knowledge of each building's architecture, HVAC design, control network and hardware functionality.

BAS abstracts these details, allowing application authors to select building components with fuzzy queries and use standardized methods built up through a hierarchy of drivers to read or actuate the building. BAS queries are key to portability; they allow authors to select components in terms of functional or spatial relationships that are implicitly portable.

We implement two different control applications, a ventilation optimization app and an occupant temperature control app, on two different large buildings on campus. Each app is implemented first using the raw building control protocol and then again using BAS. We show that the BAS code is much shorter, easier to understand and does not change for each building while preserving the same execution.

10 Acknowledgments

We thank Domenico Caramagno and Venzi Nikiforov for their support in deploying our system and their invaluable insight into building operation. We thank Stephen Dawson-Haggerty for many great discussions on the design and use of BAS. We also thank the reviewers for their helpful comments. This work is supported in part by the National Science Foundation under grants CPS-0932209 and CPS-0931843.

11 References

[1] ASHRAE. ANSI/ASHRAE standard 135-1995, BACnet, 1995.

[2] AUTOMATEDLOGIC. ALC WebCTRL, 2012.

[3] BAZJANAC, V., FORESTER, J., HAVES, P., SUCIC, D., AND XU, P. HVAC component data modeling using industry foundation classes. In *System Simulation in Buildings* (2002).

[4] CA ENERGY COMMISSION. California's energy efficiency standards for residential and nonresidential buildings, 2008.

[5] DAWSON-HAGGERTY, S., JIANG, X., TOLLE, G., ORTIZ, J., AND CULLER, D. smap: a simple measurement and actuation profile for physical information. In *SenSys '10* (2010).

[6] DICKERSON, R., LU, J., LU, J., AND WHITEHOUSE, K. Stream feeds: an abstraction for the world wide sensor web. In *IOT'08* (2008).

[7] ECHELON CORPORATION. LonTalk protocol specification, 1994.

[8] EERE. EnergyPlus. http://energyplus.gov, 1998.

[9] ENERGY INFORMATION ADMINISTRATION. Commercial buildings energy consumption survey, 1999.

[10] ERICKSON, V. L., CARREIRA-PERPIN, M. ., AND E.CERPA, A. OBSERVE: occupancy-based system for efficient reduction of hvac energy. In *IPSN'11* (2011).

[11] ISO. Industry Foundation Classes, Release 2x, 2005.

[12] KRIOUKOV, A., DAWSON-HAGGERTY, S., LEE, L., REHMANE, O., AND CULLER, D. A living laboratory study in personalized automated lighting controls. In *BuildSys'11* (2011).

[13] LIU, X., AKINCI, B., GARRETT, J. H., AND BERGES, M. Requirements for a formal approach to represent information exchange requirements of a self-managing framework for HVAC systems. In *ICCCBE* (July 2012).

[14] OPC TASK FORCE. OPC common definitions and interfaces, 1998.

[15] PostGIS. http://www.postgis.org/.

[16] Project haystack. http://project-haystack.org/.

[17] SCHEIN, J., BUSHBY, S. T., CASTRO, N. S., AND HOUSE, J. M. A rule-based fault detection method for air handling units. In *Energy and Buildings* (2006).

[18] U.S. DEPARTMENT OF ENERGY. 2011 buildings energy data book, 2012.