A Network for Customizable + Reconfigurable Housing

Kent Larson
Principal Research Scientist
House_n
Director, Changing Places
MIT Department of
Architecture and Planning
MIT Media Laboratory
kll@media.mit.edu

Tyson Lawrence
Research Assistant
House_n
MIT Department of
Mechanical Engineering
tyson_l@mit.edu

Thomas J. McLeish
Research Assistant
House_n
MIT Media Laboratory
tmcleish@media.mit.edu

Deva Seetharam
Research Assistant
Physics and Media group
MIT Media Laboratory
deva@media.mit.edu

H. Shrikumar
Research Scientist
Physics and Media group
MIT Media Laboratory
shri@media.mit.edu [*][†]

Abstract

The housing industry of today produces homes that are poorly prepared for the future.

Rapidly changing demographics and related societal pressures will inevitably transform the home into a center for preventative health care, distributed energy production, work, learning, and communication - requiring us to rethink how we design and integrate technology into our places of living. In addition, people have a powerful desire for their places of living to reflect individual needs and values.

The current housing development process, however, discourages innovation and produces mostly low-grade, generic commodities. It cannot efficiency respond to the unique requirements of individual occupants.

We propose a strategy that may enable customized, cost-effective, high performance housing solutions. In this paper, we present essential concepts for a building network that is necessary to fully realize these goals.

We present the initial implementations of building network called GSG 1.0 and c@t, a language for programming distributed embedded systems such as large-scale building networks.

[*]Authors names are listed in alphabetical order.

[†]This work was done by Deva Seetharam and H. Shrikumar while they were at the MIT Media Laboratory.

1 Introduction

A residential developer in Cambridge, Massachusetts recently said, "I would love to provide customized solutions - we could demand higher prices - but customization results in too much brain damage to be worth it." There are good reasons for this assessment.

The design process for a typical housing development project can take many years. Long before an individual occupant becomes involved, governmental permitting and construction bidding requires that key decisions be made about unit layouts, fire alarm locations, electrical outlets, wheel chair access, etc. With no individuals to design for, developers strive to hit the center of their market. When a buyer finally enters the picture, their "customization" choices are necessarily limited to a few largely superficial alternatives such as floor finishes and counter tops. More fundamental customization, such as room size, layout, or electrical access is essentially reconfiguration of a finished building requiring time, complex coordination, and disruptive operations.

To make customization practical, we propose the decoupling of the base building design, approval, and construction process from the customization of individual apartments at the time of sale. Implementation will involve the following:

1. Chassis + infill. General Motors has proposed a standard "Hywire" chassis across all of their models plus the mass-customization of body components, interiors, and electronics. Similarly, we propose that buildings consist of an efficiently constructed building "chassis" that integrates structure, power, signal, and other infrastructure plus customized interior components that allow the occupant to tailor their interiors according to their budget, needs, and values [4]. This strategy will permit the cost effective reconfiguration of interior components. This builds on "Open Building" concepts first developed by John Habrakan over 30 years ago in the MIT Department of Architecture [6].

2. Integrated sensing. The automotive, aviation, and shipbuilding industries are moving towards integrated modules supplied by Tier 1 suppliers. Similarly, we propose that interior components for housing be integrated with a variety of embedded technologies that are efficiently installed in the controlled environment of the factory. This will support, for example, the development of affordable proactive health care environments containing hundreds of low cost sensors, smoothing the transition of care from the clinic to the home.

3. Design customization for non-experts. Customization is only workable in production housing if computational tools assist in the design process and configuration since the expense of a skilled specialist is prohibitive. Work is currently underway at MIT and elsewhere that may be available to industry in the near future [7].

4. Novel building networks. Customizable and reconfigurable interior components will require new approaches to building networks. These networks must allow control connections to be software bindings rather than hardwired links, in order to allow physical components to be easily moved and reconfigured. Also, ubiquitous sensing for health care, energy management, and other technology solutions must be easily configurable, even by non-experts.

It is important to note that we have not yet built a building using this strategy. We are studying the concepts and techniques using models and other test implementations. In this paper, we present our initial ideas, current implementation and early observations.

The rest of this paper is organized as follows. We describe the architectural concepts and components in section 2 and the building control networks in section 3, current implementations in sections 4 and 5,related work in section 6 and we conclude with discussions and our plans for future work.

2 Background

The chassis+infill system mentioned in Section 1 is fundamental to providing customized, reconfigurable homes. Here is a detailed description of how a building can be constructed using chassis and infill.

The chassis is made up of beams and columns. Columns are vertical and beams are horizontal. As shown in Figure 1, the chassis provides heating, cooling and ventilation with an air duct, electrical and data wiring with raceways and an electrical track, while also providing the building structure.

The infill is the walls, floors, and ceilings of the home. The infill is the customizable part of the system and allows the user to customize the interior fit-out and exterior facade. In addition, infill components may be easily removed after construction to facilitate reconfiguration.

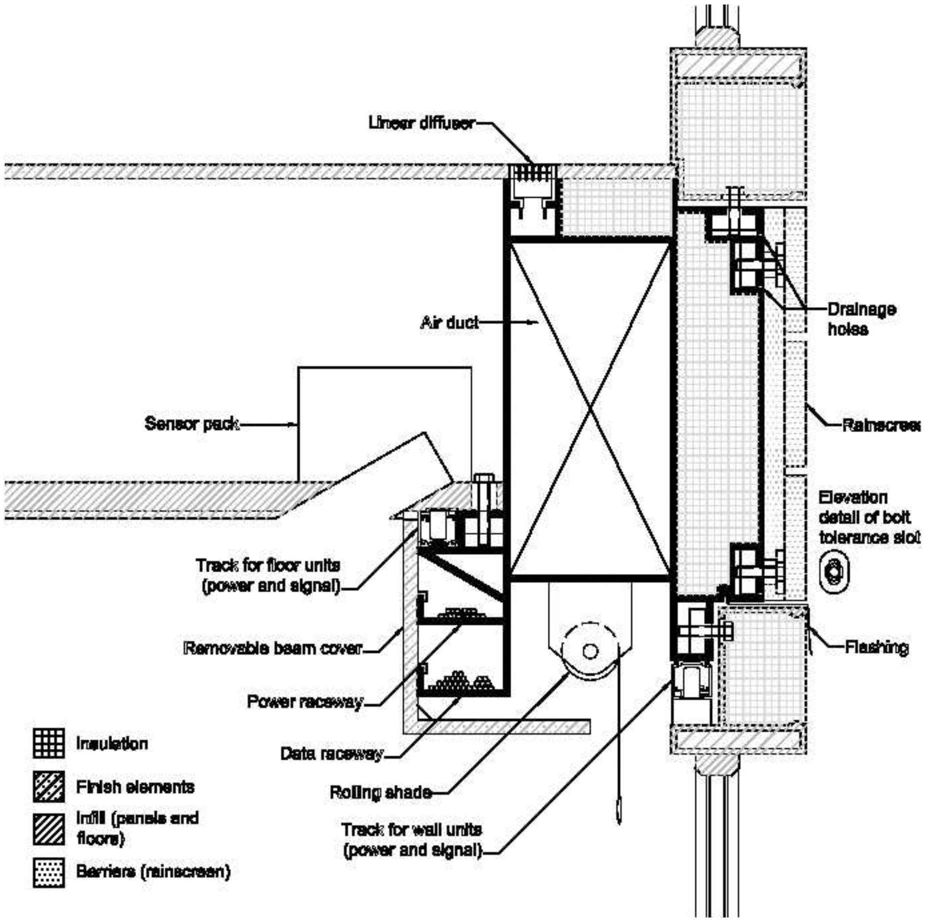


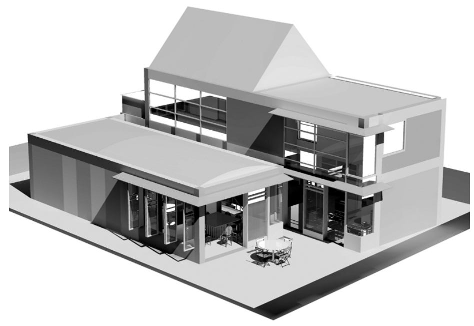
Figure 1: A detailed view of beams.

Figure 2 illustrates the steps involved in constructing a house using chassis and infill. Imagine that the house shown in Figure 2(a) is to be built (the floor plan is shown in Figure 2(b)). Figure 2(c) shows the construction sequence: First, the foundation with the first story columns partially installed. These columns are the beginnings of the chassis. Second, most of the chassis have been finished as first story beams and columns have been installed. Third, the chassis is completed and some infill wall and floor components have been installed. Fourth, shows the house almost complete. Infill roofing components have been added as well as the second story floor. All that remains to be installed are the final infill wall components.

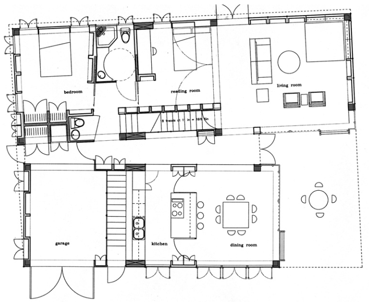
In essence, chassis and infill components can be "plugged" together to build a house. While the house is assembled, not only the structure emerges, but also completely connected data and power networks and a HVAC distribution system.

Using the chassis+infill approach it might be possible to easily construct and customize houses. Furthermore, the houses could then be easily reconfigured since most of the infill wall panels can be moved as they don't provide any structural support to the building. For instance, if there are two rooms defined by columns (a, b, c, d) and (c, d, e, f), the two rooms could be merged to form one single room (a, b, e, f) by removing the wall panel joining the corners c and d.

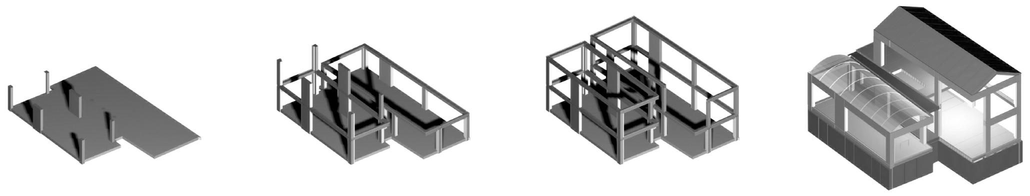
Conventional hardwired electrical connections are not suitable for these dynamically reconfigurable buildings, as any change to the physical structure would require affected portions of the building to be rewired, probably a complex task. Conventional wiring is labor intensive and expensive, installing the system during construction requires scheduling with framing, drywall, and finishing. Modifying the system after construction is complete is



(a) House



(b) Floor Plan



(c) Steps in Construction

Figure 2: An illustration of constructing a house using chassis and infill components.

even more expensive as it might require specialized labor and rebuilding of compromised finishes.

We need novel wiring schemes to support reconfigurable buildings. The next section presents these schemes and the enabling building control networks.

3 Embedded Building Control Networks

We need more "flexible" electrical networks; we are devising wiring schemes in which the control connections between components, such as lights and light switches, are not hardwired but stored in software as rewriteable bindings.

Every building component is equipped with a microcontroller that provides the computing capacity for that component. The microcontroller finds and maintains a list of other components "connected" to the component attached to that microcontroller. The computing devices communicate with each other using the data network available in the chassis.

First, we present a few system requirements we have identified:

- **Robust** - Since building control is vital, the system must be reliable and resilient to a wide variety of failures.
- **Intuitive User Interface** - The system should have convenient and simple user interfaces.
- **Self-maintaining** - Since these networks would be used by untrained users, the system must be self-maintaining.
- **Size** - The size of devices must be small enough to be unobtrusive.
- **Inexpensive** - The circuitry and operations must not drastically increase the costs of electrical networks.

Guided by these requirements, we have designed a *S*tate-Coherence Protocol for distributed control. It consists of three layers, named *G*et, *S*et, and *G*o, running on top of a diminutive TCP/IP network. In the bottom-up order, they are as follows:

1. **GET** - An idempotent transaction protocol, which ensures that control transactions occur exactly once per interaction, without race conditions, over-writes or data corruption due to distributed actions.

2. **SET** - A coherence protocol, which operates over **GET**, and brings together all the nodes in a distributed control or coordination context and ensures that they operate in unison, with a distributed consensus about the results.

3. **GO** - A concurrent programming language for the application layer using a distributed programming model based on the concept of program devolution. Devolution is an innovative distributed programming technique that allows the application designed to easily build complex applications involving thousands or even millions of nodes, without being burdened with the need to program and manage each node.

In the continued development of this theme, we are developing systems to evaluate our ideas and improve our implementations. The next sections describe the current implementations - GSG 1.0 and c@t.

4 GSG 1.0

To explore our ideas of storing the control connections as software bindings, we developed a building network system named GSG 1.0.

GSG 1.0 is a simple building control network that operates on GET and SET protocols. We made a few simplifications and the resulting limitations are:

- System is implemented on a tiny microcontroller - PIC 16F84. This device has very limited resources (1KB of code memory, 64 bytes of EEPROM and 68 bytes of RAM).

- Reconfiguration of bindings has to be performed manually either using an enhanced screwdriver or a desktop software tool.

- The system uses a simple transport protocol. It is a broadcast network with no routing functions and devices communicate in a round-robin as determined by a master device.

- Each device is assigned an 8-bit ID and thereby limiting the number of devices to 256. Also, a component can be connected only to eight other components at the same time.

We deliberately introduced these limitations for two reasons:

1. We chose the tiny PIC 16F84 since we are interested in studying how we can implement systems using the least computing resources. Also, authors have had previous experience in implementing complex systems on 16F84. In particular, one of the authors has already successfully implemented a HTTP server [11] using PIC 16F84.

2. We wanted to keep the lower-layer protocols and user interface simple in order to focus on our application protocols - GET and SET.

4.1 Hardware Details

For convenience, building components are classified into input and output components. Input components respond to user and environmental events. Output components perform some function in response to those events. A smart screwdriver[1] is used as the reconfiguration tool. For example, switches and sensors are input components, and lamps and fans are output components. Every input and output component is assigned a 8-bit ID. A block diagram of these components is given in Figure 3.

Every component is equipped with microcontroller (PIC 16F84) based circuitry and two status LEDs. Additionally, the input components have two input ports - *Add* and *Remove* and the output components have one output port - *Get*.

4.2 Software Details

GSG 1.0 software can be classified into two types - firmware that runs on microcontrollers attached to the electrical components and the remote administration tool.

[1]Referred to as just the screwdriver from now on.

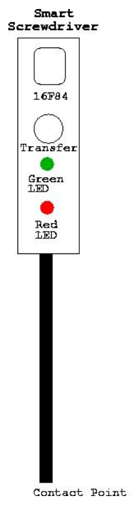


Figure 3: Block diagram of the hardware components.

4.2.1 Firmware

The primary functions of firmware are event management, connection management and communication management.

- **Connection Management**

  The screwdriver is used to make or break the connections. For example, to connect the switch (s_1) and the lamp (l_1), a user would touch the *Get* port of the l_1 (thereby pickup its ID) with the screwdriver and drop off that ID at s_1 for s_1 to add the ID to its list of subscribers.

  This module monitors the ports for contacts with the screwdriver. If a contact is made, an output component transmits it ID to the screwdriver and an input component receives a list of IDs from the screwdriver. The input component either adds or deletes the list of IDs to/from its subscriber database depending on the contact port.

  The input components maintain a database of the output components that are associated with them. As previously mentioned, the maximum number of entries in the database is limited to eight. That is, an input component can be bound to a maximum of eight output components.

- **Event Management**

  The event management functionality is implemented using the well-known publisher-subscriber paradigm. The input components observe user and environmental events and forward those events to the subscribed output components. The output components react to those events as appropriate.

- **Communication Management**

  The components are involved in three types of communications: sending/receiving IDs to/from screwdrivers, event notifications, and responding to network management messages.

  The ID exchange messages are point-to-point between the component and the screwdriver. Components exchange event management messages. The remote administration tool (explained below) sends management messages (via a gateway) to read and alter the bindings of the system.

4.2.2 Remote Administration Tool

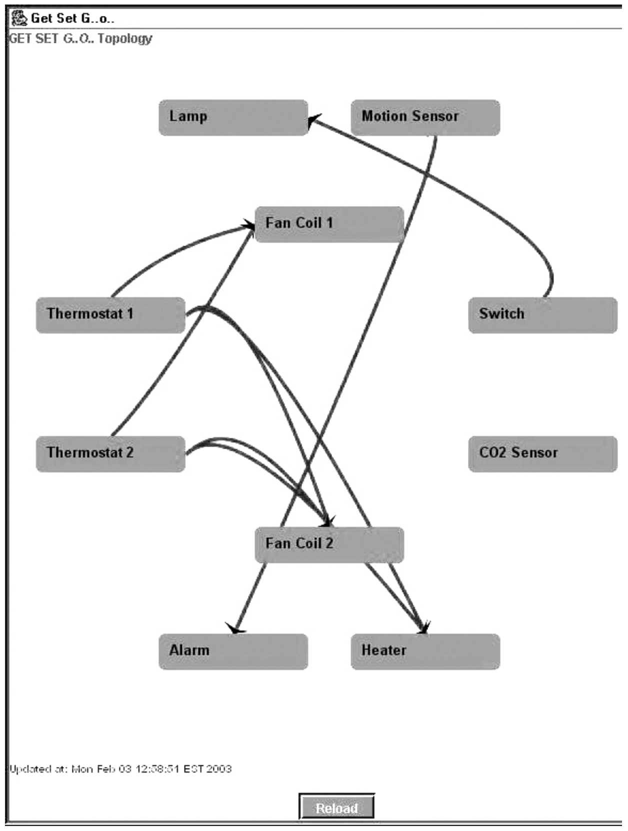
A remote administration tool called GSGViz is implemented in Java and it can be used to modify and view connections of the building network.

Every component is represented as a rectangle and the connections are represented as a line from an input component to an output component. A screenshot is shown in Figure 4.

Users can make a new connection by drawing a line from an input component to an output component or break a connection by deleting a line. GSGViz sends appropriate creation or deletion messages encapsulated in UDP packets to a gateway. The gateway translates between UDP and GSG 1.0 packets.

5 c@t

GSG 1.0 has sixteen components. We found that writing firmware for even sixteen microcontrollers using the conventional methods to be tedious. We feel that there is need a for language for programming networked embedded systems such as large-scale building networks as these systems have some unique issues that are not addressed by existing languages. The unique issues are:



Updated at: Mon Feb 03 12:58:51 EST 2003

Figure 4: GSGViz Screenshot.

- Scale complexity - A building network that supports a large building would contain extremely large numbers of different types of devices. For instance, a building could contain several types of switches, lamps, temperature sensors, motion sensors and hundreds of each of those types of devices.

- Interaction complexity - It would be necessary for many devices to cooperate to complete an application task. The computations performed by individual devices might be simple but the tasks completed collectively could be substantial. For example, neither temperature sensors nor fan controllers can complete any significant application tasks on their own. But, by cooperating, they can maintain the desired temperatures in the building.

  Such collective computations would require a large number of network communications. This is unlike conventional computers where individual nodes perform sizeable tasks and coordinate with other computers only occasionally.

- Spatial relations - Spatial and structural relations between devices form a significant part of the computations. For example, a thermostat might need to interact with all the fans/heaters in the same room. It may not be possible to specify the rooms before hand, as the rooms can be restructured on the fly.

- Resource constraints - To minimize size and cost of devices, it is important to develop systems that are efficient along multiple dimensions such as message, time, and space complexities.

H. Shrikumar [12] and Estrin et al [3] have written in detail about the issues of massively distributed embedded systems.

In an attempt to address these challenges, we are developing a novel distributed programming language called c@t.

5.1 Language Overview

Figure 5 shows a simple temperature control system written in c@t. In this application, temperature sensors monitor the ambient temperature and if the temperature is greater than or equal to seventy degrees, they invoke the function `activate` on all the fans that have more than 0.5 units of battery power.

A c@t program consists of four parts:

1. Device Declarations - Specifies the devices that are part of the system. The temperature control system consists of two types of devices - fans and sensors.

2. Cluster Declarations - Specifies the cardinality of every type of device in the system. In this example, there is one cluster called `temp-control` and it is comprised of 160 sensors and 90 fan controllers.

3. Device Set Specifications - An embedded language[2] that can be used to select a subset of devices from the set of declared devices. Device set expressions start with the `@` operator. For example, the variable `temperature` is defined on sensors using the expression `(@ (= device sensor))`.

4. Variables and Functions - As in the other high-level languages, programs can be composed using functions and variables. However, they can be defined on and referenced from multiple devices using device set specifications. For example, the function `activate` that is defined on fans is invoked from `monitor`, a function defined on sensors.

c@t employs the following techniques to enable programming networked embedded systems easily and efficiently:

- Collective programming - Multiple devices can be programmed together *"System as a unit"* approach. The language allows the programmers to view and program the system as a whole without worrying about the myriad devices and details.

  Scale complexity is alleviated using the devolution. In c@t, users program just a *small number of virtual components* which get automatically realized into potentially *a much larger number of physical components.* This is possible because building networks are usually composed of a small number of equivalent classes of devices. The equivalence could be in terms of the functions performed or their properties or their current states. For example, although the temperature control application consists of 250 individual devices, they can be classified into sensors and fans based on their roles.

  In c@t, devices can also be classified using their dynamic characteristics such as current states. For instance, the function `active` is invoked on fans, but only on those fans that have more than 0.5 units of battery power.

  The c@t compiler takes a single sequential c@t program that describes the system level behavior and produces code files for each target device that is part of the system. For this temperature control application, the compiler would produce 250 code files that would execute on 160 sensors and 90 fans.

- Associative Naming - Since the systems would be composed of equivalent sets of devices, communication would be not between individual devices, but between sets of them. To specify communications between sets of devices, Associative Naming Scheme (ANS) has been developed. This is a naming mechanism that can be used to name devices based on their static characteristics (type, role, position etc) or their dynamic characteristics (current state, variable value etc). For example, the expression `(@ (grammar relational) (filter (and (= device fan) (< hop 3))))` specifies fans that are within three hops of the sensor on which this code is being executed.

- Declarative network programming - In c@t, the interactions between devices can be implemented without writing any low-level networking code. c@t uses the paradigms of function calls and variable references to represent the interactions between devices. Further more, these

[2]It is similar to how regular expressions are embedded in languages such as Perl and Awk.

```
(declare-device sensor ((processor ''16F628'')))
(declare-device fan ((processor''18F2320'')))
(declare-cluster temp-control ((sensor 160) (fan 90)))
(define (@ (= device sensor)...) float temperature 0)
(define (@ (= device sensor)...) void (monitor)
     (if (>= temperature 70)
     (activate (@ (grammar relational)
     (filter (and (= type fan) (> battery 0.5) ))))))
(define (@ (= device fan)...) void (activate)
     (set!  RB7 #x80))
```

Figure 5: Sample Application.

transfers of control and data can be implemented without writing any low-level networking code. For instance, the function `activate` defined on fan controllers is invoked by the function `monitor` defined on sensors as seamlessly as invoking a local function.

The machine code produced by the c@t compiler is vertically integrated. That is, the c@t compiler not only produces the application code, but also every single code component that runs on devices.

Complete details of this language can be found in [10].

6 Related Work

The last few years have seen several interesting innovations in the field of networked embedded systems. New embedded computing platform such as Smart Dust (or motes) [13] and the operating system Tiny OS have been developed [5]. For GSG 1.0, we could have used Smart Dust and Tiny OS as our computing platform. Similarly, we could implement **GET** and **SET** as application layer protocols on top of JINI [14]. We didn't, because we wanted to try building the system using minimal computing resources, an application specific runtime environment, and a decentralized architecture. All the functionality of GSG 1.0 can be achieved using an X-10 [15] based system. However, X-10 require too many manual procedures to be of use in larger buildings. For that matter, even GSG 1.0 is based on manual procedures. But, GSG 1.0 is just a prototype and we are working on systems that wouldn't require any manual intervention.

The language c@t has been designed to program networked embedded systems such as building control networks. Networked embedded systems are actually a specialization of both distributed computing and parallel computing systems. Due to the common characteristics, several useful techniques can be borrowed from distributed [8, 2] and parallel computing languages [9].

Despite the similarities, it is not convenient to program distributed embedded systems using these programming languages as there are two fundamental differences:

1. Spatial relations are usually considered to be irrelevant in concurrent systems.
2. Most distributed languages are implemented on top of pre-existing network infrastructure. We believe that a delayered approach to programming embedded systems would lead to more efficient results. Please see [12] for our views on application specific network protocols.

Technically, many languages such as C, Java and processor specific assembly languages can be used to program distributed embedded systems. However, the task becomes tedious as these languages don't have the right tools, abstractions, and constructs. As Abelson et al [1] remark, a programming language is more than just a means for instructing a computer to perform tasks. The language also serves as a framework with which we organize our ideas about systems. We believe that languages specifically designed for programming networked embedded systems are necessary to program these systems conveniently and reliably.

7 Discussions and Future Work

We are in the beginning stages of developing the building networks and related tools. We know that the system needs several improvements before it can be deployed in real-life situations.

Obviously, GSG 1.0 approach is not suitable for real buildings. The seemingly simple screwdriver and GSGViz can become tedious if the number of components involved is large. Further, if the component is inaccessible, only GSGViz can be used to manage that component. We are working on systems where building components self-organize into semantically correct networks.

Figure 6 shows an architectural scale model we have built of a chassis+infill building system that contains the next generation of building network elements. In this model, devices compute their own address based on their physical location relative to a root node (cornerstone). The infrastructure provides locational cues to the components. The components use this information to establish meaningful bindings.

We are also in the process of improving the c@t language. Two important characteristics of c@t are:

1. It assumes there is no operating system available on target devices. That is, the compiler not only produces the application code, but also every single piece of code (network support, memory management, etc) that runs on those devices.

2. All the devices in the system are programmed together and their interactions are specified in a minimally constrained fashion.

We would like to take advantage of these characteristics to produce more efficient systems, as the generated code can be tailored to application needs and features.

For instance, the temperature control system presented in Section 5.1 can be realized in at least three of possible ways:

1. A centralized solution, where a powerful device (if available) is chosen as a registry and all the other devices register themselves with that registry. When the sensors need to activate a fan controller, they can search this registry to choose an appropriate device and send an activation message.

2. A completely decentralized solution. Every time there is a temperature change, the sensors could search the neighborhood for fans and notify the best one.

3. A hybrid solution, where many clusters are formed with one fan, one heater and multiple sensors. Each cluster can have its own small registry and the sensors can find a device in the neighborhood by searching this local registry.

One could object to this extensive offline analysis approach on the following grounds:

- *As the cost of microcontrollers is going down while the amount of available on-chip resources is going up, is such an extensive analysis necessary?*

  We believe this approach useful. Compiler attempts to minimize space complexity, time complexity and message complexity. Due to continuous reduction in memory costs, space complexity may not be of concern. However, response time, channel bandwidth and battery power are critical resources and they need to be used efficiently.

- *Is this approach possible? Since the number of devices in a networked embedded system such as a large-scale building networks can be extremely high, would it be possible to perform such an extensive offline analysis?*

  We don't know as we have not implemented this system. We have two primary reasons to believe that this can be possible:

  1. There are several well-known algorithm analysis and network analysis techniques that can be utilized.

  2. Workstations are becoming increasingly more powerful.

We are working on improving the various aspects of the language, protocol, algorithms, and architecture to design building networks that can be employed in real life reconfigurable buildings.

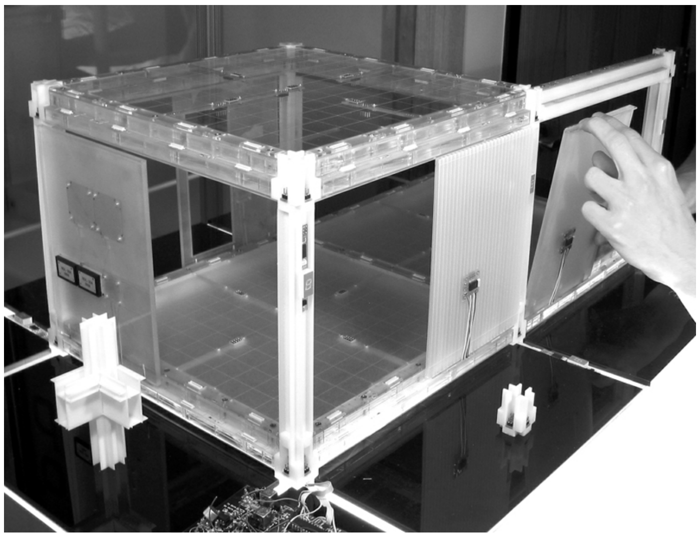


Figure 6: An architectural scale model with the next generation building network.

Acknowledgments

We would like to thank Changing Places and Things That Think Consortia of the MIT Media Lab for supporting this work. We are indebted to Prof. Neil Gershenfeld, Prof. Gerald Jay Sussman, Matt Hancher and Zoe Teegarden for inspiring and informative discussions. We are grateful to Jennifer Beaudin, Steve Form, Stephen Intille, Charles Kukla, Ran Tao, Emmanuel Tapia and other members of House_n for their constant support and encouragement.

References

[1] H. Abelson, G. Sussman, and J. Sussman. *Structure and Interpretation of Computer Programs.* MIT Press, 1996.

[2] H.E. Bal, J.G. Steiner, and A.S.Tanenbaum. Programming languages for distributed computing systems. *ACM Computing Surveys*, 21(3):261–322, September 1989.

[3] D. Estrin and et al. *Embedded, Everywhere: A Research Agenda for Networked Systems of Embedded Computers.* Computer Science and Telecommunications Board, 2001.

[4] Partnership for Advancing Technology in Housing (PATH). Whole-house and building process redesign year one progress report. Technical report, U.S. Department of Housing and Urban Development Office of Policy Development and Research, June 2002.

[5] J. Hill, R. Szewczyk, A. Woo, and D. Culler. Tinyos. http://tinyos.millennium.berkeley.edu/.

[6] K. Larson. The home of the future. *A+U*, 361, Oct. 2000.

[7] K. Larson, M. Tapia, and J. Duarte. A new epoch: Automated design tools for the mass customization of housing. *A+U*, 366, 2001.

[8] G. Leavens. Introduction to the literature on programming language design. `http://www.cs.iastate.edu/~leavens/homepage.html`, 1999.

[9] C. Leopold. *Parallel and Distributed Computing - A Survey of Models, Paradigms, and Approaches.* John Wiley & Sons, Inc., 2001.

[10] D. Seetharam. c@t: A language for programming massively distributed embedded systems. Master's thesis, MIT, Sep. 2002.

[11] H. Shrikumar. ipic - a match head sized web-server. http://www.enablery.org/iPic.html.

[12] H. Shrikumar. Data composability in myriad nets (invited talk): De-layering in billion node mobile networks. In *Second ACM international workshop on Data engineering for wireless and mobile access*, pages 43–43, 2001.

[13] Smartdust. `http://robotics.eecs.berkeley.edu/~pister/SmartDust/`.

[14] J. Waldo. The Jini architecture for network-centric computing. *Communications of the ACM*, 42(7):76–82, 1999.

[15] X-10. http://www.x10.org.