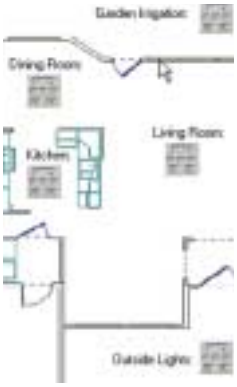
FACILITATING THE PROGRAMMING OF THE SMART HOME

JENS H. JAHNKE, MARC D'ENTREMONT, AND JOCHEN STIER, UNIVERSITY OF VICTORIA



The ongoing miniaturization and cost reduction in electronic hardware has created opportunity for equipping homes with inexpensive smart devices for controlling and automating various tasks in our daily lives. Networking technology and standards have an important role in driving this development.

ABSTRACT

The ongoing miniaturization and cost reduction in the sector of electronic hardware has created ample opportunity for equipping private households with inexpensive smart devices for controlling and automating various tasks in our daily lives. Networking technology and standards have an important role in driving this development. The omnipresence of the Internet via phone lines, TV cable, power lines, and wireless channels facilitates ubiquitous networks of smart devices that will significantly change the way we interact with home appliances. Home networking is foreseen to become one of the fastest growing markets in the area of information technology. However, interoperability and flexibility of embedded devices are key challenges for making smart home technology accessible to a broad audience. In particular, the software programs that determine the behavior of the smart home must facilitate customizability and extensibility. Unlike industrial applications, typically engineered by highly skilled programmers, control and automation programs for the smart home should be understandable by laypeople. In this article we discuss how recent technological progress in the areas of visual programming languages, component software, and connection-based programming can be applied to programming the smart home. As an example of an industrial prototype solution, we present microCommander, a visual tool for rapidly programming synergetic devices for the smart home.

PROGRAMMING CHALLENGES FOR THE SMART HOME

The ongoing miniaturization and cost reduction in the sector of electronic hardware has created ample opportunity to equip private households with inexpensive smart devices for controlling and automating various tasks in our daily lives. Networking technology and standards play an important role in driving this development. The omnipresence of the Internet via phone lines, TV cable, power lines, and wireless channels facilitates ubiquitous networks of smart devices that will significantly change the way we interact with home appliances. Home networking is expected to become one of the fastest growing markets in the area of information technology. However, interoperability and flexibility of embedded devices are key challenges for making *smart home* technology accessible to a broad audience. An increasing number of connectivity standards for net-centric smart devices have been proposed by companies and industrial consortia, such as *HAVi* (home audio-video interoperability), *JetSend* (intelligent service negotiation), *Jini*, and *Bluetooth* (proximity-based wireless networking) [1]. In particular, Bluetooth has gained a lot of attention and momentum in industry. A large number of companies led by Ericsson, Nokia, IBM, Toshiba, Intel, 3Com, Motorola, Lucent Technologies, and Microsoft have released or announced Bluetooth-enabled products, most of them in the areas of home entertainment, mobile telecommunication and personal information management.

Still, connectivity standards solve only the first part of the integration problem, dealing with the creation of a common *channel* for communicating among various smart appliances. The second part of the problem is to establish a common *language* so that home appliances can actually understand each other and function in a collaborative manner. In general, this problem of *semantic interoperability* is much harder to solve than the realization of the physical transport channel for data. The main reason for these difficulties is the great *heterogeneity* of home appliances and the large variety of their embedding context. Home appliances cover all aspects of our daily lives including environmental controls, lighting, alarm systems and security, telecommunication, cooking, cleaning, and entertainment. There exist a vast number of potential scenarios for integrating such appliances. It is not possible for vendors to foresee all these applications and equip their devices with functionality that enables collaboration with every other device a customer would like to integrate. Consequently, there is the need for customization mechanisms that can be used for integrating different appliances and sensors into a common process that controls the smart home.

Such customization mechanisms can be seen as the "programming language" for the smart home. Primary requirements for such a programming language are *ease of use* and *rapid deployment*. Unlike industrial applications that

1536-1284/02/$17.00 © 2002 IEEE

are typically engineered by highly skilled programmers, control and automation programs for the smart home should be understandable by laypeople. Analogous to other "do-it-yourself" maintenance activities around the home, programs for the smart home should be changeable by third-party service providers as well as the homeowner herself. There is a good chance of achieving this goal because applications in home automation tend to have lower complexity than industrial automation systems. Still, traditional programming paradigms like textual programming languages appear inadequate for this purpose. Effective programming mechanisms for the smart home require innovative paradigms that lift programming to a level of abstraction that is similar to plugging in a new stereo or TV set. We will introduce three such innovative paradigms in the following section. Then we will present an example solution for programming the smart home.

Enabling Paradigms

Driven by the rapidly increasing complexity of software applications over the last decades, the area of *software engineering* has been established as a discipline of systematic construction and maintenance of quality software systems. Software engineering encompasses a broad spectrum of aspects including all life cycle activities starting from the requirements analysis of a new system up to the reverse engineering and modernization of outdated legacy systems. In this article we discuss three emerging software engineering paradigms that, in combination, have great potential for facilitating the programming of the smart home. These paradigms are *visual programming*, *component-based software construction*, and *connection-based programming*. The following three subsections introduce the basic ideas behind these three paradigms. Then we use an application example to illustrate how concerted use of these paradigms can facilitate the programming of home automation systems.

Visual Programming Languages

The development of visual programming languages (VL) has been driven by the experience that developers tend to understand pictures better than plain program text. One reason for their increased expressiveness is that pictures have a two-dimensional nature in contrast to sequential program text, which covers only one dimension. Visual formalisms have been used extensively as an aid to design and visualize algorithms including their control flow or data flow. Classic examples of such formalisms are flow charts or Nassi-Schneiderman diagrams. More recently, similar concepts have been adopted in the Unified Modeling Language (UML) in the form of activity diagrams (www.uml.org). Flow charts of this kind can easily be mapped to equivalent constructs in textual programming languages. Several software engineering tool vendors offer development environments that can perform this mapping automatically, making textual programming superfluous for applications of low or medium complexity. Today, visual programming languages are often used in combination with textual languages. Moreover, visual languages and software visualization paradigms are increasingly used to aid human understanding of the existing program code in legacy systems.

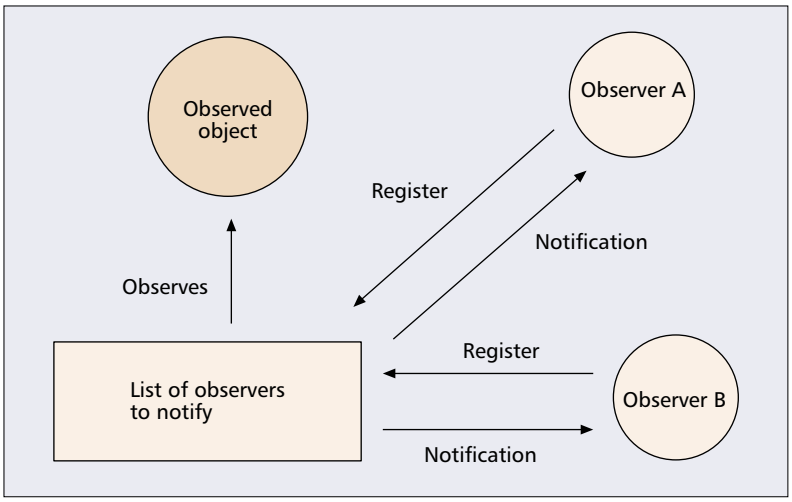
Apart from considerations of program *logic*, the area of visual languages was equally driven by progress in the domain of user interface design and human-computer interaction. Several so-called fourth-generation languages (4GL) introduced in the '80s and '90s included visual tools for user interface design as an integral component. This paradigm has been broadly adopted with popular programming tools like Microsoft's Visual Basic or, more recently, IBM's VisualAge for Java. Such visual programming languages typically promote event-driven architectures. This means that the programmer does not explicitly define the control flow; it is implicitly determined by the occurrence of user interface events (e.g., a mouse click on a button). Both visual programming paradigms, flow logic diagrams and event-driven user interface designs, are complementary rather than competing approaches. They can be integrated into a holistic solution for visual programming. We give an example of such a solution later in this article.

Both visual programming paradigms, flow-logic diagrams and event-driven user interface designs, are complementary rather than competing approaches. They can be integrated into a holistic solution for visual programming.

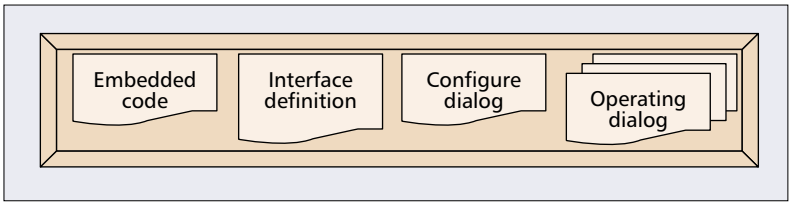
Component Software

The idea of component software has its roots in the great success of component-based manufacturing in the hardware sector. Component-based software systems are assembled from a number of preexisting pieces of software called *software components* (plus additional custom-made program code). Software components should be (re)usable in many different application contexts. Particularly, these components should be usable in unpredicted applications and by third parties. The term commercial off-the-shelf (COTS) was coined in the mid '90s as a concept for a binary piece of commercial software with a well defined application programming interface and documentation. The component market has gained momentum from the introduction of infrastructure for deploying components in programming languages and operating systems, such as Sun Microsystem's (Enterprise) Java Beans and Microsoft's COM+. Using the component paradigm for software construction has various benefits: it increases the degree of abstraction during programming, provides proven (error-free) solutions for certain aspects of the application domain, increases productivity, and facilitates maintenance and evolution of software systems.

Component-based software development has become an important part of modern software engineering methods. So-called *lightweight* components (i.e., fairly small in size) are already well understood. Lightweight components have become part of modern programming languages (e.g., the Swing library within Java). Most modern approaches to user interface development are component-based. In contrast to lightweight components in particular application domains, the construction and reuse of more heavyweight components in arbitrary application domains still pose many theoretical questions. Examples of such research questions are: how can a complex



■ Figure 1. *The observer pattern as the basis for connection-based programming.*



■ Figure 2. *Aspects of the microCommander component.*

component be documented so that it can be well understood by a third party? How can a developer find the components that best fit her current software project?

Still, our research indicates that the current state of component technology in software engineering is sufficient as an enabling paradigm for constructing software for home automation. In other words, we believe that libraries of relatively lightweight components are adequate to represent the building blocks to construct programs for the smart home. We present an example of such a library later.

Connection-Based Programming

Rather than being an independent paradigm in its own right, the introduction of connection-based programming has been driven mainly by the introduction of the previously discussed paradigms, component software and visual languages. Traditional software programs have followed the procedure call paradigm, where the procedure is the central abstraction called by a client to accomplish a specific service. Programming in this paradigm requires that the client has intimate knowledge about the procedures (services) provided by the server. However, this kind of knowledge is not present in component software because it is based on components from third parties that were separately developed. That is why component software requires a new programming paradigm called *connection-based programming*.

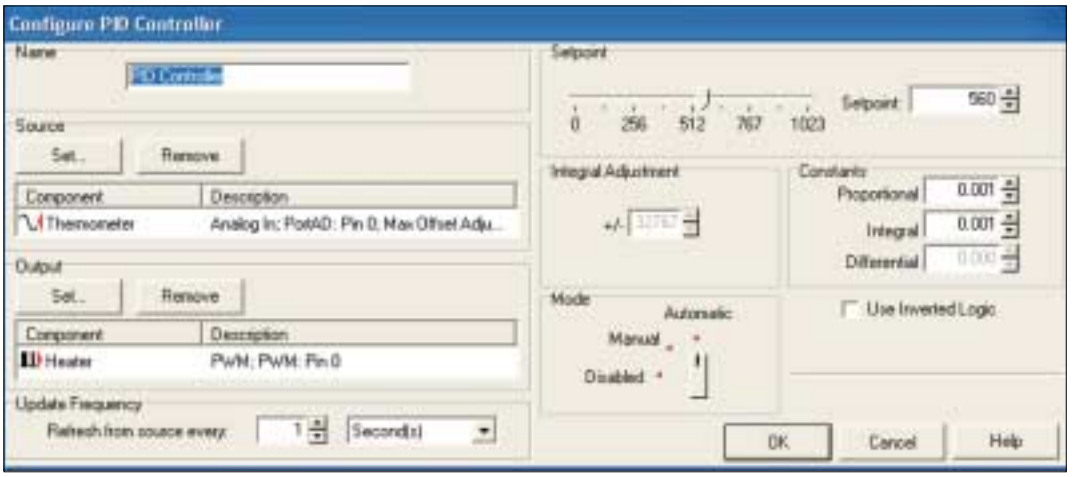
In connection-based programming, connections between pieces of software are not implicitly defined by procedure calls but are explicitly programmed. Connections represent the glue that binds together interfaces of different software components. The basis for connection-based programming is typically the so-called *observer* design pattern, which is shown in its general form in Fig. 1. The observer pattern works like a subscription mechanism that handles callbacks upon the occurrence of events. Software components interested in an event that could occur in another component can register a callback procedure with this component. This procedure is called every time the event of interest occurs. The interfaces of software components have to be tailored for connection-based programming — they have to provide subscription functions for all internal events that might be of interest to external components. This part of the interface is often called the *outgoing* interface of a component, as opposed to its *incoming* interface that consists of all callable service procedures.

Bringing It Together for Programming the Smart Home

In this section we discuss how the three enabling paradigms introduced in the previous section can be used to facilitate programming of the Smart Home. We report on the results of an industrial-driven collaborative research project carried out between the University of Victoria, British Columbia, Canada, and Intec Automation Inc., a Victoria company in the area of embedded systems. The project is supported by the Advanced Systems Institute of British Columbia.

Embedded Programming with Visual Components

Over the last few years, Intec has investigated how the visual programming paradigm can be used to facilitate the development of embedded control applications for industrial as well as private applications. As a result, Intec has developed *microCommander*, an application for presenting a visual programming interface to a system of embedded devices (www.microcommander.com). The software runs on a personal computer with access to a network connecting any number of devices. All of the devices must conform to a predefined component architecture that is recognized by microCommander. MicroCommander then allows a user to visually program these off-the-shelf software components without having to write any source code. Behind the scenes, each component consists of an *embedded code*, an *interface and behavior definition*, a *configure dialog*, and *operating dialogs* (Fig. 2). The *embedded code* containing the logic that operates a device usually executes on microcontrollers located at various places within the automated home. Depending on the complexity of the device, a single microcontroller may host the *embedded code* for a number of components. The *interface and behavior definition* describes how to interact with the device in terms of input and output messages. Messaging formats and policies are part of the component architecture, and it is critical that every device in the system strictly conform to these rules. Thus, it is guaranteed that every device is addressable and properly controllable.



■ **Figure 3.** *The configure dialog allows specification of a number of component-related parameters.*

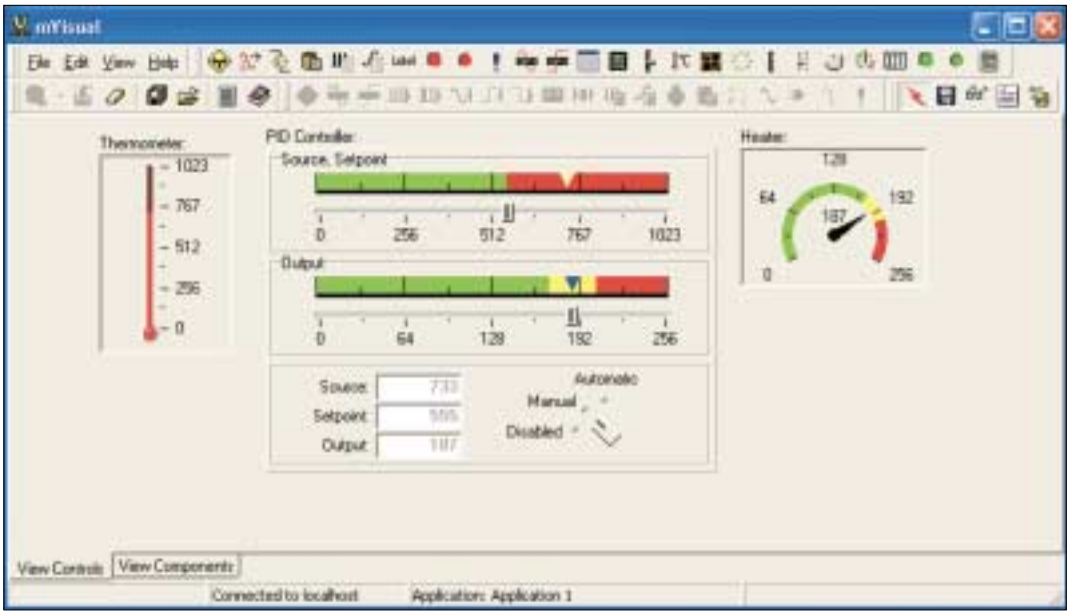
The *configure dialog* (Fig. 3) is a visual interface for programming properties such as microcontroller input and output assignments, default values, and states. This configuration setup is generally done only once during system installation, after which the component is exclusively controlled via the *operating dialogs*. The use of *configure dialogs* requires some domain knowledge, and thus would typically be done by third-party vendors during installation. For example, Fig. 3 shows the setup of a new heater system controlled by a PID control component [2].

The *operating dialogs* (Fig. 4) provide sophisticated visual interfaces to the devices embedded within the home. Each *operating dialog* is customized for the day-to-day usage of a device by a layperson. Both *configure dialogs* and *operating dialogs* reside within the microCommader application, and are part of its user interface. The application contains an extensible library of visual controls that the operating dialogs may utilize. MicroCommander thus acts as a PC-based remote control console to the device, allowing a home owner to manipulate and visually program a home from any Internet-ready PC running microCommander.

During initialization, microCommander learns about the devices present in the network by a process called *introspection*, in which each device automatically identifies itself. Operating platforms such as the Bluetooth wireless connectivity protocol or the Java Jini framework already provide mechanisms for introspection that support addressing, security, and authentication [1]. During introspection, each device publishes an *operating dialog* and, depending on security parameters, a *configure dialog*. At regular time intervals introspection is repeated in order to keep the view of the system up to date. Devices that were previously turned off or recently installed are automatically discovered.

The *operating dialogs* generally resemble real-world artifacts users are accustomed to (Fig. 4). Appliances appear on the application canvas as

In connection-based programming, connections between pieces of software are not implicitly defined by procedure calls but they are explicitly programmed. Connections represent the glue that binds together interfaces of different software components.



■ **Figure 4.** *An example of the devices found in an automated home. The controls include a thermometer, a furnace temperature gauge, and a heater controller.*

The microSynergy framework presents a development and execution environment for the connection-based programming paradigm. As a result of this programming paradigm the embedded appliances are unaware of the fact that they are part of a collaborative network.

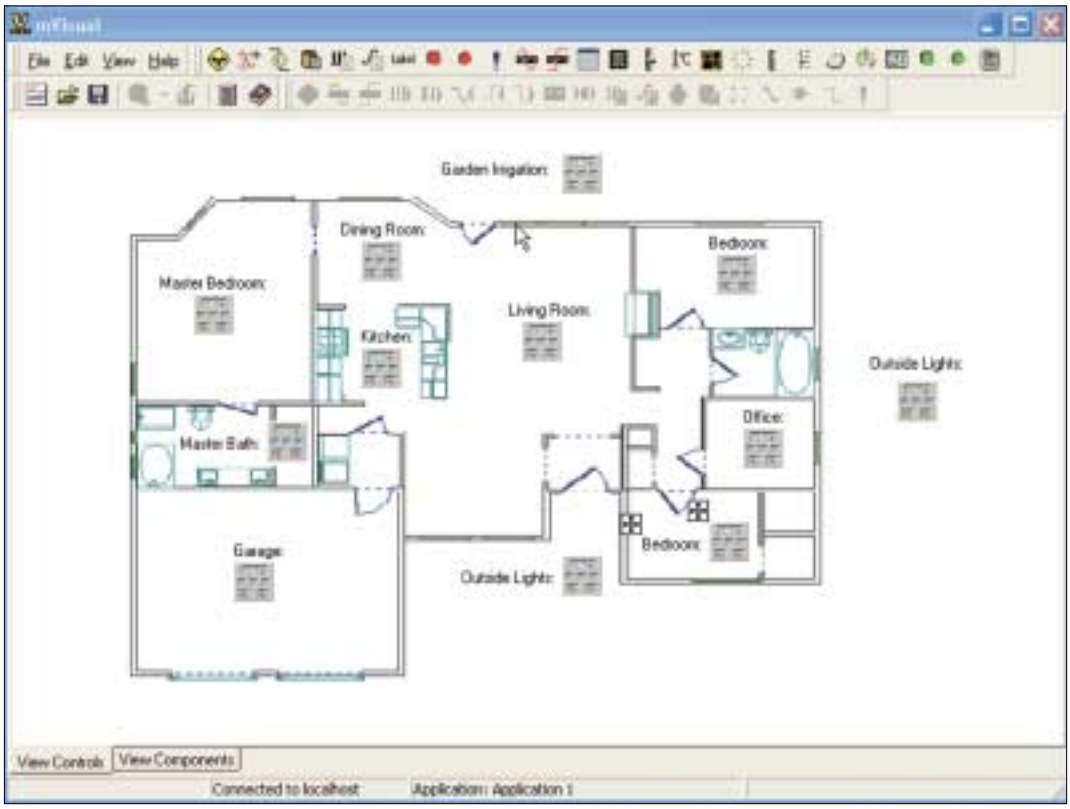


Figure 5. *Embedded in the floor plan of the house are icons representing the collection of devices within each room. By clicking on an icon a user gets a detailed view of the devices within that room (Fig. 4).*

familiar images that represent the appliances' current state by means such as plain text or colored gauges. Each component type may publish a range of *operating dialogs* that either provide a different look and feel, or are tailored to varying types of display devices. Input values to devices are changed via their graphical representation, by either entering new values in text fields or moving pictures of levers, switches, and gauges using the mouse. The *operating dialog* first translates this type of activity into commands understood by the corresponding software component, and then transmits that information in real time to the embedded code. The entire process is similar to walking up to a device and physically turning a knob. However, here it is done via a computer screen and mouse from any Internet-ready PC running the microCommander application.

As a result of periodically querying each component's state, the *operating dialog* provides feedback to changes as they occur in real time. For example, the turning of a light switch may immediately become apparent in a light sensor changing state, and adjusting the temperature of the furnace may induce a slow continuous change in a thermometer reading.

By filtering, grouping, and rearranging controls on the screen, different views of the automated home are possible (Fig. 5). Devices may be grouped by type, location, or relevance, allowing the creation of interfaces that are optimized for particular tasks such as power consumption, home security, or temperature control. Background images such as floor plans, wiring, and plumbing diagrams can be embedded into the view to provide a context in which to place devices.

Applications like microCommander act as visual control consoles for all the electronic devices embedded in the smart home, similar to the terminals used in cockpit or bridge controls of modern airliners and cruise ships. The underlying complexity of the system implementation is hidden, and the interface is reduced to the essentials. Moreover, the microCommander framework provides a general platform for visual component-based software assembly, without the need for textual programming languages. The typical edit-compile-test cycle of traditional software programming does not exist. Instead, a system is assembled and configured in real time using visual off-the-shelf components. Third-party vendors can easily develop their own net-enabled devices for use with the microCommander application by providing a conforming component interface to the device.

Using Visual Flow Charts for Connecting Smart Devices

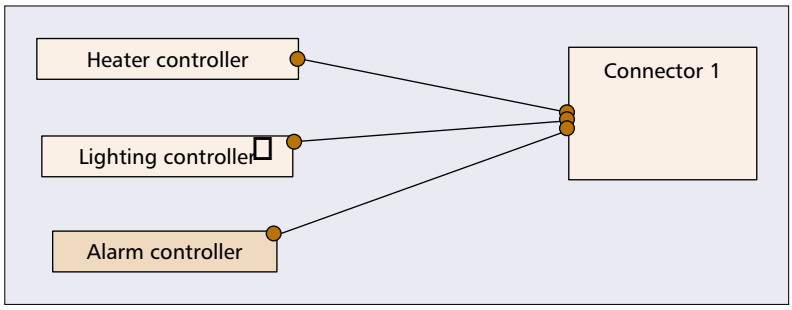
So far, we have discussed how visual programming is used for programming devices in the smart home. We now look at connecting many such devices into an integrated collaborative network. For this purpose, the University of Victoria and Intec Automation have jointly developed a technology prototype called microSynergy. MicroSynergy facilitates the development and execution of logic described with the Specification and Description Language (SDL). SDL has unambiguous formal semantics, is well accepted by the embedded systems engineering domain, and is easily understood by a layperson.

MicroSynergy consists of a *microSynergy editor* and a *microSynergy runtime engine*. Specifications created using the editor are downloaded to the runtime engine, which then controls the corresponding embedded devices accordingly. The microSynergy application applies *introspection* to discover the (type of) devices present within the home. This introspection mechanism is realized by sending out generic query messages to all possible device addresses when requested by the user of the microSynergy editor. As a result, the microSynergy editor is aware of every device in the system as well as the types of messages a device produces and the types of messages it understands. Similar to microCommander, the microSynergy editor also presents a user interface to the embedded device. However, in this case the interface omits the configure dialogs and operating dialogs and instead presents a view of the *interface definition*. Control logic is implemented in terms of input and output messages rather than visual controls. Editing SDL diagrams and establishing connections among the entities on the screen is done by simply clicking and dragging objects using the mouse and keyboard.

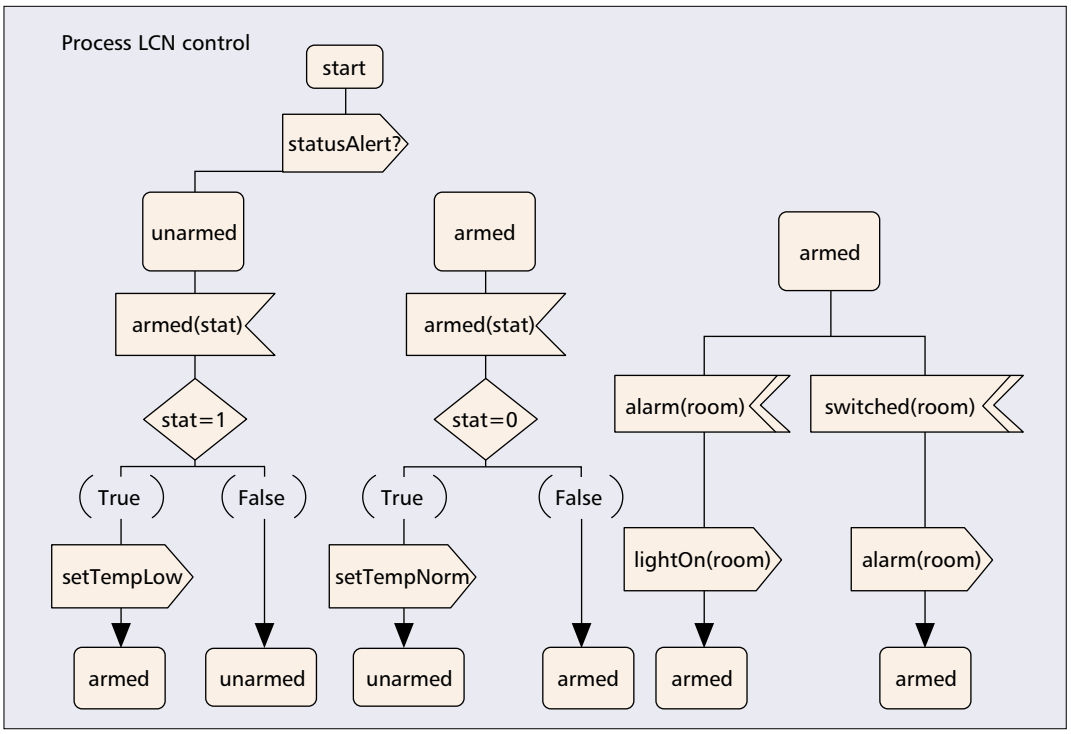
A home alarm system is an example of the need to establish elaborate logical dependencies in such a way that the events triggered by one device cause a response in another. The home alarm system, for example, once activated ought to trigger the lighting and video surveillance system. It should also automatically deactivate these systems once the threat to the home has passed. This type of scenario is easily programmed using microSynergy and SDL.

The first programming step is to connect the appropriate devices to an SDL block (Fig. 6). This step creates exclusive communication channels among the devices, ensuring that the inputs and outputs incorporated in the subsequent SDL logic are authentic. Devices may be connected to many different blocks and hence partake in separate activities. Within each SDL block resides a detailed specification of the dependencies among the connected devices (Fig. 7). Our example shows an SDL block that connects the heater controller with the alarm system and lighting control. The specification consists of *states* (represented as rectangles with round corners), *transitions* (represented by directed arcs), and conditions (diamond-shaped boxes). The flow of logic occurs from state to state via transitions. Conditional statements provide a selection mechanism for the next states. Communication with the corresponding devices occurs whenever a transition is traversed. One type of transition generates an output signal that is sent to the corresponding device, while another type of transition waits until an input signal is received. For example, the input signal at transition *armed(stat)* subsequently triggers the transition *setTempLow* if the value of *stat* is true.

After an SDL specification is complete, it can be downloaded to an embedded controller that



■ Figure 6. *A logical connector connected to three controllers.*



■ **Figure 7.** *The creation of internal connector logic with microSynergy.*

Future generations of cell phones may use Java for enabling more sophisticated interaction with the Smart Home. Moreover, emerging net-centric standards for service registry can be used to integrate the smart devices within a smart home with other community services.

executes the microSynergy runtime engine. The runtime engine provides a parallel execution platform for any number of SDL blocks. Although execution within each block is sequential, several different blocks can run at the same time.

The microSynergy framework presents a development and execution environment for the connection-based programming paradigm. As a result of this programming paradigm the embedded appliances are unaware of the fact that they are part of a collaborative network. New devices are easily added to the system, and the resulting new dependencies are quickly created.

Future Perspectives

If at least one of the devices embedded in the smart home supports TCP/IP, a small Web server can be deployed so that services may be offered via Simple Object Access Protocol (SOAP), an emerging standard for net-centric interfaces of all kinds. It combines the simple HTTP protocol with the universal data description language XML. With this kind of gateway, homeowners can use devices like Web-enabled cell phones to interact with the network just as easily as a Web browser. Future generations of cell phones may use Java to enable more sophisticated interaction with the smart home. Moreover, emerging net-centric standards for service registry like *Universal Description, Discovery, and Integration* (UDDI) and the newly standardized *Web Service Description Language* (WSDL) can be used to integrate the smart devices within a smart home with other community services. In this context, the capability of a connector being treated as a component enables scalability with respect to nested network communities. It can provide for hierarchies of components and therefore hierarchies of network communities. We will continue our collaborative research efforts in this direction.

Acknowledgments

We would like to thank Intec Automation for their collaboration on this research project. Furthermore, we thank the Advanced Systems Institute of British Columbia for supporting the research. Finally, thanks to Andrew McNair for his support in implementing the microSynergy editor.

References

[1] Sun Microsystems, "Jini Technology and Emerging Network Technologies," 2001; http://www.sun.com/jini/whitepapers/technologies.html
[2] Goodwin, Graebe, and Salgado, *Control System Design*, Prentice Hall, 2000.

Additional Reading

[1] Mitchele-Thiel, *Systems Engineering with SDL*, Wiley, 1997.

Biographies

JENS JAHNKE (jens@cs.uvic.ca) is an assistant professor in the Department of Computer Science at the University of Victoria, Canada. He holds a Ph.D. degree from the University of Paderborn, Germany (1999) and an M.Sc. degree from the University of Dortmund, Germany (1994). He is a Fellow of the Advanced Systems Institute of British Columbia and was awarded the prestigious Ernst-Denert Software Engineering Award (2000). His current research interest lies in advanced topics of network-centric software engineering.

MARC D'ENTREMONT (mdentrem@cs.uvic.ca) is an M.Sc. candidate in the Department of Computer Science at the University of Victoria, Canada. He holds a B.A. in economics (1993) from the University of Guelph, and a B.Sc. (2002) from the University of Victoria, Canada. His research interests involve visual languages and evolution of embedded networks.

JOCHEN STIER (jstier@cs.uvic.ca) is a Ph.D. candidate in the Department of Computer Science at the University of Victoria, Canada. He holds an M.Sc. (1996) and a B.Sc. (1994) from the University of Victoria. His research interests are in the area of rapid development techniques for mechatronic systems. Prior to his current position, he held various positions as a software engineer.