Engineering Component-based Net-Centric Systems for Embedded Applications

Jens H. Jahnke
University of Victoria, Dept. of Computer Science
Engineering Office Wing 321
Victoria, V8W3P6, Canada B.C.
+1 (250) 472 4542
jens@acm.org

ABSTRACT

The omnipresence of the Internet and the World Wide Web (Web) via phone lines, cable-TV, power lines, and wireless RF devices has created an inexpensive media for telemonitoring and remotely controlling distributed electronic appliances. The great variety of potential benefits of aggregating and connecting embedded systems over the Internet is matched by the currently unsolved problem of how to design, test, maintain, and evolve such heterogeneous, collaborative systems. Recently, component-oriented software development has shown great potential for cutting production costs and improving the maintainability of systems. We discuss component-oriented engineering of embedded control software in the light of emerging requirements of distributed, net-centric systems. Our approach is based on applying the graphical specification language SDL for composing complex networks of embedded software components. From the SDL specification, we generate internet-aware connector components to local embedded controller networks. The described research is carried out in a collaborative effort between industry and academia.

Categories and Subject Descriptors

D.3.3 [**Software Engineering**]: Design Tools and Techniques – *component-orientation, network integration*

General Terms

Design

Keywords

Component-oriented development, embedded software, SDL, network-centric computing

1. INTRODUCTION

During the last decade, miniaturized computers (micro controllers) programmed with dedicated application software (*embedded systems*) have replaced conventional electronics in almost every application domain [1]. The cost effectiveness of mass-produced, multi-purpose micro controllers and the flexibility provided by the embedded control software has created a great spectrum of new applications.

Today, embedded systems are ubiquitous. They can be found in a vast variety of products ranging from cars over cellular phones and cameras up to household appliances. Home networking is now considered to be one of the computer industry's fastest growing markets. The omnipresence of the Internet and the World Wide Web (Web) via phone lines, cable-TV, power lines, and wireless RF devices has created an inexpensive media for telemonitoring and remotely controlling distributed electronic appliances.

Traditionally, the emphasize in developing software for embedded systems has been on maximizing run-time and memory efficiency to minimize hardware costs as much as possible. In the presence of continuously decreasing hardware costs and the increasing complexity of tasks controlled by embedded systems, additional goals like maintainability and reliability have recently gained importance. In particular, these requirements are of utmost importance in safety critical applications domains, e.g., like transportation, factory control, and telecommunication. Still, current industrial development practices (processes, tools, and techniques) for software in embedded systems lags far behind the state-of-the-art in normal software engineering areas. Despite all the progress made in the general software engineering arena (e.g., specification languages, frameworks, CASE tools, code generation etc.), most embedded software is still developed using primitive programming languages like assembler or C without formal design or requirements analysis. This development practice is inefficient for complex systems because it impedes software reuse and maintenance. Moreover, it requires significant amount of expertise and is prone to error. These problems have begun to become even more severe with the current trend to interconnect embedded systems in net-centric architectures. The great variety of potential benefits of aggregating and connecting embedded systems over the Internet is matched by the currently unsolved problem of how to design, test, maintain, and evolve such heterogeneous, collaborative systems.

A component-oriented approach can be used to tackle the problems stated above. Recently, the notion of reusable software components has proven extremely useful in various software engineering domains. Most currently available integrated software development environments provide an extensible library of user interface components to rapidly prototype graphical user interfaces [2]. In the area of embedded control systems, component oriented software development has shown to cut production costs and improve the maintainability of systems [3]. In this paper, we present an approach to component-oriented engineering of embedded control software for network-centric systems. This approach is based on applying the graphical specification language SDL [4] for composing complex networks of embedded software components. From the SDL specification, we generate internet-aware connector components to local embedded controller networks. The described research is carried out in a collaborative effort between the University of Victoria and Intec Automation Inc.

The core of this paper is divided in two parts: the following section describes architectural requirements and aspects for net-centric embedded components. Subsequently, Section 3 outlines our approach to meet these requirements. Related work is discussed in Section 4. We close with concluding remarks in Section 5.

2. NET-CENTRIC EMBEDDED COMPONENTS

The ongoing miniaturization of micro controller hardware in combination with the broad acceptance of the Internet in the private and the public sector has created a huge market for networked embedded devices. Experts predict a significant market in particular in the area of home networking. Beginning with the alarm sensors, door and window locks, heating, lighting, and ranging over all kinds of electronic appliances (e.g., VCRs, TV, oven, etc.), the possibilities for networked applications are countless. Access to the Internet allows for monitoring and controlling selected parts of such networks from remote locations. For example, a home alarm system could directly be connected to a security agency or the police station. The similar setup can be used to remotely control home appliances from a computer at work or a cellular phone.

In order to make such applications affordable for a broad number of customers and due to the high dynamicity of embedded systems networks, ease of development and maintenance is a number one requirement. In contrast to conventional programming techniques and languages that require a significant amount of expertise and time, component-oriented software development is based on a *plug & play* paradigm: the developer composes an application by (1) selecting predefined software components from a component library, (2) customizing generic parameters of these components, and (3) connecting them to obtain the entire application. In general, reusable software components have been defined as *units of composition with contractually specified interfaces and explicit context dependencies only. They can be deployed independently and are subject to composition by third parties* [6].

In the domain of network-centric embedded systems, we have to consider components on different levels of abstraction (cf. Figure 2). On the lowest abstraction level, the primitive components represent the basic building blocks for micro controller programs. We denote such primitive components as *process components*, because they perform basic functions in the control process. Process components can for example be *analog or digital sensors*, *(PID) controllers*, *pulse width modulators*, etc. A micro controller program can be developed as a configuration of several connected process components. This configuration can again be treated as a more abstract component, i.e., an *application block component*. Several application block components can be composed to build a *local component network* (LCN). An example for an LCN would be a single home network of embedded micro controllers (cf. Figure 2). Several such LCNs can again be composed to global component networks (GCNs). For this composition the LCNs can be treated as black box components with defined interfaces and behavior.

Process components

In addition to their embedded code, process components have to provide a clear interface to facilitate reuse. In fact, a process component with a single interface might provide several variants of implementations for different embedded micro controller platforms, e.g., Motorola, Siemens, Intel X86, etc. Furthermore, if we consider network-centric applications, process components should provide *user interfaces* for tele-monitoring, operating, and configuring their parameters. These user interfaces will typically be executed on a different host than the system that executes the functional embedded code. Moreover, in a heterogeneous net-centric environment, there is the need for various versions of user interfaces for different client platforms, e.g., visual GUIs for workstations and hand-held PCs, audio controls for telephone lines, etc. Figure 1 summarizes these aspects of process components.

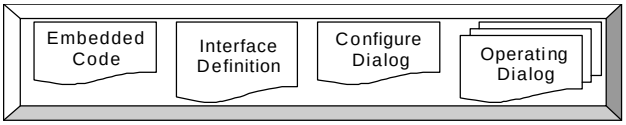

Figure 1. Aspects of Net-Centric Process Components

Application Block Components

An application block consists of all process components that run on a single micro controller. Typically, process components are executed concurrently. Consequently, each micro controller has to have the central *scheduler kernel* managing this parallel execution. The application block itself is treated as a composite component with a precisely defined interface. Typically, the interface defines a set of possible incoming and outgoing signals (or events) with or without associated data parameters.

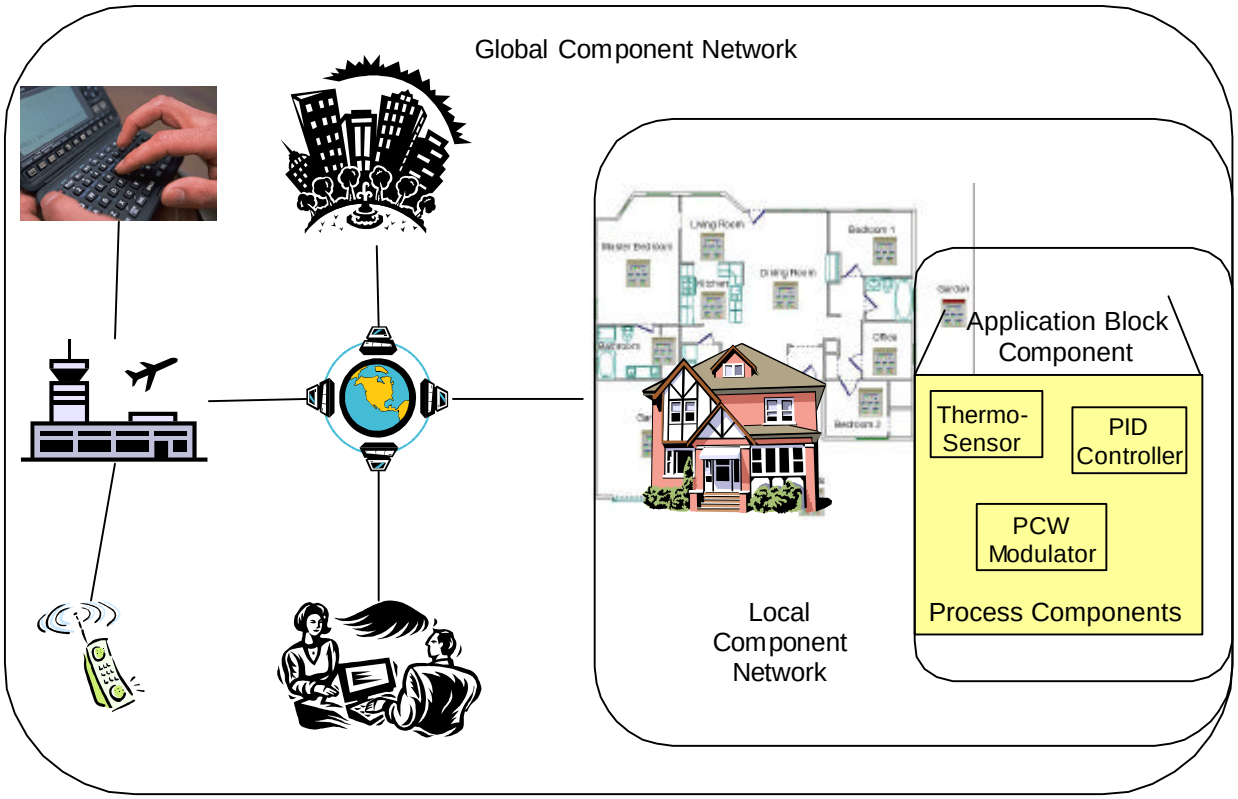

Figure 2. Architecture of network-centric embedded components

Local Component Networks

The rationale behind the distinction between the local and global component networks is that they require different technologies because of their different requirements for performance and flexibility. LCNs are local networks of micro controllers with relatively high interdependency. They have to perform their task efficiently and cost effectively. Abstraction mechanisms like location transparency and dynamic service binding are less important than real time characteristics, runtime optimization, and a small footprint of the micro controller operating system. Thus, it is not needed nor desirable to use heavyweight communication middleware (e.g. CORBA or CORBA/RT [7]) to integrate application block components in LCNs.

Global Component Networks

In comparison to LCNs, the architecture of the global component network (GCN) is much more dynamic. New services are added on the fly and integrated with existing devices. Existing services are retired, modified, relocated, or removed from the network. Moreover, some remote services might not be available all the time. This results in a continuously evolving configuration of distributed embedded systems. In order to deal with this pace of evolution, GCNs require powerful mechanisms and middleware to hide various details of system implementation and deployment from the client of the network.

Another characteristic of GCNs is that the degree of heterogeneity is significantly higher than in LCNs. Middleware technology for GCNs has to enable the integration of components that were not specifically developed to work together. This means that their interfaces might not fit exactly. Of course, it is possible to develop adapters to integrate incompatible component interfaces. Still, in a highly dynamic network, the effort involved in manually coding such adapters is often not practical. Therefore, the integration middleware has to provide means for rapidly mapping component interfaces. *Introspection* and interface query mechanisms are extremely important for building spontaneous collaborations of components and performing (partial) mappings at run time.

Obviously, the described transparency and introspection mechanisms have their price in terms of memory and runtime overhead. Compared to LCNs, however, the real time requirements in a GCN are much lower. Likewise, a GCN can employ larger and more powerful computers to serve as gateways among different LCNs.

3. *microSynergy* – ENGINEERING NET-CENTRIC EMBEDDED COMPONENTS

This section describes a component-oriented approach to engineering embedded micro controller software in a network-centric environment. This research has been carried out in tight collaboration with Intec Automation Inc., a local company in the area of embedded systems. Our approach is mainly based on a combination of readily available net-centric component technology (*Java Beans* [2]) with the emerging *Jini* [5] connectivity middleware and a rapid approach for interface mapping based on the graphical specification language *SDL* [4]. In the future, Intec Automation intends to implement the result of our studies with the current prototype (called *microSynergy*) in a product for rapidly developing and evolving distributed embedded applications. We will now use the architectural structure of net-centric embedded networks (introduced in the previous section) to systematically describe our approach.

3.1 Realization of Process Components

In Section 2, we argued that there are various aspects to embedded process components for net-centric applications (cf. Figure 1). Net-centric process components themselves are executed in a distributed fashion. In particular, their user interfaces are typically executed remotely from the micro controller that runs the embedded code performing the actual functions. Figure shows an architectural overview on the approach to deploying and executing process components chosen for *microSynergy*. Note, that distributed libraries are employed as repositories for the different user interfaces and the embedded code of process components, respectively. This approach reflects on the previously discussed distributed nature of net-centric embedded components. In Figure 3, we use numbers to uniquely identify process components, and we use different shapes to mark the parts that implement their different aspects. More precisely, we use ovals for representing embedded code and squares (resp. diamonds) for representing user interface code for configuration purposes (resp. operational purposes).

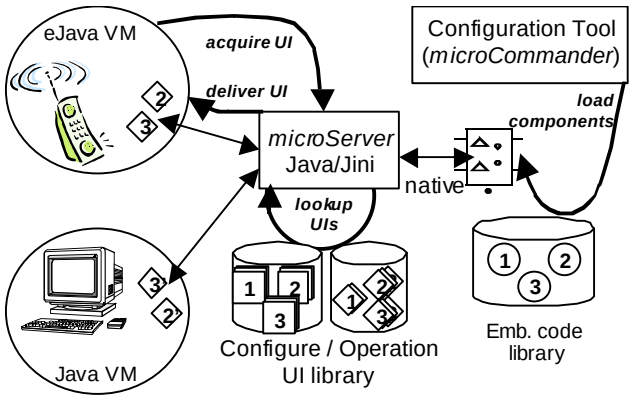

Figure 3. *microSynergy* architecture for deploying and executing Process Components

The configuration of process components and their deployment on the micro controller(s) is performed using an integrated development environment (IDE) called *microCommander*. This activity replaces the traditional manual programming coding activity. Using *microCommander* is similar to using modern component-based IDEs for user interfaces [2]. The IDE provides predefined user interfaces for any deployed components that can be arranged on a canvas for configuration, simulation, and incremental debugging purposes.

Figure 4 shows such a canvas for the example of a home heater controller, using three process components, namely a *thermometer*, a *PID-controller*, and a pulse-width-modulator that is connected to a simulated *heater*. Note that the actual connections between deployed components are established using the configuration dialog for each component. For example, the upper-left part of Figure 5 shows that the configuration dialog of the PID controller component defines that its input ("*Source*") is connected to the (analog) output of the thermometer component. Other process components can be connected in a similar fashion to build complex embedded applications. The actual embedded code is written to the micro controllers' non-volatile Flash memory. Because of the strong limitation of resources on the embedded platform, the code for each component type exists just once even if it might be instantiated multiple times.

The user interfaces (for configuration or operation purposes) can be executed on remote hosts distributed over the Internet (cf. Figure 3). These remote hosts might have very different hardware and software characteristics, ranging from small personal devices like cell phones and palm tops, over smart TVs, up to PC workstations and mainframes. We solve a part of this heterogeneity problem by using *Java* as the implementation language for the user-interface parts of process components. Java runtime environment can be found on almost every computing platform. Subsets of Java (*eJava – embedded Java* or *pJava – personal Java*) are increasingly used for small devices .

Still, just using Java solves only a part of the heterogeneity problem. Obviously, different versions of user interfaces are needed to adhere to the various characteristics of the different client platforms. Moreover, not all micro controllers do have TCP/IP ports to the Internet onboard. They typically employ proprietary protocols using inexpensive RS-232, RS-485 or similar ports. We have introduced an additional network server object (called *microServer* in Figure) to solve both problems:

?? *microServer* translates the internet protocol (TCP/IP) to the native protocol understood by the micro controller, and

?? it sends appropriate user interfaces to remote clients. For this purpose, each remote client has to determine the type of the required user interface (e.g., cell phone, web browser, graphical, textual etc.). Then, a Java object that represents the appropriate visual control is looked up from the user interface repositories and sent to the client.

3.2 Networks of application block components

The technology described so far provides means for component-oriented development of software for micro controllers and their remote configuration and operation. Additional mechanisms are needed for networking several such application blocks. Two main issues have to be addressed:

(1) the interfaces of each application block component has to be declared, and

(2) the mapping between the interfaces of collaborating application block components has to be defined.

3.2.1 Interfaces of application blocks

Interaction between embedded micro controllers is represented as an exchange of signals (or events), possibly with data parameters. For the example, let us assume that we would like our sample heater controller to understand four external input signals, namely *switchNorm* for switching the heater to normal operation, *switchLow* for switching the heater to low heat, *setTemp* for setting the standard temperature (normal heat mode), and *turnOff* for turning it off. Additionally, we would like our heater controller to send an external signal when it is turned on or off, respectively. In our approach, we use dedicated process components, so-called *in-gates* and *out-gates* for letting the developer add ports for input and output signals, respectively. These *gate* components can be deployed in the "*drag&drop*" style analogously to other process components. They can be connected to internal events and data sources that are relevant in external applications.

More precisely, each instance of an in- and out-gate has a unique *name* that defines the name of the signal associated to it. Out-gates have a data input that can be connected to any data output of local process components. In addition, out-gates have another input *strobe* that sources a digital input. Whenever the state of *strobe* changes, the out-gate component generates an external signal *name(data)*, where *name* is the name of the gate and *data* is the current value at the data input.

Analogously, *in-gate* components have a data output that can be sourced by any other process component. In addition, in-gates have a digital output *pending* and a digital input *clear*. Incoming messages are queued in instances of *in-gate* components. Whenever the queue is not empty the *pending* output is active (set to 1). Messages (and their data values) can be cleared by triggering the clear input of an *in-gate* (cf. Figure 6).

3.2.2 Collaboration among application blocks

The possibility of deploying *in-* and *out-gate* components on micro controllers provides a simple yet powerful mechanisms for defining external interfaces to application block

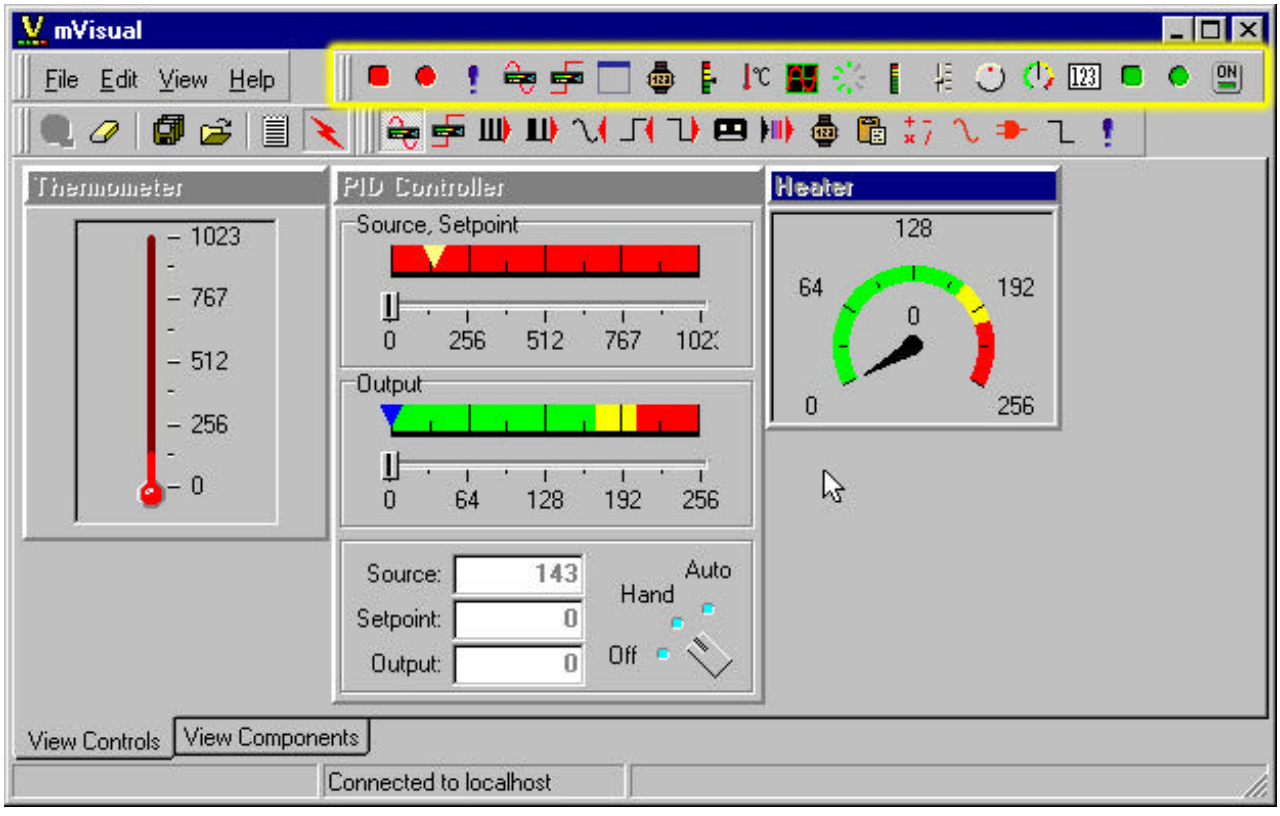

Figure 4. Component-based IDE *microCommander*

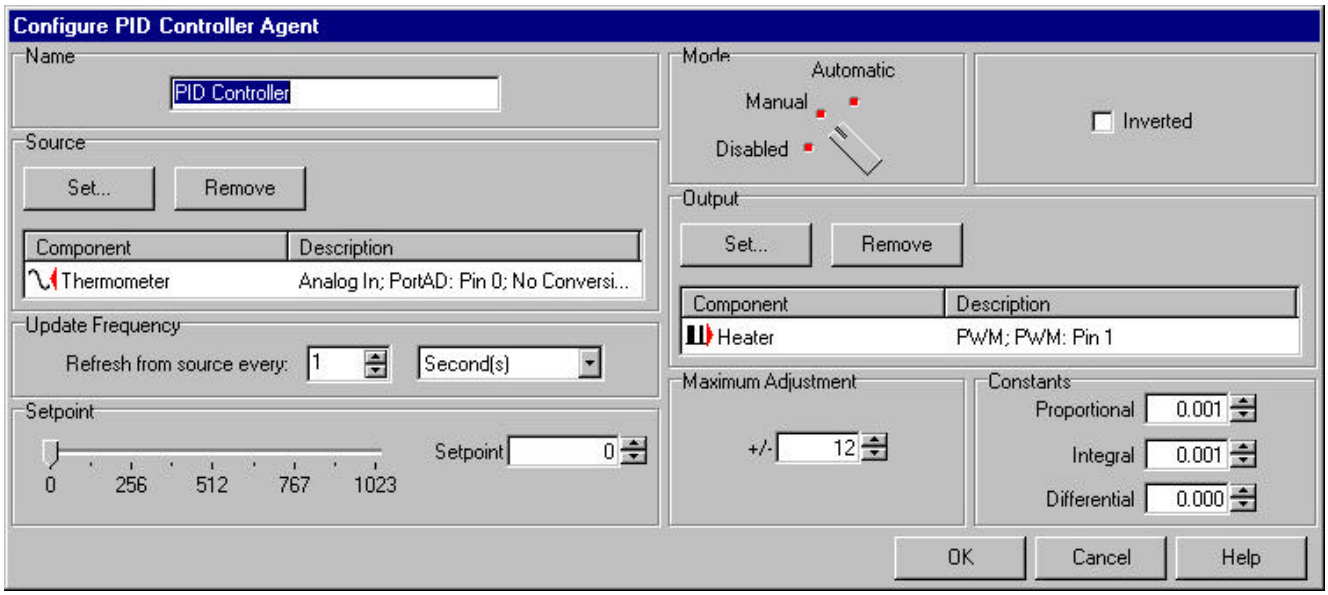

Figure 5. Configuration Dialog for component *PIDController*

components. The actual interaction among several such net-centric micro controllers is defined with a tool separate from *microCommander*, called *microSynergy editor*. This tool is based on the graphical specification and description language SDL [4]. SDL has been developed for the specification of complex, event-driven, real-time, and interactive applications involving many concurrent activities that communicate using discrete signals. Being developed by the *International Telecommunications Union* (ITU), SDL initially was intended to serve as a specification language for telecommunication systems. Today, it is increasingly adopted for other application areas, in particular, in the domain of engineering embedded systems. A major advantage of SDL over alternative specification languages like e.g., the *Unified Modeling Language* (UML) [8], is its formally defined and standardized syntax and semantics. Moreover, SDL has a graphical notation, as well, as an equivalent textual representation. This feature facilitates the exchange of SDL data among different tools.

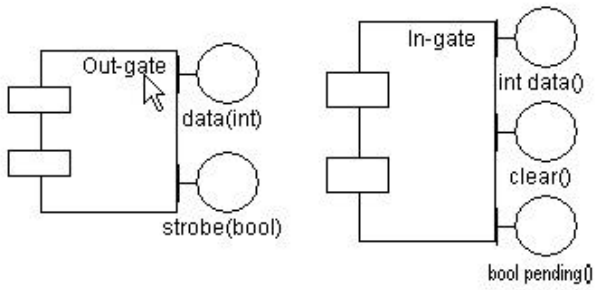

Figure 6. Design of In/Out Gate Components

Upon invocation the *microSynergy editor* hooks up to a selected LCN by connecting to the TCP/IP enabled *microServer* component that controls the LCN (cf. Figure 3). Then it requests this *microServer* to introspect all application blocks existing in the LCN and queries them for their *in-* and *out-gate* components. Figure 7 shows a simple interface as the result from introspecting the previously discussed application block *heater controller*. In this example, application block *HeaterController* comprises four in-gates (*setLow*, *setNorm*, *turnOff*, and *setTemp*) and two out gates (*turnedOn* and *turnedOff*).

According to the graphical SDL syntax, *microSynergy* visualizes the application blocks deployed on each micro controller as rectangles. Figure 8 shows an SDL model with our sample application block *Heater Controller* and two additional application blocks *Alarm Control* and *Lighting Control*. In order to combine these distributed micro controllers to a collaborative network, we need a mechanism to dispatch external signals between among them. Note that in the general case, we cannot simply map in-gates to out-gates, because we have to provide mechanisms for *a-posteriori* integration of components. This means that have to be able to integrate micro controllers in distributed applications in a way that were not anticipated during their development. Consequently, there is the requirement for more sophisticated connectors than simple channels or pipes for signals.

```
Interface HeaterController;
    In Signal setLow(), setNorm(), turnOff(),
              setTemp(integer);
    Out Signal turnedOn(), turnedOff();
Endinterface HeaterController;
```

Figure 7. Interface of *Heater Controller*

We use SDL to specify and generate these sophisticated connectors. The example in Figure 9 contains one such connector block named *LCN Control*. *microSynergy* renders connector blocks differently from application blocks (dashed border) to make it easy for the developer to distinguish between both concepts. Note that the developer is free to specify more than one connector blocks serving different purposes. We have chosen SDL *channels* for connecting connector blocks to application block components. Channels are represented as (directed) lines in Figure 8. Rectangular brackets are used for specifying the signal names that travel in each direction. The tool validates that these names correspond to names of in- and out-gates of the connected application block components.

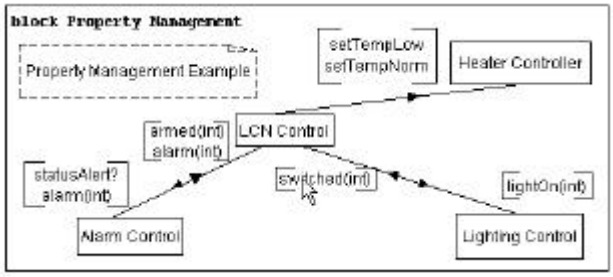

Figure 8. SDL model of LCN *Property Management*

According to the formal SDL semantics, channels delay signals during transport by a non-deterministic duration. Therefore, no assumptions can be made about the arrival time of two signals that have been sent to two different channels at the same time.

In addition to channels, SDL provides the notion of signal *routes* and *non-delaying channels* for connecting blocks. We decided to use standard (delaying) channels, because the current implementation of our signal distribution mechanism does not consider hard real-rime constraints. We are aware of the fact that this decision restricts our current approach to applications that do not require real-time communication among distributed micro-controllers. Still, there are a large number of net-centric applications that meet this requirement.

For the actual specification of connector blocks, we use the concept of (extended) finite state machines (FSMs) provided by SDL. This is illustrated in Figure 9 for the example connector block *LCN Control*. An FSM (in SDL) consists of *states* (represented as rectangles with round corners) and *transitions* (represented by directed arcs). The initial starting state is clearly marked by an oval. In contrast to other state machine models (e.g., UML State Charts), transitions are always drawn from top to bottom and states can have multiple occurrences in the diagram.

For our example, let us assume that the developer of the Alarm Controller used *microCommander* to deploy an *in-gate* component in such a way that a signal *armed(stat)* is send whenever the Alarm Controller is switched on or off. Note that in this example *stat* is a parameter (with value 1 or 0) representing the status of the alarm controller, i.e., on or off, respectively. Furthermore, let us assume that the LCN developer now wants to integrate the alarm controller with the heater controller of the property. The idea is to switch the heater to low temperature mode whenever the property's residents are absent, that is, whenever the alarm control is

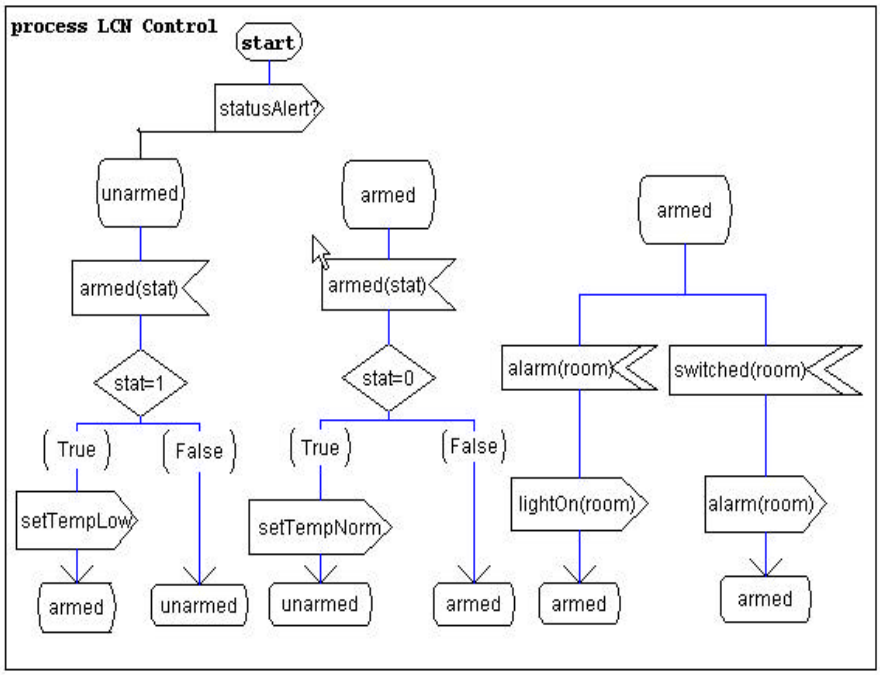

Figure 9. Specification of connector *LCN Control*

armed. This scenario requires a mapping between the *armed(stat)* signal of the alarm controller and the *setLow* (resp. *setNorm*) signal of the heater controller.

Figure 9 shows that such a mapping can easily be created using FSMs. The SDL symbol for an in-going signal is a rectangle with a cut out ("in-going") triangle. Inverse to this, the SDL uses a rectangle with an "out-going" triangle to specify an out-going signal. The left-hand side of Figure 9 specifies that if an *armed(stat)* signal occurs while the LCN is in state *unnamed*, the LCN will change states to *armed* if the condition *stat=1* holds. In this case the desired signal *setTempLow* is generated. The middle part of Figure 9 shows an analogous specification for setting the heater control mode back to normal whenever the alarm controller is unarmed.

Note that signals of type *armed(stat)* are created whenever the alarm controller is switched on or off. Hence, initially, the LCN Controller does not have information of the status of the alarm controller, until the first status switch has occurred. Therefore, we have specified a signal *statusAlert?* that is initiated by the *LCN Controller* at startup. *StatusAlert?* triggers the *alarm controller* to publish its current status, i.e., to send an *armed(stat)* signal. Such a status enquiry can easily be implemented as a convention of all net-centric micro controllers by configuring a dedicated in-gate component with *microCommander*. Note that this can be done without prior knowledge about the specific networked applications the controllers will participate in.

The right-hand side of Figure 9 shows another example for networking the alarm controller with the lighting control system. Here, *microSynergy* allows for treating the light switches as additional sensors for the alarm system: an *alarm(room)* signal is raised, whenever a light is switched and the system is armed. On the other hand, the light is automatically switched on in a room where the alarm controller has detected an alarm. Note signal symbols with a double triangle mark priority signals in SDL (e.g., *alarm(room)* in Figure 8).

There are many additional SDL modeling concepts that we currently do not cover within *microSynergy*, e.g., macros, procedures, exceptions, etc. Still, even the currently supported subset of SDL is powerful enough to create flexible mappings between embedded application block components running on distributed micro controllers.

Deployment of Connector Components

The generation of executable from the described SDL specification of connector components is straightforward. Analogously to several other SDL tools, we could generate procedural code (e.g., in C or C++) that handles the signal distribution in the *microServer* that controls the selected LCN. The problem of this approach is, however, that this code would have to be compiled and linked statically to the rest of the *microServer* software. Hence, this solution does not meet our requirement of being able to dynamically deploy and change connector components at run-time.

Therefore, we have chosen an interpretative approach. We translate the SDL specifications into highly compact scripts that can dynamically be downloaded and executed by a *microServer*. Since the embedded platform that executes microServer only has very limited resources, we did not use any off-the-shelf scripting interpreter like. Rather we defined very concise binary format called CEL (Connector Execution Language). Compilation of the specified FSMs to CEL is done in two steps (cf. Figure 11). Firstly, we unparse the FSMs into a textual representation based on XML. This textual representation is called Connector Description Language (CDL). We generate CEL in a subsequent compilation step. The CEL scripts are then send to the *microServer* target via the Internet.

Note that we use an XML-based textual representation instead of the native SDL text format because it is simpler and enables us to leverage from a great variety of readily available libraries and tools for the development of the compiler. Figure 10 shows the beginning of a CDL script generated from the FSM in Figure 9. Basically, each state flow from top to bottom in Figure 9 is translated into a so-called *thread* in CEL. Threads have a start state and a sequence of statements that can be out signals <OUT>, in signals <IN>, conditions <CSTMT>, and state transitions <TRANS>. Note that Figure 10 only shows the left-most thread from Figure 9. In addition to threads, a CEL connector contains the names of all connected components in a so-called link section <LINK>.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE CONNECTOR SYSTEM "CDL.dtd">
<CONNECTOR id="LCNControl">
    <LINK>LightingController</LINK>
    <LINK>HeaterController</LINK>
    <LINK>AlarmControl</LINK>
    <THREAD state="*START*">
        <STMT>
            <OUT>statusAlert?</OUT>
            <TRANS>unarmed</TRANS>
            <IN param="stat">armed</IN>
            <CSTMT>
                <COND>
                    <PARAM>
                        <VAR idr="stat"/>
                    </PARAM>
                    <REL op="EQ"/>
                    <PARAM>
                        <CNST>1</CNST>
                    </PARAM>
                </COND>
                <STMT>
                    <OUT>setTempLow</OUT>
                    <TRANS>armed</TRANS>
                </STMT>
                <STMT>
                    <TRANS>unarmed</TRANS>
                </STMT>
            </CSTMT>
        </STMT>
    </THREAD>
    <THREAD state="armed">
        <!–REST OF CONNECTOR SKIPPED -->
```

Figure 10. CDL generated for connector *LCN Control*

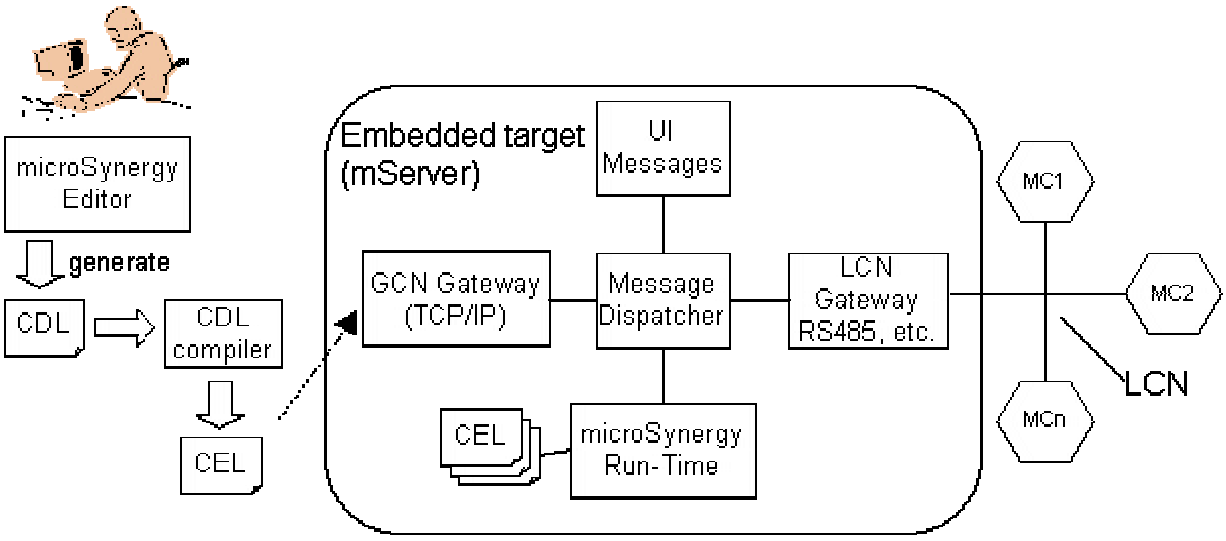

Figure 11. Deployment of Connector Components

3.3 Global networking of embedded components

Network evolution is one of the major challenges that have to be resolved in a successful approach to integrating global component networks (GCNs). Today, there exist a number of *distribution middleware* products, e.g., CORBA, DCOM, DCE, Java/RMI, Jini, etc. [13]. In our opinion, the concepts provided by Jini are most suitable for networking highly dynamic networks in distributed embedded applications. Jini technology has been publicly released in January 1999 by Sun Microsystems. It has been built on top of Java technology. Thus, it is actually not a *middleware* in the traditional sense, because it does not deal with heterogeneous programming languages like, e.g., CORBA. Rather, it is a connection technology that has specifically been developed for Java. At first sight, choosing a language-specific platform for integrating distributed components might appear unwise. However, by building on top of Java, Jini technology is leveraged by all modern software engineering concepts that have been built into this net-centric programming language, e.g., portability, mobility, component-orientation, introspection, security etc. Given the omnipresence of Java execution environments on all kinds of platforms starting from micro controllers to workstations, trading language independence against the benefits attached to Java technology appears to be a small sacrifice. Java is about to become a major integration technology for GCNs in the foreseeable future. Even for LCNs Java has become increasingly relevant. However, depending on the specific requirements and constraints of particular LCNs, application-specific protocols and platforms will remain important in this area. Still, Java/Jini nicely interfaces with these other emerging network technologies like *Bluetooth* (proximity-based wireless networking), *JetSend* (intelligent service negotiation), *HAVi* (Home Audio-Video interoperability) [9].

A central feature of Jini is that it provides mechanisms for dynamically building and evolving distributed *communities* by plugging components in and out the network. Each community has a Jini *Lookup Service* which acts as a broker between available Jini services and their clients. Generally, a GCN includes many such lookup servers. When a component that offers or requests Jini services is plugged into the network, it tries to find at least one lookup server. This is typically done using the Jini *multicast request protocol* [5].

In a future version of *microSynergy*, we intend to use this protocol to register the *microServers* of each LCN with at least one Jini lookup server. Note that the encircled *Int* in Figure 12 stands for a registered *interface* of an LCN. Analogously, local *microServers* can query the Jini lookup service for the interfaces of all LCNs in their "network neighborhood". With this information, the developer can use techniques similar to the approach presented in Section 3.2.2 in order to federate several LCNs to global networks.

Dealing with evolution

In contrast to other distribution technologies, Jini service objects cannot be used for an indefinite period of time. Rather, Jini introduces the concept of *resource leasing*, i.e., resources can be reserved (*leased*) for a certain time frame only. When this time has expired and the service is still needed, the lease has to be renewed. This simple but effective concept provides Jini networks with a sort of *self-healing* quality, because proxy objects for services that become unavailable will expire over time. Moreover, Jini clients *necessarily* have to take into account that services can become unavailable. In a way, the leasing mechanism has a function similar to a garbage collector: it eventually removes all traces of services that have become unavailable.

4. Related work

The idea of constructing software by configuring and connecting proven, reusable components (as opposed to manual programming) has existed for several decades. During the 90's component-oriented construction has gained increasing interest in the commercial section. This popularity has been driven by the availability of reusable frameworks and pattern libraries for object-oriented languages like C++ and Java [10] [11]. Johnson gives a good overview on the pros and cons of employing components and other reusability technology for software construction [12]. One prominent problem of component and framework reuse is how to efficiently store, maintain, and look up a generally very large number of reusable components. Several representation and query languages and algorithms have been proposed for this purpose, e.g., by Sahraouim and Benyahia [13]. Even though the problem of component-oriented construction for general software has not yet been sufficiently solved, current industrial practice proves that this approach becomes viable and productive for specific application domains. For example, component-oriented techniques play an important role in constructing current graphical user interfaces, e.g., *Java Beans* [2].

Stewart has shown that similar advantages of *domain-dedication* apply for the use of component-orientation in the design of embedded systems [3]. The notion of making component reuse feasible by focusing on a particular domain is related to the idea of *product lines* as presented in [14]. In this sense, *microCommander* and *microSynergy* is clearly focused on supporting control applications. They might not provide enough freedom for developing other types of embedded system application, e.g., software for cellular phones.

Our approach to connect different application block components is related to work performed in the area of architectural interconnectors, e.g., as presented by Allan and Garlan [15]. The difference to our approach is that it is currently restricted to asynchronous (signal-based) communication only. Furthermore, we deal with a-posteriori integration. Finally, we have chosen SDL for specifying the integration among components. This is in contrast to many other modeling approaches that employ the Unified Modeling Language (UML) for this purpose [8]. We have made this decision because SDL has a formal semantics and is widely used in the embedded systems domain [4].

5. Conclusion

We have presented an approach to component-oriented development of net-centric embedded systems. This approach is based on a graphical composition paradigm of reusable process components. In contrast to traditional component-based development of embedded software, *microCommander* also considers reusable user interfaces for operating and configuring components. These user interfaces can be used to monitor parameters of components from a remote location on the Internet. Furthermore, we have discussed requirements and characteristics for a hierarchical architecture for collaborative networks of embedded controllers. Such networks will play an increasingly important role in our society. We have described an approach for creating flexible mappings between distributed micro controllers based on SDL specifications. This approach has been implemented and evaluated in the *microSynergy* development tool in tight collaboration with our industrial partner Intec Automation Inc. Intec has a keen interest in exploiting our results and integrating them in their integrated development environment. Our future work will include a large-scale case study to evaluate and refine our technique. We have identified a possible candidate for such a study at the Herzberg Institute of Astrophysics at the National Research Council of Canada.

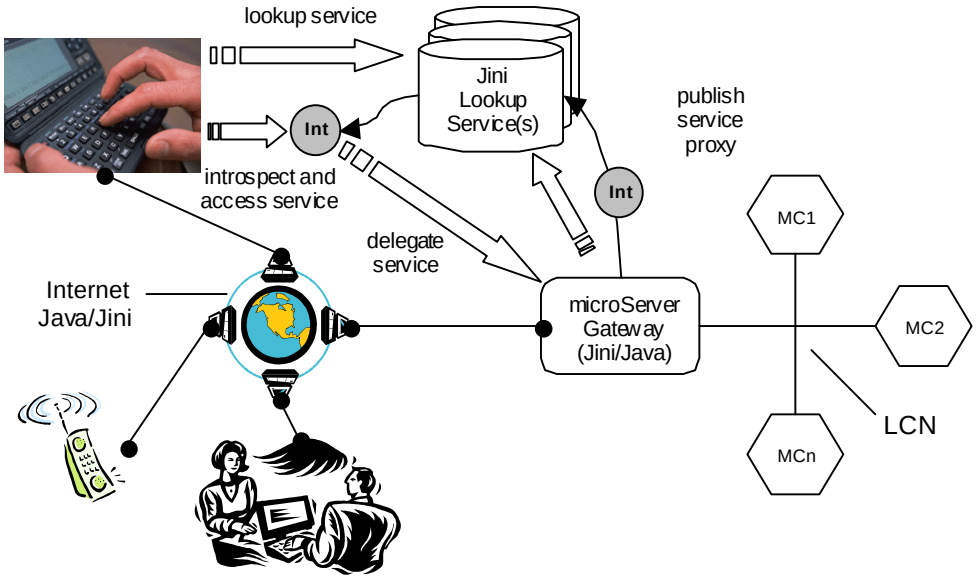

Figure 12. Federation of several LCNs to global networks based on Jini technology

Acknowledgements

We would like to express our gratitude towards the Advanced Systems Institute of British Columbia (ASI) and Intec Automation Inc. for their ongoing support and funding of our research.

References

1. Estrin, D., R. Govindan, and J. Heidemann, *Embedding the Internet.* Communications of the ACM, 2000. **43**: p. 38-50.
2. Deitel, H.M. and P.J. Deitel, *Java : how to program*. 1999, Prentice Hall: Upper Saddle River, N.J.
3. Stewart, D. *Designing Software Components for Real-Time Applications*. in *Embedded System Conference*. 2000. San Jose, CA, USA.
4. Ellsberger, J., D. Hogrefe, and A. Sarma, *SDL - Formal Object-oriented Language for Communicating Systems*. 1997: Prentice Hall Europe.
5. Edwards, W.K., *Core Jini*. 2nd ed. The Sun Microsystems Press series. 2001, Upper Saddle River, NJ: Prentice Hall. xliii, 962.
6. Szyperski, C., *Component Software, Beyond Object-Oriented Programming*. 1997: Addison-Wesley.
7. Polze, A., D. Plakosh, and K.C. Wallnau, *A study in the use of CORBA in real-time settings model problems for the manufacturing domain*. 1997, Carnegie Mellon University, Software Engineering Institute: Pittsburgh.
8. Stevens, P. and R.J. Pooley, *Using UML software engineering with objects and components*. 2000, New York: Addison-Wesley.
9. *Jini Technology and Emerging Network Technologies*. 2001, Sun Microsystems.
10. Nierstrasz, O., S. Gibbs, and D. Tsichritzis, *Component-Oriented Software Development.* Communications of the ACM, 1992. **35**(9): p. 160-165.
11. Leavens, G.T. and M. Sitaraman, *Foundations of component-based systems*. 2000, Cambridge, [England] ; New York: Cambridge University Press. ix, 312.
12. Johnson, R. *Components, frameworks, patterns*. in *1997 symposium on Symposium on software reusability*. 1997. Boston, USA: ACM Press.
13. Mili, H., H. Sahraouim, and I. Benyahia. *Representing and querying reusable object frameworks*. in *Symposium on software reusability*. 1997. Boston, USA: ACM Press.
14. Donohoe, P., *Software product lines : experience and research directions : proceedings of the First Software Product Lines Conference (SPLC1), August 28-31, 2000, Denver, Colorado*. 2000, Boston, MA: Kluwer Academic. xv, 532.
15. Allen, R. and D. Garlan. *Beyond definition/use: architectural interconnection*. in *Proceedings of the workshop on Interface definition languages*. 1994. Portland, USA: ACM Press.