ROBUST SOFTWARE COMPOSITION FOR SENSOR NETWORKS

Péter Völgyesi
Embedded Information Technology Research Group,
Hungarian Academy of Sciences – Budapest University of Technology and Economics,
Magyar tudósok körútja 2, Budapest, 1117, Hungary

Abstract: Sensor networks constitute a relatively new category of systems that necessitates the development and application of novel software engineering techniques. Partitioning the embedded application into several components enables engineers to develop and test isolated software components. The interface specification formalism is of crucial importance in the design and architecture of component-based systems. We propose a hierarchical interface automaton language for the design of embedded software components and applications. The formalism allows the specification of structural, as well as behavioral aspects of components. Our proof-of-concept graphical environment is based on GME, a metaprogrammable modeling framework, while compatibility checks are accomplished by using SPIN, a well-known model checker for reactive distributed systems. Textual PROMELA programs required by SPIN as input are automatically generated from the graphical models.

Keywords: software composition, interface modeling, sensor networks, Interface Automata, temporal logic, model checking

1. Introduction

Wireless sensor networks pose several difficulties to the embedded software developer, which are inherent in the complex interactions between heterogeneous software and hardware modules, the distributed nature of the application and algorithms, the unreliable attributes of the communication channel, the direct exposition to the environment and the requirements of long-run autonomous operation. Most of these problems must be addressed regardless of the actual application, therefore robust composition technologies are need to integrate existing service components and application logic. The way we describe these building blocks and process the captured information is substantial to support isolated component development and testing. Although several languages aim at describing and modeling reactive systems, few of these methods are geared for capturing component interactions within the embedded application.

2. Component-Based Runtime Environments

The advantages of using components stem from the fact that they can be in different applications. Furthermore, building the operating system layers from components makes it possible to include only the services needed by the application conserving precious system resources [3].

TinyOS is a component-based configurable operating system with a very small footprint specifically designed for severely resource constrained devices such as the nodes in a typical networked embedded systems [2]. A TinyOS application is a hierarchical graph of components that is scheduled by a simple FIFO-based non-preemptive scheduler. Components communicate with each other through commands and events, which are executed immediately via function calls. [4].

3. Temporal Models of Component Interfaces

Traditional programming languages and interface description methods capture only the type aspects of software components. The access points of a given component are enumerated along with their accepted and returned parameter types in terms of values and domains. TinyOS and its implementation language *nesC* [4] is no exception to this: component interfaces are defined by a set of function declarations.

Even in trivial applications, the access points of a software component are not isolated, dependencies and complex relationships might impose additional constraints on the use of their services. A component providing communication services may have more sophisticated restrictions that are inherent in the communication protocol. Temporal interfaces aim at capturing these behavioral aspects of software components.

In the following paragraphs the description of a temporal interface language will be given along with the definitions of formal rules of interface compatibility. Our interface modeling language is based on the definition of Interface Automaton [1], which we will reproduce here.

Definition 1 An interface automaton *P* consists of the following elements:

- S_P , a set of *states,*
- S_P^{init} , a nonempty subset of S_P , known as the initial *states,*
- A_P^I , A_P^O and A_P^H , mutually disjoint sets of *input, output* and *internal actions*.

  We denote by $A_P = A_P^I \cup A_P^O \cup A_P^H$ the set of all actions, and
- T_P , a set of steps, where $T_P \subseteq S_P \times A_P \times S_P$.

A simple example of an interface automaton is given in Fig 1. The model describes the interface of the *Comm* TinyOS component, which provides communication services to its clients. The component accepts the *init* and the *sendMsg* input actions and signals the *sendDone* output action.

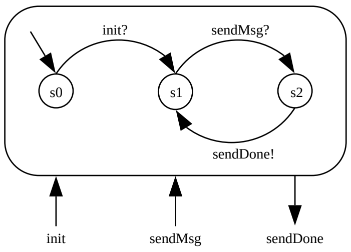

Fig 1 An Interface Automaton

Interface Automata have similar limitations to Finite State Machines: in their flat form both languages have problems with describing complex behavior and state space. We propose additional constructs to the original automata language to overcome these problems.

In embedded applications external events from the physical environment might arrive in any moment regardless of the current state of the application. These external events are propagated through the software components via interrupts and function calls. Therefore, to build compatible components, their interface models need to handle these events in all states resulting in a potentionally large number of steps. Hierarchical states enable us to simplify these possibly incomprehensible models (Fig. 2 and Fig. 3)

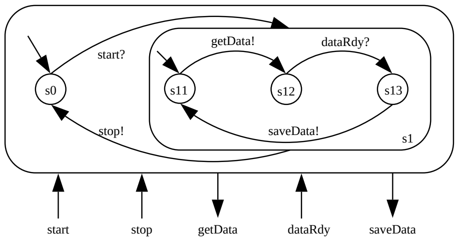

Fig. 2 Hierarchical Interface Automata

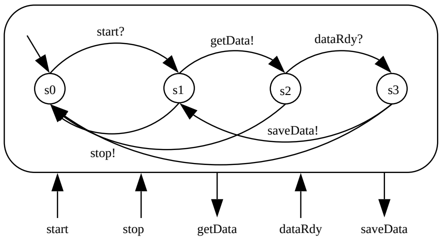

Fig. 3 The same model without using hierarchy

The original Interface Automata language is a superb formalism for specifying interfaces in event driven systems where each component has its own thread of control and the components engage one another asynchronously via events. However, in monolithic embedded applications the assumption of parallel execution and asynchronous message passing no longer hold. Implicit constraints restrict the execution of an automaton that are inherent in sequential execution. Therefore, we have introduced *non-preemptable* states, which enable us to specify atomic action sequences. Non-preemptable states can be implemented in several ways: the most trivial approaches are interrupt masking or the use of mutexes. (Fig. 4 and Fig. 5)

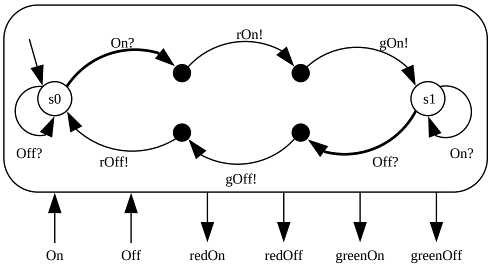

Fig. 4 Interface Automaton with non-preemtable states

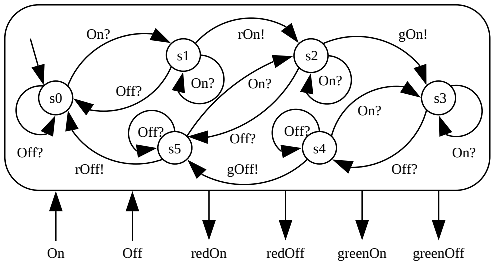

Fig. 5 Interface model using the original automata language

Definition 2 A hierarchical interface automaton *P* consists of the following elements:

- elements of regular interface automata as defined in Definition 1.
- H_P, a set of *hierarchical states*, each of which is a subset of S_P together with a selected element, called the *initial sub-state* of the hierarchical state. Steps originating from hierarchical states are implicitly defined for each contained sub-state. Steps entering into hierarchical states are implicitly defined for the contained initial sub-state.
- S_P^{np}, a set of *non-preemptable* states, $S_P^{np} \subseteq S_P$. The automaton does not accept input actions in non-preemptable states.

We consider two hierarchical interface automata composable if there is no conflict between their actions, thus they only possess common actions which move the product automaton along shared steps.

Definition 3 Two interface automata *P* and *Q* are *composable* if

$$A_P^H \cap A_Q = 0,\ A_Q^H \cap A_P = 0,\ A_P^O \cap A_Q^O = 0,\ \textit{and}\ \ A_P^I \cap A_Q^I = 0,$$

Compatibility analysis must focus on composite states, where one of the original automata initiates a *shared* step, but the other component is not prepared for accepting this action in its respective state [6]. We denote these composite states as *illegal states*. In our previous implementations we identified these illegal state configurations using logic programs or constraint programs [7]. Because the current model checker approach automatically discovers deadlocks and other signs of incompatibility, we do not give the formal definition of compatibility here.

4. Model Checking

Spin is a tool for analyzing the logical consistency of concurrent systems, specifically of data communication protocols. The system is described in a modeling language called

Promela. The language allows for the dynamic creation of concurrent processes. Communication via message channels can be defined to be synchronous or asynchronous.

Given a model specified in Promela, Spin can either perform random simulations of the system's execution or it can generate a C program that performs an efficient online verification of the system's correctness properties. During simulation and verification Spin checks for the absence of deadlocks, unspecified receptions, and unexecutable code. [5]

In the current implementation, compatibility among software components is checked by automatically generated Promela programs from the hierarchical interface automata models. For each interface automaton we generate an individual process in the SPIN framework, which communicates with other processes (interface automata) via synchronous message channels.

5. Case Study

To demonstrate the expressiveness of the hierarchical interface automata language and the benefits of automatic composition checks, we present the visual model of the Surge application as it was modeled using our graphical environment. The application preiodically samples a photo sensor and reports the measured readings to a base station. The communication is based on ad-hoc multi-hop routing over the wireless network, where the nodes organize themselves into a spanning tree rooted at the base station.

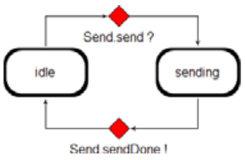

Fig. 6 Model of the MultiHop component's service loop

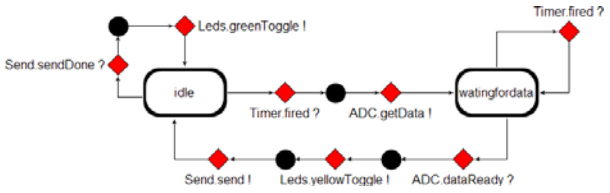

Fig. 7 Model of the faulty Surge component

The temporal models of the multi-hop routing component is shown in Fig. 6. Albeit only the inner service loop is shown, the model presents the restriction of the component clearly: it is not prepared to process multiple requests simultaneously. The Surge component (Fig. 8) overcomes this limitation by waiting for a *Send.sendDone* event before advancing to subsequent states and completing the iteration, thereby facilitating trivial flow control in the system.

The erroneous implementation of the Surge component shown in Fig. 7 differs exactly in this regard. The automaton depicts a typical mistake; it essentially discards an event coming from the multi-hop component. After its first iteration the Surge component may acquire a new sample from the A/D module, while the communication component is still in its *sending* state, where the *Send.send* event is not accepted. This application is unreliable, its operation depends on the timing properties of the data acquisition, periodic timer and task scheduling. This error---a reachable illegal state---is caught by automatic verification. In contrast, manual debugging of similar problems may easily become a time consuming task.

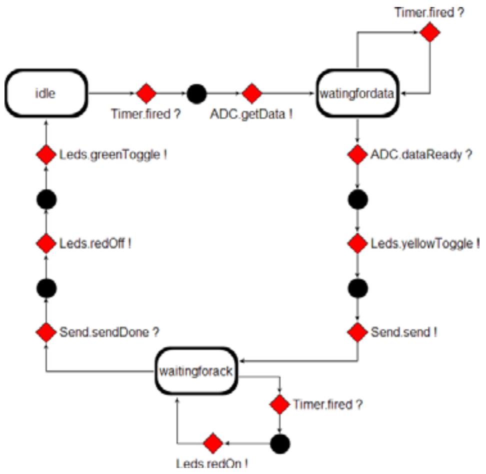

Fig. 8 Interface model of the Surge component

References

[1] de Alfaro, L., Henzinger, T. A.: *Interface Automata.* Proceedings of the Ninth Annual Symposium on Foundations of Software Engineering (FSE), pp. 109-120, ACM Press, 2001.

[2] Hill, J. et al: *System Architecture Directions for Networked Sensors.* Proceedings of ASPLOS, 2000.

[3] Völgyesi, P., Lédeczi, Á.: *Component-Based Development of Networked Embedded Applications.* 28th EUROMICRO Conference (EUROMICRO 2002), Dortmund, Germany, September 4-6, 2002.

[4] Gay, D., Levis, P., von Behren, R., Welsh, M., Brewer, E., Culler, D.: *The nesC Language: A Holistic Approach to Networked Embedded Systems.* Proceedings of Programming Language Design and Implementation (PLDI) 2003, June 2

[5] Holzmann, G. J.: *The SPIN Model Checker: Primer and Reference Manual.* Addison-Wesley, 2003.

[6] Pataricza, A., Bartha, T., Majzik, I., Csertán, Gy.: *Formális módszerek az informatikában.* Typotex, 2004.

[7] Völgyesi, P., Maróti, M., Dóra, S., Osses, E., Lédeczi, Á. and Páka, T.: *Embedded Software Composition and Verification.* ISIS-04-503, February 13, 2004.