Model Based Software Synthesis for Distributed Control Systems and Sensor Networks*

Péter VÖLGYESI[1], Miklós MARÓTI[2] and Ákos LÉDECZI[2]
[1] Embedded Information Technology Research Group,
Hungarian Academy of Sciences – Budapest University of Technology and Economics,
Budapest, Hungary, volgyesi@mit.bme.hu
[2] Institute for Software Integrated Systems,
Vanderbilt University,
Nashville, TN, USA, {miklos.maroti, akos.ledeczi}@vanderbilt.edu

Abstract: Control systems implemented by distributed embedded nodes require additional coordination and management services. Despite the fact that most of these coordination services are similar across different applications the proliferation of embedded platforms renders it almost impossible to provide these software components in a uniform way. We propose a model based embedded programming approach for specification of individual middleware services at the algorithm level that supports automatic code synthesis for different embedded platforms while making sure that only services with compatible interfaces are used. These executable models allow more extensive analysis of temporal behavior, thus provides better support for composition and compatibility checking between software components. The paper also describes the Distributed Services Composition and Synthesis Technology tool, the prototype implementation of our modeling language capable to generate code for three different platforms: a Java based I/O automata simulator, a resource constrained embedded operating system and a CORBA based control platform.

Key words: sensor networks, embedded software, structural I/O automaton

1 Introduction

Global services required by distributed sensor and control systems range from simple event- and time-based coordination to complex algorithms for spanning tree formation, clock synchronization, leader election, protocols for distributed consensus, distributed

* This research was funded in part by the DARPA's Software-Enabled Control Program under AFRL contract F33615-99-C-3611.

transactions and others. These services go beyond the usual capabilities provided by network protocols and they might be originated by different groups with different expertise. Because of resource limitations a complex monolithic middleware layer that contains all services for all applications is not feasible. Moreover, the wide range of programming languages and component frameworks necessitates the repeated implementation of the same algorithms. To meet these challenges new software engineering techniques and tool support is required that enable the decoupling of embedded software design from implementation concerns and support composition and run-time compatibility checking of software components.

Different levels of abstraction can be used to describe an algorithm. This level of abstraction determines the strength of the language considering its verification, composition and platform independent capabilities. Our highly abstract structural automata language merges the formalisms of *executable models* (implementabilty), *interface automata* (optimistic approach to composition) [Henzinger 2001], and *I/O automata* (nondeterministic execution order).

2 Structural I/O automaton

Our notation and definitions are based on the I/O automata language [Lynch 1997]. First, we give the general definition of a structural I/O automaton, than we will introduce basic automata serving as building blocks along with compositional operations to obtain more complex automata. A *structural I/O automaton A* consists of eight components:

- *acts(A),* a structured set of *actions*
- *states(A),* a structured set of *states*
- *start(A),* a nonempty subset of *states(A),* known as *start states*
- *trans(A),* a *state-transition relation,* where $trans(A) \subseteq states(A) \times acts(A) \times states(A)$
- *in(A),* a set of data ports of the set *acts(A)*
- *out(A),* a set of data ports of the set *acts(A)*
- *data(A),* a set of data ports of the set *states(A)*
- *tasks(A),* a *task partition* on *acts(A)*.

The scope and space limitations of the current publication do not allow to cover the full language in detail, the rest of the paper aims at giving an informal description of the concepts above. For precise mathematical definition of structural I/O automaton see [Maróti 2003].

Actions represent signals – with optional parameters – accepted or generated by the automaton. The input actions are generated by the environment and transmitted instantaneously to the automaton. An automaton is not able to somehow "prevent" input actions from occurring.

The collection of all possible configurations of an automaton is represented by its state space, *states(A)*. The initial configuration within this space is described by *start(A)*. Just like with Finite State Machines (FSM), the original I/O automata have key weakness in their simple flat form: the number of states and the state-transitions can get quite large even for moderately complex systems. Such models quickly become chaotic and

incomprehensible when one tries to understand abstract I/O automata, hence we describe the sets of states and actions in structured form. Structured sets are defined recursively:

- The domain of basic data types are structured sets
- Finite products of structured sets are structured
- Disjoint unions of structured sets are structured
- Finite powers of structured sets are structured
- The Kleene[1] star of structured sets is structured

We classify the structured sets into five types, called *variables, products, unions, arrays* and *queues*, according to the above rules, respectively.

To overcome the problem of accessing state variables and action parameters in deeply nested data structures we introduce *data ports*. The purpose of these ports is twofold. First, it provides controlled access to the data structure, exporting only relevant parts of it. Second, a port may exercise specific constraints on its supervised state space. A data port can be read if these constraints are satisfied by the current configuration of the state space. Writing to the data port is always possible, however – as a side effect of the write operation – the port will bring the current state of the automaton to fulfill all defined constraints.

An action can accompany a *state transition*, where upon the transition a shift is made to a new state. To achieve platform and programming language independence, we limit the complexity of the specification of these transitions:

transition (*arg1, … , arg*$_n$)
Precondition:
$cond_1$ (par_1, … , par_n, var_1, … , var_m)
…
$cond_k$ (par_1, … , par_n, var_1, … , var_m)
Effect:
$var_{m+1} = expr_1$ (par_1, … , par_n, var_1, … , var_m)
…
$var_{m+l} = expr_l$ (par_1, … , par_n, var_1, … , var_m)

where par_1, … , par_n are action parameters and var_1, … , var_{m+l} are state variables accessed through data ports as described above, $cond_1$, … , $cond_k$ are simple comparisons (the allowed relations are: =, <, > or <>) and $expr_1$, … , $expr_l$ are basic expressions using only simple arithmetic operators. These simple conditions and basic expressions enable us to generate source code for a wide range of programming languages.

The most basic I/O automaton is the *Variable*, which can store a single value of a simple data type T. The state space of the variable is therefore $\{T\}$, that is accessible through the data port, and the start state is one possible value of the data type.

To add new state transitions to an automaton we use the *Activator* operation, which may introduce new actions and action data ports. The *Product* and *Union* operations enable us to compose a finite list of existing automata into one. We also defined an *Array* operation as a shorthand notation of repeated use of the product operation.

[1] The union of all finite powers of a set: $\{0\} \cup A \cup A^2 \cup \ldots$

3 Case study

The Distributed Services Composition and Synthesis Technology (DISSECT) tool is our prototype implementation of the structural I/O automata language and provides a design environment for embedded software development. The tool was constructed by configuring the Generic Modeling Environment (GME) framework with the *metamodel* of the language. The metamodel is a set of syntactic, semantic and presentation information on the concepts of our domain, the relationships among them, and how these concepts should be visualized [Lédeczi 2001].

To demonstrate the expressiveness of the language we have built a highly distributed tracking application running on the UC Berkeley mote platform [Hill 2000], where a homogenous network of intelligent sensors attempts to localize a moving beacon. The beacon periodically broadcasts a radio message and emits a sound signal for half a second. Those sensors whose microphones picked up the sound, compute a distance to the beacon based on the measured time of flight of the sound signal. Some of the components in the model represent operating system components, only the interfaces of these are modeled. The most important component of the system is the *tblmgr* automaton, which maintains a table containing the latest measurement results of all trackers in the network. This table is updated if the node itself has measured a new distance or one of its neighbors has sent an update ticket. All nodes should broadcast their table changes periodically in update tickets. The source code of the entire tracking application is automatically generated, and ready to run on the embedded sensors.

The provided middle service is not specific to acoustic tracking, but can be used in other applications as well – like structural vibration damping – where information spreading must be implemented. We believe that the formal specification of these generic services enables us to use them as safe building blocks in future applications.

References

DE ALFARO, L., HENZINGER, T.A., 2001. Interface Automata. In *Proceedings of the Ninth Annual Symposium on Foundations of Software Engineering (FSE)*. ACM Press, 2001, pp. 109-120.

LYNCH, N.A., 1997. *Distributed Algorithms*. Morgan Kaufmann Publishers, 1997, ISBN 1558603484

HILL, J., SZEWCZYK, R., WOO, A., HOLLAR, S., CULLER, D., PISTER, C., 2000. System architecture directions for network sensors. *ASPLOS 2000*, Cambridge, November 2000.

LÉDECZI, Á., BAKAY, Á., MARÓTI, M., VÖLGYESI, P., NORDSTROM, G., SPRINKLE, J., KARSAI, G., 2001. Composing Domain Specific Design Environments, In *IEEE Computer*, November 2001, pp. 44-51.

MARÓTI, M., VÖLGYESI, P., SIMON, G., KARSAI, G., LÉDECZI, Á., 2003. Distributed Middleware Services Composition and Synthesis Technology, *IEEE Aerospace Conference*, Big Sky, MT, USA, March 2003.

VÖLGYESI, P., LÉDECZI, Á., 2002. Component-Based Development of Networked Embedded Applications, *28th Euromicro Conference*, Dortmund, Germany, September 2002.