From Statecharts to ESP*: Programming With Events, States and Predicates For Embedded Systems

Vugranam C. Sreedhar and Maria-Cristina Marinescu
IBM TJ Watson Research Center, Hawthorne NY 10532

ABSTRACT

Statecharts are probably the most popular mechanism for behavior modeling of embedded system components. Modeling a component involves using a mainstream language for features that statecharts cannot express: detailed behavior of conditions and actions, object-orientation and distributed computing features. Debugging is done at the level of the generated native code. Rather than treating statecharts as a separate programming model from the native programming model, we extend a (Java-like) language with support for key concepts of statecharts: (1) explicit *states*, (2) asynchronous *events*, and (3) conditional execution. This paper presents ESP*, a language that supports statecharts and a set of other advanced programming concepts to make programming embedded systems easier. The paper also shows how to translate statecharts to ESP*.

Categories and Subject Descriptors: D.2.2 [Software Engineering]: Design Tools and Techniques

General Terms: Languages, Design.

Keywords: Statechart, Multiple Classification, Predicate Dispatch

1. INTRODUCTION

The electronic industry has entered a new era with the convergence of three critical areas: computers, consumers, and communication. The amount of electronic content and the complexity of the embedded software in automobiles, planes, medical devices and other products that have traditionally developed around mechanical product innovation is increasing exponentially. A key complexity faced by these industries is the ability to model the behavior of large and complex embedded system components.

Statecharts and related state machines such as State Diagrams are probably the most popular approach for modeling embedded system components. Modeling an embedded component involves using: (1) Statecharts to describe the state machine and (2) a mainstream language (C, C++, Java) to describe more detailed behavior such as the statechart actions and conditions. The dual programming model forces the end user to master both statecharts and the native programming model. Moreover, because there is very little support for debugging statechart models, end users must debug the generated native code. Finally, since statecharts do not directly support useful features of object-orientation and distributed computing, end users must rely on the native programming model for such features. Rather than treating statecharts as a separate programming model from the native programming model, we start from a Java-like language and bring some of the key concepts of statecharts as first-class elements within our language. Statecharts bring together three important concepts: (1) the notion of explicit and hierarchical states, (2) conditional execution of actions, and (3) asynchronous events. ESP* reifies these concepts and treats them as first-class elements of the language. ESP* provides a unified programming language that supports not only statechart concepts, but also advanced programming language concepts inspired from existing work on predicate dispatch, multiple and dynamic classification [5, 9], and typestates.

ESP* is intended to be a language for modeling, analyzing and implementing embedded components, and supports legacy Statechart models via straightforward translation. ESP* was motivated by the problems that we faced during our consulting engagements in industry. One of the main difficulties faced by our clients is the "out of synch" that arises between the code generated from statecharts and the original statechart model when the code is modified after it is generated. Incidentally, debugging and performance tuning of embedded application is almost always done on the generated code rather than on the statechart model. In this context, we believe that our approach has several advantages: (1) The user does not need to master two different programming models, (2) We can provide better debugging and software maintenance capabilities if the application is written using a single programming model. (3) We directly support features of object-orientation, concurrency and strong typing. (4) A language such as ESP* can help end users with the acceptance of modeling, replacing a documentation burden with a unified programming model that can simplify the implementation of embedded software.

Section 2 introduces a simple example. Section 3 introduces charts and chart-states. Section 4 presents predicate methods and predicate dispatch. Section 5 discusses asynchronous programming in ESP*. Section 6 presents a simple translation scheme from statecharts to ESP* programs. Section 7 compares ESP* with related work.

2. EXAMPLE

This section presents the Statechart specification and the requirements of a simple coffee vending machine (CVM). The CVM that we use as our running example is expected to at least: (1) Not dispense drinks if the customer has not deposited enough money. Change is returned after the coffee is dispensed. (2) Not dispense the drink if a cup is not in place. (3) Accept drink selection. The CVM has three buttons for selecting one (or more) of the following types of drinks: `CAFE`, `DECAFE`, and `CHOC`. If a customer presses more than one button and the particular combination is not allowed, no drink is dispensed and CVM returns the money. Since it is not obvious what size, composition or price a mix of drinks should have if allowed, there must be an explicit specification for each possible drink mix. For instance, `CAFE` and `CHOC` result in a standard size cup of mocha—a 50-50% mix of coffee and chocolate with a price of 85 cents.

Figure 1 illustrates a hierarchical statechart for a subset of the CVM requirements. A statechart consists of a finite collection of states and transitions. Throughout this paper we use the original statechart model. We assume that statecharts are "normalized" according to Sekerinski and Zurob [11]. A state can be either (1) a *simple state* which cannot be further decomposed, (2) a *composite state* which

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
EMSOFT'05, September 19–22, 2005, Jersey City, New Jersey, USA.
Copyright 2005 ACM 1-59593-091-4/05/0009 ...$5.00.

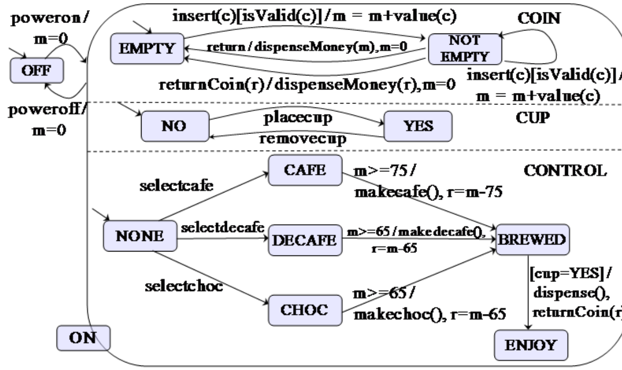

Figure 1: Statechart for a simple coffee vending machine

can be further decomposed into nested states, (3) an or-state, or (4) an and-state. A *transition* is a labeled binary relation between states. A *transition label* is a triplet $(e[c]/a)$, where e is an *event*, c is a *condition*, and a is a *set of actions*. A transition $s_1 \stackrel{e[c]/a}{\rightarrow} s_2$ is *enabled* in state s_1 if when the event e occurs in s_1 the condition c is `true`. Once enabled, the transition takes place and the new state is s_2. The set of actions a is executed as part of the state transition.

The statechart in Figure 1 has two states: one for power off (OFF) and one for power on (ON). The ON state consists of three and-charts: `COIN`, `CUP`, and `CONTROL`. `COIN` handles state changes as effect of inserting a coin, asking for a money refund, and for change being returned after the drink is delivered. `CUP` keeps track of whether there is a cup in place for dispensing the drink. `CONTROL` allows the user to select a drink type. When there is enough money the CVM brews the requested drink. If the cup is in place the CVM dispenses the coffee and returns the change. Notice that the charts can create events as result of a transition, which are consumed by other charts.

3. THE ESP* LANGUAGE

An ESP* program consists of classes, predicate methods, charts and chart-states. Classes describe data that objects will have at runtime, are similar to OO classes, and may `have` charts associated with them.

Charts and Chart-States: A *chart* has a name and a fixed set of possible *chart-states*. A chart uses chart-states to specify optional, mutable information that instances of a class may acquire and lose during program execution. Charts are defined outside the class hierarchy, can be associated with and shared by multiple classes, but cannot exist independently. Charts and chart-states are conceptually different from class fields and `enum` types and are closely related to algebraic data types. Charts externalize and explicitly reify transient internal states of an object. This is useful in (1) reducing the number of classes, (2) ensuring that the sub-class mechanism is used for substitutability, and (3) allowing us to take advantage of finite state machines concepts to explicitly control what the application code is allowed to do via protocols.

Exclusive and Subset Charts: ESP* charts can be either exclusive or sub-set charts. An *exclusive chart* can be in only one of its chart-states at a time: a `Switch` may be `ON` or `OFF`. A *sub-set chart*, on the other hand, can be in any sub-set of the chart-states at a time. `CoffeeMachine` in Figure 2 is classified from both `PowerSwitch` and `CoffeeType` points of view. `PowerSwitch` is an exclusive chart with two chart-states: `ON` and `OFF`. `CoffeeType` is a sub-set chart declared via the `&=` symbol. At any instance, `CoffeeType` can be in any sub-set of the declared chart-states. If the object constructor does not initialize a chart, the initial state of that chart is by default `Start`. Method `selectMocha()` shows a transition to a sub-set of the possible chart-states.

Entry and Exit Methods: When we turn CVM on we may want some indication that the power is on, such as a light indicator. This

```
chart PowerSwitch = {ON, OFF}
chart CoffeeType &= {NONE, CAFE, DECAFE, CHOC}
class CoffeeMachine has PowerSwitch, CoffeeType {
  int money  = 0 ; CoffeeMachine(...) {...} // constructor
  void powerOn() { PowerSwitch = ON; Control = NONE};
  void powerOff() { PowerSwitch = OFF};
  void selectMocha() { if PowerSwitch = ON
           then CoffeeType = CAFE&CHOC};
  void makeCoffee() when CoffeeType = CAFE && money >= 75
           {CoffeeType = BREWED}    /*1*/
  void makeCoffee() when CoffeeType = DECAFE && money >= 65 {...}
  void selectCafe {CoffeeType = CAFE ; } ...}
```

Figure 2: Subset Charts and Predicate Methods

```
chart Coin = {EMPTY, NOTEMPTY}; chart Cup = {YES, NO}
chart Control &= {NONE,CAFE,DECAFE,CHOC, BREWED,ENJOY}
     with {NONE -> CAFE; NONE -> DECAFE; NONE -> CHOC;
     NONE -> CAFE&CHOC; CAFE -> BREWED;
     DECAFE -> BREWED; CHOC -> BREWED;
     BREWED -> ENJOY; CAFE&CHOC -> BREWED}
chart PowerSwitch = {ON has (Coin, Cup, Control), OFF}
class CVM has PowerSwitch{ CVM() {} // empty constructor
  void   powerOn() { PowerSwitch = ON; Control = NONE};
  void powerOff() { PowerSwitch = OFF};
  void selectMocha() { PowerSwitch.ON.Control = CAFE&CHOC};
  void brewBeverage() { PowerSwitch.ON.Control = BREWED}}
```

Figure 3: ESP* specification of CVM

extra behavior can be coded in methods as part of taking a transition between chart-states. If entering (leaving) a chart-state always requires this behavior, the user must replicate the behavior in each method that enters (leaves) the chart-state. An alternative is to borrow the concept of *entry* and *exit* methods from statecharts and associate such methods with chart-states. `PowerSwitch` becomes:

```
chart PowerSwitch =
   {ON {entry(){system.lightOn();}},
   OFF {entry(){system.lightOff();}}}
```

When a chart enters or leaves a chart-state it must also invoke the corresponding `entry()` or `exit` method. If the chart-state is of type sub-set—e.g. `CoffeeType = CAFE&CHOC`—then it must invoke the methods for each of the basic chart-states in the subset. One can pass parameters to an `entry()` method, but neither `entry()` nor `exit()` methods can return values. The `entry` method can be overloaded.

Hierarchical Charts: Consider the statechart shown in Figure 1. At the top level, the statechart hast two or-states: `ON` and `OFF`. `ON` consists of three and-states: `COIN`, `CUP`, and `CONTROL`. Each of the three and-states are made of a set of or-states. We can model the hierarchical statechart using hierarchical chart-states as shown in Figure 3. `PowerSwitch` has two states: `ON` and `OFF`. The `ON` chart-state has three orthogonal sub-charts: `Coin`, `Cup`, and `Control`. By default, when an object `cvm` of type CVM comes to existence, all its charts—including those hierarchically defined— are initialized to `Start`. Calling `selectMocha()` on `cvm` will raise an error because `Control` can be updated only when the `PowerSwitch` is `ON`. In general it is impossible to say whether an erroneous path is realizable.

Protocols: In statecharts one can enforce the kinds of transitions that are allowed within the model via transition edges. In ESP* we can define allowable transitions in a chart using protocols. For the `Control` sub-chart of `CVM` in Figure 3, the only sub-set chart-state allowed is `CAFE&CHOC`. Consider now the following piece of code:

```
CVM cvm = new CVM(); cvm.powerOn() ; /* 1 */
cvm.brewBeverage() ; // error!
```

The chart-state of `PowerSwitch` after `/*1*/` is `ON`, and the chart-state of `Control` is `Start`. Invoking `cvm.brewBeverage()` raises an error; according to the protocol, it is illegal to go from chart-state `Start` to `BREWED` in sub-chart `Control`. It is in general undecidable to statically determine whether a piece of code violates a protocol.

Constraints: Unlike Statecharts, ESP* allows to explicitly specify what relationships must hold and what relationships are forbidden between chart-states of different charts. For CVM in Figure 3 we can define a constraint that says that if chart `Control` is in chart-state `CAFE`, chart `Coin` cannot be in chart-state `EMPTY`:

```
constraint CoinControl (Coin c, Control ctrl) {
    ctrl = CAFE => c != EMPTY }
```

4. PREDICATE DISPATCH

ESP* uses predicate methods and predicate dispatch to express conditional execution in statecharts. Methods can be predicated with a when-expression. Each when-expression is constructed using chart-states and object field values. A predicate method is a method which can be invoked only when its when-expression evaluates to `true`. The method lookup for correctly invoking `p.foo()` has two steps:

- **1.** We first determine the run-time type `T` of the object addressed by `p` and look for the method `foo` in the receiver class `R`, where `R` is either `T` or the closest ancestor class of `T` where `foo()` is defined. During this step we only use the method name and signature when searching for method `foo()`.
- **2.** We then take the when-expression into consideration. (a) If there is only one method `foo()` defined in the receiver class `R` whose when-expression evaluates to `true` then we invoke that method. (b) If there is more than one such method `foo()` then we pick the method with the most specific when-expression. A when-expression e_1 is more specific than a when-expression e_2 if e_1 implies e_2 and e_2 does not imply e_1. (c) If none of the when-expressions of the methods `foo()` defined in the receiver class `R` evaluates to `true`, we recursively continue up the class hierarchy and apply step 2 until a method `foo()` is found.

`p.foo()` can raise two kinds of method-lookup errors: (1) A "method not found" error happens if either `foo()` is not defined in any of the receiver classes, or `foo()` is defined, but none of the when-expressions is satisfied by the runtime state of the object pointed to by `p`. (2) An "ambiguous method" error occurs if the when-expressions of more than one method `foo()` in a receiver class are satisfied, and none of these expressions is more specific than all the others.

We can make ESP* type sound if we guarantee no "method not found" and no "ambiguous method" errors at runtime. Both tests depend on proving method predicate implication. If the predicates only test chart-state information the implication problem is decidable (and NP-complete). Similar to Ernst, Kaplan and Chambers [5], we treat arbitrary boolean-valued expressions as black boxes, and we consider two such expressions equivalent if their canonical forms are structurally equivalent. This makes predicate implication NP-complete. We can use a SAT solver to solve the NP problems.

5. ASYCHRONOUS PROGRAMMING

We may not always want to wait until an event is processed before executing other transitions. This approach is particularly useful for systems that are concurrent and/or distributed. We can model the transition condition c as the method predicate expression, and the event, action, or event/action pair as the method itself. For example, we have been modeling the user event `selectCafe` as `void selectCafe {CoffeeType = CAFE}`, and the action `makeCoffee()` as `void makeCoffee() when CoffeeType = CAFE && money >= 75 {...}`.

ESP* uses asynchronous methods to model *asynchronous events* in statecharts. The occurrence of an event can trigger more than one state transition in statecharts that contain concurrent and-statecharts. ESP* supports parallel dispatch to successfully model such and-statechart semantics. ESP* `async` methods can be predicated using when-expressions. A synchronous method and an asynchronous method cannot have the same name and method signature. If a method is declared asynchronous (or synchronous) in a class C, none of the children classes of C can override that method with a synchronous (or asynchronous) method. The constructor methods of a class cannot be asynchronous.

Asynchronous Predicate Dispatch: Consider the `CUP` statechart in Figure 1. When a customer places a cup he generates a `placecup` user event, which triggers a state transition of `CUP` from `NO` to `YES`.

```
chart CupTest = {NO, YES}
class CoffeeMachine has CupTest  {
   async void placeCup() {CupTest = YES ; }
   boolean isCupPlaced() {return CupTest = YES ; }
CoffeeMachine cm = new CoffeeMachine(CupTest=NO) ;
cm.placeCup() ;
if(cm.isCupPlaced()) { ... }
```

The keyword `async` defines an asynchronous method. When an `async` method is invoked a new thread of computation is created and the caller continues with the rest of the computation immediately after the dispatch. Therefore the test `isCupPlaced()` may lead to a race condition when accessing `CupTest`. In statecharts on the other hand, the transition to state s via `cm.placeCup()` is not finished until `CupTest = YES`. Testing `cm.isCupPlaced()` occurs in s, and therefore cannot happen until the previous transition has taken place. Making the code match the intended behavioral semantics of a statechart specification implies the introduction of a mechanism to avoid race conditions. One straightforward way to do that is to (1) make all asynchronous methods synchronized (a la Java) and (2) introduce `synchronized` operations to make a synchronous method wait for the modifications performed by the asynchronous method to take place: `boolean isCupPlaced() {synchronized(this) {return CupTest = YES;}}`

Futures: By default, the return type of all `async` methods are of `future` type. Since the control returns immediately after an asynchronous dispatch we use `futures` as a placeholder for storing the "future" return value of an asynchronous computation. Consider the following piece of code for the example in Figure 2 enhanced with the method `async boolean orderCoffee() {return CoffeeType = CAFE;}}`:

```
CoffeeMachine cm = new CoffeeMachine(CoffeeType=NONE) ;
cm.selectCafe() ;
future boolean f = cm.orderCoffee() ;
if(f = true) { ... }
```

Invoking `cm.orderCoffee()` tries to lock `cm`. If it succeeds, it creates a new thread and immediately returns the control to the current thread. The return value from the dispatch will be stored in `future` variable `f`. When the caller thread attempts to access the future value of `f`, if the asynchronous thread has not finished executing, the caller thread blocks. A `return` statement in an `async` method does *not* transfer the control back to the caller thread—it simply terminates the current thread after storing the value in the `future` variable designated by the caller thread. This will awake all the threads blocked waiting to access `f`. Asynchronous methods are closely related to Java threads, although Java does not directly support futures. An asynchronous method can be implemented as a `run` method.

6. TRANSLATING STATECHARTS TO ESP*

Consider the *flat statechart* S shown in Figure 4. Disregard for the moment that it is part of hierarchical statechart T. S has two or-states: $s1$ and $s2$. We assume that every statechart has a unique name which we use to form the name of the class in the ESP* program. Therefore we map S to class `SClass`. We map each statechart to a chart and a set of chart-states—we therefore create a chart named `SChart`. States $s1$ and $s2$ of S are mapped to chart-states in `SChart`. For each transition edge in a statechart we map the event to a predicated `async` method and the action to a predicate synchronous method. For S we generate the following ESP* code:

```
chart SChart = {s1, s2}
class SClass has SChart {
   async void E () when SChart = s1 { as(); }
   void as() when cs { ...; SChart = s2; } }
   async void E () when true {}
   void as() when true {} }
```

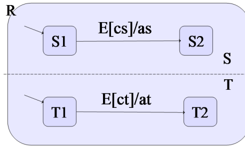

Figure 4: Statechart with and-states

We encode an event E as a predicate asynchronous method `E() when SChart=S1`. Next we map the action as to synchronous predicate method `as() when cs`. We also update the state transition in method `as()`. According to statechart semantics, when E occurs and the current state of S is not $s1$ we simply drop the event. Such dropping of events also happen when the condition cs is `false`. According to ESP*ś method lookup, when the predicate associated with `E()` or with `as()` is `false` we raise "method not found" error. To avoid this mismatch we generate two additional side-effect free, empty `when true` methods that will succeed if the existing ones fail.

Hierarchical Statecharts: Consider the flat statechart CVM in Figure 1 with two or-states: `ON` and `OFF`. State `ON` is a composite state that contains three and-states: `COIN`, `CUP` and `CONTROL`. We first define a `class CVM` and a `chart PowerSwitch` for statechart CVM. We must also define `ON`'s chart and chart-states. The result is the specification in Figure 3. Notice that we use the has construct for denoting children of a composite state that are and-states.

Concurrent Events: Consider the hierarchical statechart shown in Figure 4. Composite state R contains two and-states: S and T. Assume that the event E occurs when S is in state $s1$ and T is in state $t1$. According to the statechart semantics both transitions $s1 \rightarrow s2$ and $t1 \rightarrow t2$ will occur (assuming that the corresponding conditions cs and ct are `true`). The following translation from Figure 4 illustrates the semantic mismatch between method lookup in ESP* and concurrent transitions in statecharts with and-states. S and T are states in R—not statecharts—so no class is generated for them.

```
chart SC = {s1, s2}; chart TC = {t1, t2}
chart RC = {R has (SC,TC)}
class RClass has RC {
  RClass() { RChart = R; RC.R.SC = s1; RC.R.TC = t1;}
  void as() when cs { RC.R.SC = s2;}
  void at() when ct { RC.R.TC = t2;}
  async void E() when RC.R.SC = s1 {as();}
  async void E() when RC.R.TC = t1 {at();}...}
```

The translation defines two `async` methods for event `E`, with different when-expressions. The dispatch `rc.E()` for an object `rc` of type `RClass` is ambiguous since the predicates of both methods `E` evaluate to `true` and neither is most specific. A solution is to analyze `S` and `T` and, because they are independent and-charts, define a single method `E` that simply calls both `as()` and `at()`: `async void E() when RC.R.SC=s1 && RC.R.TC=t1 {as(); at();}}`. Ideally, `as()` and `at()` can be concurrently executed, but ESP* does not currently support parallel dispatch. Alternatively we can make both `as()` and `at()` asynchronous methods.

7. DISCUSSION AND RELATED WORK

Harel introduced statecharts to overcome the limitations of conventional finite state machines [8]. Due to their popularity, statecharts in many semantic variations [14], [3], [6] are part of many modeling tools. The Object Management Group (OMG) has standardized state diagrams as part of the UML (http://www.uml.org). Niaz and Tanaka [10] present an approach to generate Java code from UML state diagrams. In QHSM, Samek introduces Quantum Hierarchical State Machines to represent state hierarchy and efficiently implement transition dynamics. ESP* was influenced by our earlier work on multiple and dynamic classification and by predicate dispatch. To our knowledge, neither classifications mechanisms nor predicate dispatch have been explored as a way to model statecharts.

nesC [7] is a language for network embedded systems that supports asynchronous event-driven execution, a flexible concurrency model, and component-oriented application design. galsC (and TinyGALS) is a globally asynchronous, locally synchronous model for event-driven embedded systems [4]. Unlike galsC, ESP* supports both event-based and state-based programming. The TinyGALS programming model is very similar to the connection-oriented software architecture model [12]. The model of computation in languages such as Esterel [2] and Signal [1] is synchronous. The concurrency can be compiled away, and the system behaves like a state machine at run time. Ptolemy II (http://ptolemy.eecs.berkeley.edu/ptolemyII/) is a framework that supports many models of computation.

Craig Chambers and his group introduced predicate classes (PC) and predicate dispatch (PD) [5, 9]. PC only use internal states of classes to evaluate predicate expressions. ESP* can also use external charts and chart-states. ESP* supports protocols on charts to restrict the kinds of state updates that are legally allowed on charts. In PC such restriction have to be programmed as predicate expression. JPred [9] extends PD for Java. The predicate dispatch in ESP* is similar to PD. ESP* is an extension of some of our previous work with focus on modeling of embedded systems.

The notion of chart-states has some relation to typestates [13]. DeLine and Fahndrich extend the classical type-state mechanism for objects. Relations between type-states of fields of different classes cannot be expressed. Also, it is a type error to have more than one method in a class with the same name and signature, but different pre-conditions. Foster et al. present a mechanism to add type qualifiers as a first class concept in C.

8. CONCLUSION

This paper presents ESP* a language that reifies key features of statecharts and a set of other advanced programming concepts, and expresses them as part of a Java-like language. ESP* provides support for: (1) explicit *states*, (2) asynchronous *events*, and (3) conditional execution. The paper also shows how to translate statecharts to ESP*.

9. REFERENCES

[1] A. Benveniste, B. Caillaud, and P. Le Guernic. From synchrony to asynchrony. In *CONCUR'99, LNCS 1664, Springer*, pages 162–177.

[2] G. Berry. *The Foundations of Esterel*. MIT Press, 2000. Editors: G. Plotkin, C. Stirling and M. Tofte.

[3] D. Bjorklund, J. Lilius, and I. Porres. Towards efficient code synthesis from statecharts. In *Practical UML-Based Rigorous Development Methods, Lecture Notes in Informatics*, 2001.

[4] E. Cheong, J. Liebman, J. Liu, and F. Zhao. TinyGALS: A programming model for event-driven embedded systems. In *Proceedings of the 2003 ACM Symposium on Applied Computing*, pages 698–704, 2003.

[5] M. Ernst, C. Kaplan, and C. Chambers. Predicate dispatching: A unified theory of dispatch. In *ECOOP '98*, LNCS 1445.

[6] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements od Reusable Object-Oriented Software*. Addison-Wesley Publishing Company, New York, NY, 1995.

[7] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesc language: A holistic approach to networked embedded systems. In *Proceedings of PLDI '03*, 2003.

[8] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[9] T. Millstein. Practical predicate dispatch. In *Proceedings of OOPSLA '04*, Vancouver, British Columbia, 2004.

[10] I. A. Niaz and J. Tanaka. Mapping uml statecharts to java code. In *Proceedings of the IASTED International Conference on Software Engineering (SE 2004)*, pages 111–116, 2004.

[11] E. Sekerinski and R. Zurob. iState: A statechart translator. In *UML 2001*, LNCS 2185, pages 376–390. Springer-Verlag, 2001.

[12] Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice-Hall, 1996.

[13] R. Strom and S. Yemini. Typestate: a programming language concept for enhancing software reliability. *IEEE TSE*, 12(1), jan 1986.

[14] M. von der Beeck. A comparison of statecharts variants. In *Proceedings Formal Techniques in Real Time and Fault Tolerant Systems, Springer, LNCS 863*, pages 128–148, 1994.