Fusing Statecharts and Java

MARIA-CRISTINA MARINESCU, Computer Science Dept., Universidad Carlos III, Leganés, Spain
CÉSAR SÁNCHEZ, IMDEA Software Institute, Spain and Institute for Applied Physics, CSIC, Spain

This paper presents FUSE, an approach for modeling and implementing embedded software components which starts from a main-stream programming language and brings some of the key concepts of Statecharts as first-class elements within this language. Our approach provides a unified programming environment which not only preserves some of the advantages of Statecharts' formal foundation but also directly supports features of object-orientation and strong typing. By specifying Statecharts directly in FUSE we eliminate the out-of-synch between the model and the generated code and we allow the tuning and debugging to be done within the same programming model. This paper describes the main language constructs of FUSE and presents its semantics by translation into the Java programming language. We conclude by discussing extensions to the base language which enable the efficient static checking of program properties.

Categories and Subject Descriptors: C.2 [**Special-purpose and application-based systems**]: Real-time and embedded systems; I.6.5 [**Simulation and Modeling**]: Model Development; D.2.2 [**Software Engineering**]: Design Tools and Techniques; D.3.3 [**Language Constructs and Features**]: Control Structures

General Terms: Languages, Design, Theory

Additional Key Words and Phrases: Embedded Systems, Programming Languages, Modeling, State-charts

ACM Reference Format:
Marinescu, M. C., and Sánchez C. ACM Trans. Embedd. Comput. Syst. X, X, Article XX (March 2012), 20 pages.
DOI = 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

1. INTRODUCTION

One of the most popular approaches to modeling embedded software components outside the academic world are Statecharts and related state machine abstractions. Different types of state machines are better suited for different kinds of embedded devices. For example, Activity Diagrams are appropriate when state changes are a result of the completion of internal operations rather than the occurrence of an asynchronous external event. Statecharts, on the other hand, are well suited to the opposite scenario in which transitioning from one state to another happens as result of an event. Additionally, Statecharts overcome limitations of other state machine formalisms, such as the complexity of modeling and the difficulty in modeling concurrent and distributed systems. These are serious limitations for concurrent, real-time embedded systems such as those used in the avionic, automobile, and medical industries. Not only must these devices react timely to events, but they need to do it in a reliable, correct manner. Other formalisms exist that allow the specification of communicating concurrent processes such as CSP and its offshoots. In fact, for hard real-time applications Timed

Author's addresses: Maria-Cristina Marinescu, Computer Science Dept., Universidad Carlos III de Madrid, Leganes, Spain; César Sánchez IMDEA Software Institute, Spain and Institute for Applied Physics, CSIC, Spain.

DOI 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

CSP is usually a more suitable choice than most Statecharts implementations since these do not usually have a notion of measured time. On the other hand, Statecharts are better suited than CSP for describing the internal behavior of a process.

While Statecharts do allow describing the internal operation of concurrent processes, the level of detail is not as fine as one can achieve when using a mainstream language such as Java, C, or C++. Therefore when modeling embedded components using Statecharts, programmers typically fall back to using such mainstream languages whenever they specify detailed behavior such as the actions or conditions associated with Statecharts transitions. Rather than treating the native programming language and Statecharts as separate programming models, we propose an approach for modeling, analyzing and implementing embedded components which starts from Java and brings some of the key concepts of Statecharts as first-class elements within this language. One of the main difficulties faced by industrial clients when using Statecharts is the out-of-synch that arises between the code generated from Statecharts and the original Statecharts model when the code is modified after it is generated. Debugging and performance tuning of embedded application is almost always done on the generated code; in this phase there is little incentive to track back the changes to the model itself in such a way that the implementation and the model remain consistent throughout their life-cycle. This becomes a problem later on when one needs to understand the functionality of the application, modify the model, or prove properties about and maintain the application. At this point, it is very difficult to automatically infer the changes in the model based on the implementation, so Statecharts are typically used exclusively as a starting point for modeling rather than a live application specification. In this context, we believe that FUSE offers the following advantages:

- It reduces out-of-synch issues since the Statechart specification is part of the code.
- It helps programmers with the acceptance of modeling by replacing the burden of documenting and maintaining the model with a unified programming framework.
- The user does not need to master two different programming languages.
- It can provide better support for debugging and software maintenance. In other approaches, when generating Object-Oriented code from Statecharts, states are transformed into objects; this makes it difficult to later correlate these objects back to states in Statecharts.
- It supports features of object-orientation, concurrency and strong typing, and incorporates advanced programming language concepts inspired from predicate dispatch and multiple and dynamic classification [Chambers 1993; 1997; Ernst et al. 1998; Millstein 2004; Sreedhar and Marinescu 2005]. Within the same unified programming language we also preserve some of the the advantages of Statecharts of having a formal basis. This feature complements the strengths of Java to allow building complex, concurrent applications.

The rest of the paper is organized as follows. Section 2 discusses the related work. Section 3 introduces the running example that we use to present our language. Section 4 introduces FUSE's main language features and describes a simplified syntax for FUSE. Section 5 presents its semantics. Section 6 summarizes the paper with a few directions for ongoing and future work.

2. BACKGROUND

Statecharts and State Diagrams in OO Languages. Harel [Harel 1987] introduced Statecharts to overcome the limitations of conventional finite state machines. Due to their popularity, Statecharts in many semantic variations [von der Beeck 1994; Björklund et al. 2001; Gamma et al. 1995] are part of many modeling tools. The Object Management Group (OMG) has standardized state diagrams as part of the Unified Modeling Language [UML 2011]. To implement Statecharts in OO languages Ran [Ran

1994] represents the concept of state as OO classes. Sane and Campbell [Sane and Campbell 1995] adopt the same mapping while representing transitions via operations and embedded state via a table for the superstate. FUSE keeps the class hierarchy and the chart-states separate. This helps with reducing the number of classes and using the class hierarchy without breaking behavioral subtyping. Chow et al. [Chow 1978] represent Statecharts states as constant attributes of a class and use an additional attribute to represent the current state that the object is in. Quantum Hierarchical State Machines [Samek 2002] is an event-based framework based on the pub-sub model and implements run-to-completion (RTC) semantics for active objects (AO). AOs cannot share data. QHSM follow the UML semantics for Statecharts. In comparison, FUSE is a thin layer on top of method calls, only enriching the programming language, which in turn can be used in both event- and data-based programming styles and frameworks, like for example pub-sub. FUSE implements RTC semantics only for single events (as opposed to threads/AOs). In FUSE, data sharing is possible. FUSE adpots a variant of Pnueli/Shalev's fix-point semantics. Lastly, the main goal of FUSE is to mechanically generate Java code with good traceability properties.

To translate Statecharts into an OO language Ali and Tanaka [Ali and Tanaka 1999] introduce a helper object to encapsulate all the state-specific behaviour of the object representing a Statecharts. Transitions involve helper object creation and destruction. Niaz and Tanaka [Niaz and Tanaka 2004] use design patterns to generate Java code from UML state diagrams, representing states as objects and hierarchical states using object composition and delegation. Rhapsody [Gery et al. 2002] allows creating UML models and generates C, C++, or Java from them based on the Open XML Framework (OXF)—an active object-based framework. The dynamics are defined within the OXF framework. States are represented as data values, transitions as variable assignments, and events as classes. A number of other tools support code generation from Statecharts. iState [Sekerinski and Zurob 2001] implements a variant of UML Statecharts, is event-centric and provides a construct for parallel composition as a means to translate concurrent states. The code generator implements RTC for transitions. Work by Mikk et al. [Mikk et al. 1998], based on the STATEMATE semantics, accommodates compound events without parameters, and compiles Statecharts into extended state machines. In our experience with FUSE, the absence of event parameters simplify the semantics but also limits the expressive power.

[Köhler et al. 2000] proposes to generate code from Statecharts in which every state is a subclass of a generic *FReactive* class, which provides a pointer to the current state and handles events via one of its methods. This approach follows the UML semantics. Events are targeted to specific receivers, simulating broadcast with a pub-sub mechanism. Tomura et al. [Tomura et al. 2001] present the Statecharts design pattern for modeling the dynamic behaviour of components of an open distributed control system. In Hugo [Knapp and Merz 2002] a state is an object with methods for activation/deactivation, initialization, and event handling.

FUSE supports the key features of Statecharts but is more expressive; it is also unique, to the extent of our knowledge— in exploiting multiple and dynamic classification mechanisms and predicate methods to model Statecharts.

Embedded Languages. nesC is a programming language targeted to networked embedded systems [Gay et al. 2003] that supports asynchronous event-driven execution and a flexible concurrency model. The model of computation in languages such as Esterel [Berry 2000] and Signal [Beneviste et al. 1999] is synchronous. The concurrency can be compiled away, and the system behaves like a state machine. Esterel handles reactions to absence of events. Following the tradition of fix-point semantics in Statecharts, in FUSE the absence of an event is only evaluated at the time of reac-

tion, which only happens when another event triggers a reaction (see Section 4). Also, *logical causality* checks needed in synchronous languages are not necessary in FUSE as all programs have a unique semantics. Other approaches exist to extend traditional programming languages by synchronous constructs. Reactive C [Boussinot 1991] is inspired by Esterel but adopts explicit execution instead of Esterel-style FSM execution. FairThreads [Boussinot 2006] and SynchCharts in C [von Hanxleden 2009] follow reactive-style semantics and provide true, and deterministic, concurrency within C programs. In ECL [Lavagno and Sentovich 1999] C programs are annotated with Esterel-like constructs. The Esterel-style ECL modules are compiled into EFSMs. PRET-C [Roop et al. 2009] is another synchronous extension of C which focuses on temporal predictability. [Edwards and Zeng 2007] propose to dynamically generate C code that runs threads concurrently, each of which execute short groups of instructions without a context switch. Inspired by synchronous languages, SHIM [Tardieu and Edwards 2006] uses both rendezvous-style communication as well as thread communication at synchronization points. Concurrency is explicitly specified.

Pioneered by Ptolemy [Buck et al. 1994] several approaches advocate the heterogeneous combination of semantics to design embedded systems. Metropolis [Gößler and Sangiovanni-Vincentelli 2002] introduces communication refinement as a mechanism for specializing general models of computation for specific domains. Lee and Zheng [Lee and Zheng 2007] leverages principles of synchronous languages as a coordination language rather than a programming language. The POLIS [Balarin et al. 1999] co-design approach uses an event-driven model for both the hardware and the software components. galsC is a globally asynchronous and locally synchronous (GALS) model for programming event-driven embedded systems [Cheong et al. 2003; Cheong and Liu 2005]. Unlike galsC, FUSE supports both event-based and state-based programming. SystemJ [Gruian et al. 2006] is a system-level language which extends Java with synchronous Esterel-like features and asynchronous CSP-like constructs for modeling GALS systems. A reaction consists of the synchronous composition of a set of threads, that communicate and synchronize via signals. Output signals must be resolved in each tick.

BIP [Basu et al. 2006] is a framework for modeling heterogeneous real-time components which is more general than FUSE. BIP allows both synchronous and asynchronous composition of components. Our approach follows more closely the Statechart design methodology. While FUSE does not allow to delay the processing of events for a later time, the coarse-grained control-flow is separated from the rest of the application. The execution depends on the occurrence of events, the state of the charts and the values of the program data.

3. A SIMPLE STATECHARTS EXAMPLE

This section presents the requirements and the Statecharts specification for a simple vending machine which can deliver several types of coffee-based beverages (CVM). CVM is expected to perform some of the following types of operations:

- Accept coins for money, making sure that all coins are either nickels, dimes, or quarters. Return invalid coins.
- The customer may change her mind after inserting some coins and request a refund: no beverage is dispensed and the money is returned.
- Accept drink selection. CVM has three buttons for selecting cafe, decafe, or choc. Mixed drinks may be allowed; for instance, cafe and choc results in a mocha—a $50/50$ mix of coffee and chocolate with a price of 85 cents. If the customer presses more than one button and the combination is not allowed, no drink is dispensed.
- If the customer has not deposited enough money CVM does not dispense the drink until enough money is available.

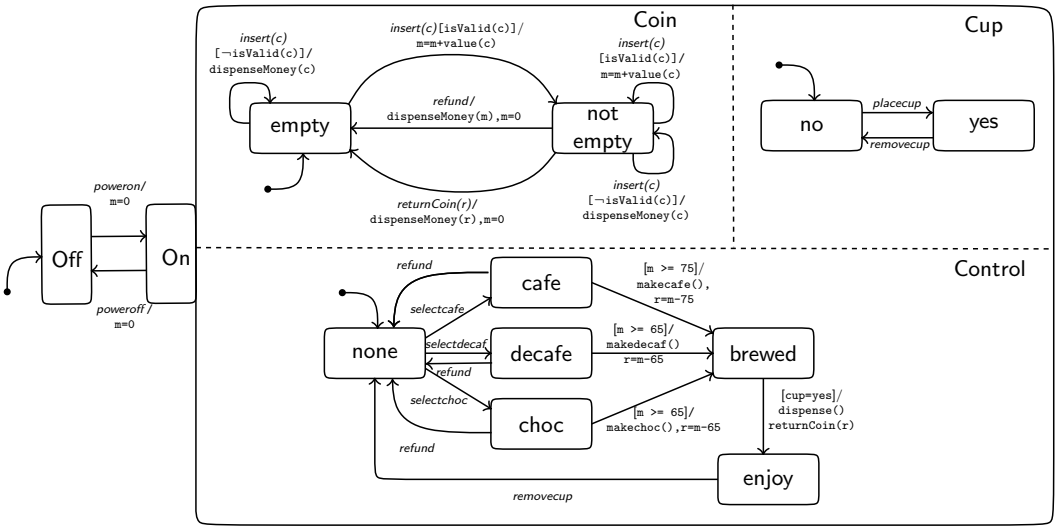


Fig. 1: Statechart for a simple coffee vending machine

- Change is returned after the beverage is dispensed.
- The customer is expected to place a cup which will collect the beverage. CVM does not dispense the drink if a cup is not in place.
- There exists a light indicator which turns itself on when the CVM is turned on and stays lit until the CVM is turned off.
- The CVM starts in the state Off. When being switched on it goes into a configuration where no money is deposited, no cup is placed, and no beverage choice is made.

Fig. 1 illustrates a hierarchical Statechart model for a subset of the requirements above. The events are specified in *italics*; the conditions are enclosed in [square brackets]. This hierarchical statechart has two states: Off for power off and On for power on. The designated Start state for CVM is Off. On consists of three and-charts: Coin, Cup, and Control with Start states empty, no, and none. The Coin chart handles state changes as result of a person inserting a coin, asking for a money refund, and for money being returned as change or refund. CVM only accepts valid coins and updates the total amount of money based on each coin inserted. The transition condition `[isValid(c)]` tests that the coins are either nickels, dimes, or quarters, whenever an *insert(c)* event occurs. Note that events can have parameters. The Cup chart keeps track of whether there is a cup in place for dispensing the drink or not. The Control chart allows the user to select a drink type. When there is enough money deposited, the CVM brews the requested drink. Note that the statechart does not order the operations of inserting money and ordering a drink. If the cup is in place then CVM dispenses the drink into the cup and returns the change. As part of a transition execution new events—internal or external—may be generated. An event is *external* if it is a primitive event emitted to the charts environment by the user or FUSE code. If the event is consumed within the current execution step without it being observable by the environment it is called *internal*. In our example, as result of transitioning from brewed to enjoy in Control, the CVM generates the internal *returnCoin(r)* event which is consumed by the chart Coin.

The statechart in Fig. 1 does not allow to request mixed beverages such as a mocha. Fig. 2 shows alternative specifications which support this functionality; for reasons of space we only show the parts of the specifications which bear the changes. A new state mocha is introduced, as well as transitions to and from this state—specifically from

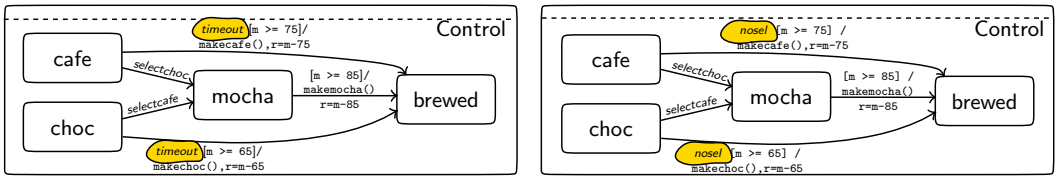


Fig. 2: Statechart for a CVM dispensing mocha with **(a) timeout (b) user input**.

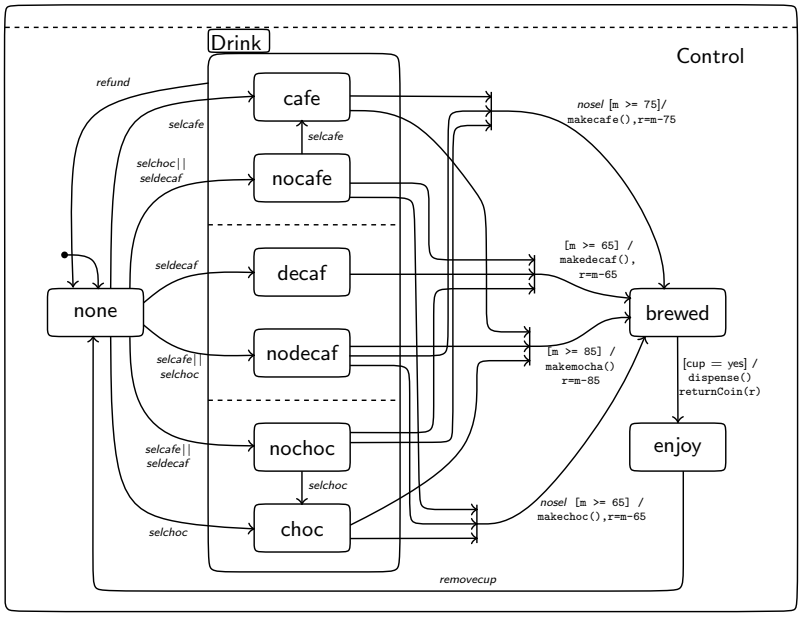


Fig. 3: Statechart for a CVM without state explosion.

cafe and choc, and into brewed. Neglect for the moment the newly introduced *timeout()* and *nosel()* events. Each of the states cafe and choc now has two outgoing transitions: one taken on the occurence of either *selectchoc()* or *selectcafe()*, and another one on the occurence of any event given that the associated condition holds. If a transition does not have a triggering event it may be triggered by any external event and it is called a *completion transition*. Let's take the case of state cafe in Fig. 2. If *selectchoc()* does not occur in state cafe and `m >= 75` then when any other event occurs the CVM would move into brewed and prepare a coffee drink. What this means is that whatever external event happens before the user has a chance to generate *selectchoc()* will disable the choice of ordering a mocha drink. On the other hand, if the user generates a *selectchoc()* event in state cafe and `m >= 75` then both transitions are valid. The priority rules for FUSE give precedence to the external event *selectchoc()* and execute the transition to mocha instead of the completion transition to brewed.

Fig. 2 shows two alternatives for eliminating this behavior of unwillingly triggering a completion transition. **(a)** employs a timeout strategy; the user is given a time to press the coffee button in state choc or the chocolate button in state cafe. If this time lapses without her making a choice, the CVM will dispense the beverage which corresponds to the current chart-state. If the user chooses a mixed drink which is not

allowed— such as coffee and decaf—the CVM remains in the current state. FUSE does not allow the processing of multiple events in the same execution step, so if two external events *timeout()* and *selectcafe()* happen "at the same time", their occurrence is serialized by the runtime-system. The statechart in **(b)** describes an alternative design; when entering either of the states cafe or choc the CVM prompts the user whether he wants to make a further selection for a mixed drink. If the answer is no the CVM dispenses the coffee or the chocolate drink; otherwise it waits for a further selection. The explicit user choice eliminates the undesirable behavior present in **(a)** in which the choice of the event to process may be made based on a policy outside the control of FUSE. To limit the state explosion in the case of allowing an increasingly large number of drink mixes we may instead create a set of and-charts, one for each mix component—as shown by Drink in Fig. 3. Each such and-chart can be in the state yes—i.e. the component is part of the mix—or no—otherwise. Note that if the user chooses a mixed drink which is not allowed the CVM goes into a state which may only be left by requesting a refund. Other solutions are possible at the expense of introducing extra transitions. We will see that adding behavior such as ordering a mocha in FUSE requires only small changes and no additional chart-states. It is possible to avoid incompatible drink choices by introducing a transition condition which disallows moving into cafe or choc if the current state is decafe. This makes FUSE specifications more concise than Statecharts, and modifications more independent.

4. FUSE ELEMENTS BY EXAMPLE

A FUSE program consists of classes, predicate methods, charts, and chart-states. Classes describe the data that objects will manipulate at runtime. FUSE classes are essentially OO classes that *may* have charts associated with them. In Fig. 4, in addition to all definitions allowed by Java classes, CLASS-BODY also contains transition definitions of type METH-DECL, which we describe later in this section. FUSE charts and chart-states model the concept of state in Statecharts. The predicate expression of a predicate method [Chambers 1993] models the concept of a transition condition in Statecharts; events act as triggers of predicate methods. The actions associated with transitions are modeled as predicate methods. These methods cannot explicitly access Statechart states but can generate events.

FUSE adopts a variant of the fix-point semantics of Pnueli and Shalev [Pnueli and Shalev 1991]. Under the Pnueli/Shalev semantics an execution step consists of immediately and logically instantaneously executing all of the transitions transitively triggered by the occurrence of the external event in the current state. In FUSE, internally generated events are consumed in the same reaction step; they are not queued and consequently, they are lost at the end of the step. This semantics implies that multiple transitions in concurrent and-charts could execute in parallel but at most one transition can be fired for each set of non-orthogonal charts. The main differences between the Pnueli/Shalev and FUSE fix-point semantics are that we do handle data in our model and, as we explain in Section 5, we execute a maximal compatible set of transitions per reaction. This set is an approximation of the maximal set of transitions as defined by Pnueli/Shalev. Apart from efficiency reasons when computing the set, there could be scenarios in which it isn't known upfront whether some event will be generated, due to the fact that FUSE allows modeling at a finer granularity. This means that in some cases the set of executing transitions will be a subset of the one an oracle may choose. FUSE does not include the notion of compound events; external events are serialized per object at the caller site. When processing an event *e* we assume that no other event is present. Negated events, in the sense of the absence of an event *f* triggering a transition, can be modeled as the occurrence of another event *e* taking place at the time when *f* is expected.

```
1        OBJ-DECL ::= class-name obj-name
2      CLASS-DECL ::= class class-name [has chart-name (EXT-EVENTS )* (INT-EVENTS )* ] CLASS-BODY
3     CHART-STATE ::= chart-st-name [has { chart-name(,chart-name )*}]
4      ST-CH-STATE ::= start: CHART-STATE
5      CHART-DECL ::= chart chart-name = {ST-CH-STATE (, CHART-STATE )*}
6       CH-ST-EXPR ::= [chart-st-name] (.chart-name .chart-st-name)*
7       BASIC-EXPR ::= CH-EXPR (== | !=) chart-st-name
8          CH-EXPR ::= (var-id | chart-name) .(chart-st-name.chart-name )* | chart-st-name
9       METH-DECL ::= trans method-id [triggered by event-name TRANS-MB] [when PRED-EXPR TRANS-MB]
                      [remove (CH-EXPR.chart-st-name; )+] [add (CH-EXPR.chart-st-name; )+] TRANS-MB |
                      trans method-id (entry | exit) state CH-EXPR = chart-st-name TRANS-MB
10       PRED-EXPR ::= BEXPR | PRED-EXPR && PRED-EXPR
11         TRANS-MB ::= TRANS-MB ; TRANS-MB | RESTR-MB
12         RESTR-MB ::= JAVA-METH-BODY
15       EXT-EVENTS ::= external: {ext-event-name ( (type var-id )*) (, ext-event-name ((type var-id )*) )*}
16       INT-EVENTS ::= internal: {int-event-name ( (type var-id )*) (, int-event-name ((type var-id )*) )*}
20            BEXPR ::= BEXPR&&BEXPR | BEXPR||BEXPR | BEXPR->BEXPR | notBEXPR | BASIC-EXPR
```

Fig. 4: FUSE-specific syntax.

4.1. Charts and Chart-States

A *chart* has a name and a fixed set of *chart-states*, used to specify mutable information (the state of the machine described by a Statechart). Instances of a class associated with a chart can receive events and execute reactions that manipulate this mutable information. For instance, the chart Control from the Statechart in Fig. 3 is specified as a `chart` in FUSE. The possible states of this or-chart are {none, Drink, brewed, enjoy}. Fig. 5 shows a snapshot containing parts of the FUSE code implementing the Statechart in Fig. 3. 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. Charts externalize and explicitly reify transient internal states of an object. Take the example of a person who is single and gets married. Without charts, you may specify `single` and `married` to be two distinct classes which inherit from a class `Person`. Bob, who is single, cannot easily change his class during his lifetime. If he gets married a new `Married` class needs to be instantiated and the old instance of `Single` destroyed. In FUSE, the social status can be modeled as a chart `SocialStatus` with chart-states `single` and `married`. When Bob gets married the object updates its chart state from `single` to `married` and the way inheritance is often used, which breaks the Liskov Substitution Principle [Liskov and Wing 1994] of subtyping, is avoided. This approach also reduces the number of classes. Alternatively, one could use a tagged enum type with values `Single` and `Married`. However, union types do not provide the same advantages as the FUSE approach. Using Statecharts allows the programmer to separately specify the coarse-grain control of the application and enable the use of verification tools. These tools include static calculation of event dependencies, reachability of states or synthesis of specialized event loops by calculating upfront the set of running transitions.

4.2. Triggering Events

Each `trans` construct in Fig. 5 defines a method of type `void` which corresponds to a Statechart transition. The event *e* that triggers a transition is declared with the `triggered by` construct; if this construct is missing then the method models a completion transition. A METH-DECL in Fig. 4 defines a FUSE transition. A method modeling a Statecharts transition is `triggered by` an event and may be predicated. The body of the actions associated with Statecharts transitions cannot refer to Statechart states. Instead, the `remove` and `add` constructs explicitely specify the effect of the transition in terms of the old states which are left and the new states which are entered.

```
chart Coin = {start: empty, notempty}   chart Cup   = {start: no, yes}
chart cafe = {start: no, yes}          chart decaf = {start: no, yes}
chart choc = {start: no, yes}
chart Control = {start: none, drink has {cafe, decaf, choc}, brewed, enjoy}
chart PowerSwitch = { start: off, on has {Coin, Cup, Control}}

class CVM has PowerSwitch {
   int money =0, rem =0;

   use ctrl as PowerSwitch.on.Control; // ctrl is an alias for PowerSwitch.on.Control
   use cup as PowerSwitch.on.Cup;     use coin as PowerSwitch.on.Coin;
   use drink as ctrl.drink;           use ischoc as drink.choc;
   use iscafe as drink.cafe;          use isdecafe as drink.decafe;

   external: powerOn(), powerOff(), selectcafe(), selectchoc(), selectdecafe(), nosel();
   internal: returnCoin(int rem);

   CVM() {} // constructor

   trans off2on triggered by powerOn() when PowerSwitch == off
     remove PowerSwitch.off; add PowerSwitch.on; {}
   ...
   trans none2cafe triggered by selectcafe() when ctrl == none
     remove ctrl.none; add ctrl.drink; iscafe.yes; ischoc.no; isdecafe.no; {}
   trans none2cafe triggered by selectcafe() when iscafe == no && isdecafe == no
     remove iscafe.no; add iscafe.yes; {}
   ...
   trans cafe2brewed triggered by nosel() when iscafe == yes && money >= 75
     remove iscafe.yes; add ctrl.brewed; {makecafe(); rem = money - 75;}
   ...
   trans cafe_choc2brewed
     when iscafe == yes && ischoc == yes && money >= 85
     remove iscafe.yes; ischoc.yes; add ctrl.brewed; {makemocha(); rem = money - 85;}
   trans brewed2enjoy() when ctrl == brewed && cup == yes
     remove ctrl.brewed add ctrl.enjoy
     { dispenseBeverage(); internal_event_process(new returnCoin(rem));}
   ...
};
```

Fig. 5: Partial FUSE specification of `CVM` without state explosion

4.3. Enabling Conditions

FUSE borrows the concept of predicate methods from the predicate dispatch paradigm [Chambers 1993] to express conditional execution in Statecharts. A predicate method is a method which can be invoked only when its predicate expression—also called the `when`-expression—evaluates to `true`. Specifically, FUSE models the transition condition `c` as part of the `when`-expression. Additionally, the `when`-expression must test that the transition occurs only in the designated chart-state. For instance the enabling condition of method `cafe2brewed` establishes that the current chart-state is `PowerSwitch.on.Control.drink.cafe == yes` and `money >= 75`. It is possible that the transition condition contains boolean tests involving the parameters of the triggering event. Currently, FUSE enforces that no primitive boolean test mixes data tests, event tests and chart-state tests. This restriction allows checking the data condition and the chart-state condition at the beginning of a reaction when no transition has

started running and no internal event is present yet. Consider the following piece of code for the chart and class definitions in Fig. 5:

```
CoffeeMachine cm = new CVM() ;
cm.external_event_process(new powerOn());
cm.updateMoney(75) ;
cm.external_event_process(new selectcafe()) ;
cm.external_event_process(new nosel());
```

We use the `external_event_process()` method to trigger a reaction from the environment, passing an external event as argument. After processing *selectcafe()* `CVM` is in chart-state cafe and the field variable `money` has value 75. In this state `CVM` receives the external event *nosel()*. There is only one method triggered by this event and whose predicate expression evaluates to `true` in the state cafe, namely `cm.cafe2brewed()`. The invocation dynamically changes the state of chart `Control` to brewed. If the call to `updateMoney()` had as argument the value 30 instead of 75, the `when`-expressions of all transitions triggered by *nosel()* with source state cafe would evaluate to `false` and the event would be dropped.

In the simplified syntax in Fig. 4, predicate expressions include PRED-EXPR, which are Boolean expressions over the set of chart states that ensure that transitions occur only in the designated chart states. In this version of FUSE, the part of the predicate expression that tests the object state is provided by the user as Java code—via the method body following PRED-EXPR; the event parameters are tested in the body of the `triggered by` construct.

4.4. Entry and Exit Methods

One of the application requirements is that the `CVM` has a light indicator which will turn itself on and off to reflect the state the coffee machine is in at any time. This behavior can be coded in methods as part of taking a transition. If entering or leaving a chart-state always requires this behavior, the user must replicate it in each method that enters or 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. For instance, the class `CVM` may define the entry method below when turning off:

```
trans in_off entry state PowerSwitch == off {system.lightOff();}
```

Whenever PowerSwitch enters the chart-state off the `entry` method is also invoked, which in turn invokes `system.lightOff()`. A METH-DECL in Fig. 4 may also model `entry` or `exit` methods associated with entering or exiting the specified chart-state. The `entry`/ `exit` methods do not have either a trigger nor a predicate expression; their bodies cannot refer to the Statechart states but can generate events.

4.5. Hierarchical Charts

Consider again Fig. 5. A chart consists of a set of chart-states which can be hierarchically defined. At the top level, the Statechart hast two *or*-states: On and Off. An or-chart simply enumerates its chart-states. On consists of three *and*-states: Coin, Cup, and Control. Each of the three concurrent states are made from a set of or-states. We can model this hierarchical Statechart in FUSE as hierarchical chart-states using the keyword `has`. The algorithm that controls the reaction to an external event guarantees that if the chart is in a nested chart-state then the higher containing states are also active.

4.6. Event Generation

External events are created and thrown by invoking the `external_event_process()` method. Internal events created within the class are processed using an invocation

to the method `internal_event_process()`. Both of these methods are synthesized by FUSE (see Section 5). The body of `brewed2enjoy` illustrates how internal events are generated. Note that the event *returnCoin(rem)* has an integer parameter. Although the syntax in Fig. 4 does not allow using explicit event parameters when triggering a transition, they must be of the correct type given the specified event signature, which is imposed by the strong typing discipline of the programming language. To ensure that the sets of internal and external events are disjoint we explicitly declare these as part of the class definition and raise an error if there exists an event with the same signature which is declared to be both internal and external. The body of a method modeling a Statecharts transition is a Java method which may additionally generate internal and external events via the `internal_event_process()` and `external_event_process()` Java calls, where the events must comply with the event signatures as specified by INT-EVENTS and EXT-EVENTS.

4.7. Start States

In order to preserve a consistent chart-state, when entering a hierarchical state it is necessary to mark some sub-states as well. The default option is described by Start states that are specified via the construct `start:` of FUSE. An or-chart must designate a single sub-state as the Start state. A hierarchical and-state must designate a Start state for every orthogonal and-chart that it contains. Syntactically, the first state in the definition of an or-chart is the start state of the or-chart. The object constructor initializes by default its chart to its designated Start states recursively. If this is an and-chart then it will enter all start states of the composing and-charts. When a transition into a hierarchical state runs, it implicitly initializes all of its immediately hierarchical charts to the designated Start states.

5. FUSE SEMANTICS

The semantics of FUSE are based on the fix-point semantics for Statecharts. Under this semantics, the execution of a chart consists of a continuous sequence of reactions, each of which occurs in response to an external event. Each reaction transforms the internal state of the chart, and possibly emits external events to the chart's environment. Additionally, internal events can be generated during the execution of a reaction, but these internal events are not visible by the environment of the chart. At the end of the reaction all internal events generated are consumed and destroyed.

Each reaction in FUSE is composed from the effects of the execution of zero or more transitions. According to the fix-point semantics, a maximal set of transitions is run in each reaction. Some transitions are triggered directly by the external event that sparks the reaction. Some others are triggered by internal events generated by other transitions that are fired in the same reaction. Conceptually, all transitions observe – as their initial state – the state of the chart at the beginning of the reaction. The only chain of information to decide which transitions are part of a reaction happens through internal events.

We describe in this section the semantics of FUSE by presenting a translation of each of the language constructs into Java (the *target language*). Every class associated with a chart is translated into a class in the target language that contains some extra nested classes and methods.

5.1. Events, Event Handlers and Transitions

We show now how the description of events, event handlers, enabling conditions and bodies of transitions get translated to the target language.

Events. The communication and synchronization between the environment and the chart is carried out using *external* events, that may contain data. More concretely, in the target language a reaction is started by an external event being thrown to an object that is an instance of the class associated with the chart. External events are modeled as a class that inherits from an abstract event class. As in conventional OO languages shared data is also available. For example, transitions can read and modify data members of the object associated with the chart receiving the event, or access and modify global data. In FUSE, external and internal events are explicitly declared in the generated class associated with the chart. For example:

```
class A has mychart
     external: insertCoin(int val),placeCup() /* external events */
     internal: returnCoin(int val)            /* internal events */
```

declares two types of external events (`insertCoin` and `placeCup`) and one internal event (`returnCoin`). In the compiled code, the following class hierarchy, nested within `class A` is created:

```
class A { ...
   public abstract ExternalEvent extends AbstractEvent {}      /* external events */
   public InsertCoin  extends    ExternalEvent { int val; }
   public PlaceCup    extends    ExternalEvent {}
   private abstract InternalEvent extends AbstractEvent {}     /* internal events */

   private ReturnCoin extends    InternalEvent {
     int val;
     void generate(void) { ... }
   }
}
```

Note that external events are public, so the environment can generate and throw external events. Internal events are private. The separation between external and internal events is a design principle of FUSE. Internal events are a tool for developers to decompose the behavior of the chart into simpler activities that communicate. External events are part of the external interface of the chart. In FUSE we use the encapsulation provided by the visibility rules (private and public qualifiers) of Java methods to enforce this separation. Non-abstract internal events contain the method `generate`, which is synthesized by the FUSE compiler, that calculates which transitions become enabled by the internal event:

```
void generate(void) {
  /* for all transitions t1, t2, ... that may be triggered by this event */
  check_and_activate(t1,this);
  check_and_activate(t2,this);
  ...}
```

The auxiliary method `check_and_activate` first makes sure that the transition has not been already considered candidate for running in the same reaction. Otherwise, it uses the method `is_triggered_by` () to check whether the event is actually triggering, in order to declare the transition a candidate to run.

Event Handlers. The reaction to an external event is handled by a method called `external_event_process()`. This method implements the *reaction loop*, which incrementally computes the maximal compatible set of transitions, and which is synthesized in the resulting class. Some transitions generate internal events during their execution, invoking method `internal_event_process()`. In turn, this method simply invokes `generate` as described above.

```
/* the reaction loop */
public void external_event_process(ExternalEvent ev) { ... }
```

```
/* internal event handler */
private void internal_event_process(InternalEvent ev) {
  ev.generate();  }
```

The current version of FUSE is sequential as it uses a single thread of execution for the whole reaction loop. As a result, `internal_event_process()` runs in the same thread of control as `external_event_process()`, and no synchronization with the running set of transitions is needed.

Transitions. The declaration of transitions in FUSE from line `9` in Fig. 4 contains enough information to generate the enabling condition and the action for every transition. The FUSE compiler first checks that the `when` clause, and the `add` and `remove` sets define an admissible transition: for each consistent chart state for which the `when` clause holds, if one applies the `add` and `remove` sets then the chart state obtained is also consistent. Moreover, every transition must also be well-defined, in the sense that the only two kind of states that can be hierarchically related are two `add` states. The reason to allow adding sub-states is to activate a non-default state. A transition that is either not admissible or not well-defined causes a compilation error. FUSE then generates, for each admissible transition `t` the following:

```
Transition t = new Transition () {
   boolean is_triggered_by(AbstractEvent ev) { ... }
   boolean is_enabled()              { ... }
   void    body()                    { ... }
}
```

Each transition extends the abstract class `Transition`:

```
abstract class Transition {
   public boolean discarded = false;
   public boolean ready     = false;
   public abstract boolean is_triggered_by(Event ev);
   public abstract boolean is_enabled();
   public abstract void    body();
...}
```

The method `void body()` performs the computation of the action of the transition, including effects like updating variables, etc. The enabling condition of a given transition `t` is split into three different activities. This separation is crucial for the correct incremental computation of the maximal set of transitions:

(1) `bool is_triggered_by(Event ev)`: returns whether a given event is triggering for the transition. In other words, this method corresponds to the part of the enabling condition related to events. For transitions triggered by internal events, this method is invoked within the `generate` method of the event. For transitions triggered by external events, this method is invoked directly by the reaction loop.
(2) `bool is_enabled()`: this method checks whether the values of the data members handled by the object enable the execution of the transition. This feature is inspired by predicate methods in the predicate dispatching paradigm, except that a transition is only triggered as a part of a reaction.
(3) the enabling condition of the transition also depends on the current active states of the chart, as described by the `when` clause. This clause is evaluated by the reaction loop, so no explicit method is generated for it.

5.2. Configuration, Scope, Admissibility and Compatibility

We introduce now the mathematical definitions to design and reason about the reaction loop algorithm, which provides the operational semantics of FUSE.

Configurations. A *chart vocabulary* is a labelled tree where the root is labelled with the id of the chart. A non-hierarchical chart has, as its only children, one node for each state. A hierarchical chart has one child per sub-chart. We represent these labelled trees as prefix closed sets of strings over the alphabet of chart names. This way, for every labelled tree t, $\epsilon \in t$ and if $s.a \in t$ then $s \in t$. For conciseness, we represent these prefix closed sets by the set of their leaves. For example, the chart `PowS` (power switch):

```
chart Coin = {start: empty, notempty}
chart Cup  = {start: no, yes}
chart PowS = { start: off, on has {Coin, Cup}}
```

has the following chart vocabulary:

{ `PowS.off`, `PowS.on.Coin.empty`, `PowS.on.Coin.nonempty`, `PowS.on.Cup.no`, `PowS.on.Cup.yes` }

We use *configuration* to refer to a given set of active chart states.

Definition 5.1 (*configuration*). A configuration σ of a chart vocabulary c is a subtree (prefix closed set of strings) of c that satisfies:
— if an *and* node is in σ then all its children are in σ;
— if an *or* node is in σ, exactly one of its children is in σ.

In the `PowS` example above, the following are two of the possible configurations: {`PowS.off`} and {`PowS.on.Cup.no`, `PowS.on.Coin.nonempty`}. A chart *state* is a path in the chart from the root, in other words, an element of the chart vocabulary c. A non-hierarchical state is a path leading to a leaf node, while a hierarchical state is a path leading to a non-leaf node. A *state* s is present in a configuration σ whenever $s \in \sigma$. Consider now the BEXPR expressions used in FUSE to define the chart-state enabling condition for a transition (lines 7 and 20 in Fig. 4). The semantics of BEXPR define whether a predicate expression is satisfied in a given chart configuration σ. For basic expressions BASIC-EXPR:

– $\sigma \models a$ `==` b whenever $a.b \in \sigma$, and $\sigma \models a$ `!=` b whenever $a.b \notin \sigma$.

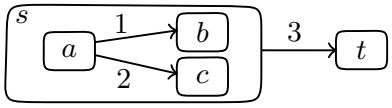
For Boolean combinations BEXPR we use the standard semantics:

– $\sigma \models x$ `&&` y whenever $\sigma \models x$ and $\sigma \models y$,
– $\sigma \models x||y$ whenever $\sigma \models x$ or $\sigma \models y$,
– $\sigma \models x$ `->` y whenever either not $\sigma \models x$ or $\sigma \models y$,
– $\sigma \models$ `not` x whenever $\sigma \not\models x$.

Finally, the enabling condition for a transition `t` with a `when` clause w is $\sigma \models w$.

Effect and Scope. Effects describe changes in the chart state. Scopes capture whether a set of transitions is compatible and therefore can run simultaneously. The effect of a transition is characterized by a pair (A, R) where A is the add set and R is the remove set. A well-defined effect is such that the only elements of $A \cup R$ that can be hierarchically related are A states. The composed effect of two effects (A_1, R_1) and (A_2, R_2) is the pair $(A', R') = (A_1, R_1) \sqcup (A_2, R_2)$ where R' contains all elements in $R_1 \cup R_2$ that are not under elements in $A_1 \cup A_2 \cup R_1 \cup R_2$. Similarly, A' contains all elements from $A_1 \cup A_2$ that are not under elements in $R_1 \cup R_2$. This produces a well-defined effect.

Example 5.2. Consider for example the following chart (left), and the effects (A_1, R_1), (A_2, R_2) and (A_3, R_3) of transitions 1, 2 and 3 (right):



$$R_1 = s.a \quad R_2 = s.a \quad R_3 = s$$
$$A_1 = s.b \quad A_2 = s.c \quad A_3 = t$$

The composed effect $(A_1, R_1) \sqcup (A_2, R_2)$ is $(\{s.b, s.c\}, s.a)$. The composition of the three effects is (t, s).

Definition 5.3 (*Cover*). A cover c of a well-defined effect (A, R) is a set of states such that:

— no two states s_1 and s_2 in c are a prefix of each other.
— every state s in c is the prefix of at least one state in A and one state in R.
— every state a in $R \cup A$ has a prefix in c.

An order $\sqsubseteq$ can be defined between covers as follows: $c \sqsubseteq d$ whenever for every $s \in c$ there is an $s' \in d$ with s' a prefix of s. A *scope* is a minimal cover. For example, in Ex. 5.2, the scope of transition 1 is $\{s\}$, the scope of 2 is also $\{s\}$, and the scope of 3 is the root state $\{\epsilon\}$.

LEMMA 5.4 (SCOPE). *The scope of a transition is uniquely defined.*

PROOF. The existence of minimal covers follows directly from the finiteness of the set of covers. Assume there are two different minimal covers c and d. It is easy to show that the union of c and d, removing states that are strictly under other states, is also a cover that is under both c and d, which either contradicts that c and d are minimal or that c and d are different. □

Transition Effects. Given a transition τ and a state $s \in A_\tau \cup R_\tau$ we use $scope(\tau, s)$ for the (unique) state in $scope(\tau)$ that is a prefix of s. The chart-effect of an effect (R, A) from configuration σ is defined as

$$Act_{(A,R)}(\sigma, \sigma') \stackrel{\text{def}}{=} \sigma' = (\sigma \ominus R) \oplus A$$

The set operations $\ominus$ and $\oplus$ are the set difference and union operators, enriched to
— remove all children of a removed state r (for $\ominus$), and all ancestors up to $scope(\tau, r)$
— add all necessary ancestors of an added state a, upto $scope(\tau, a)$ to preserve prefix closed sets, and add all default sub-states of a that are not above a state in A_τ.

For a single transition τ, Act_τ is defined as $Act_{(R_\tau, A_\tau)}$. Similarly, the effect of simultaneously taking a set of transitions $I = \{\tau_1, \ldots, \tau_n\}$ is $Act_I = Act_{(A,R)}$ where $(A, R) = \big((A_1, R_1) \sqcup (A_2, R_2)\big) \sqcup (A_3, R_3) \ldots$ is the composed effect.

Admissibility. A transition (or set of transitions) is admissible if when running from a legal configuration in which the transition is enabled, the resulting subtree is a legal configuration.

Definition 5.5 (*admissibility*). Given a configuration σ, a transition $\mathtt{t}$ is admissible whenever, if $En_\mathtt{t}(\sigma)$ then (1) $R \subseteq \sigma$ and (2) every σ', with $Act_\mathtt{t}(\sigma, \sigma')$, is a configuration.

Admissibility is checked for each individual transition at compile time. The semantics of FUSE dictate that maximal admissible sets of transitions must run, but unfortunately admissibility is not a monotone (incremental) property. Consider Ex. 5.2 above. Every transition is individually admissible. The set $\{1, 2\}$ is not admissible because it would leave the chart in an illegal configuration with both $s.b$ and $s.c$ marked in or-chart s. Finally, the set $\{1, 2, 3\}$ is an admissible set.

In principle, a maximal admissible set can only be computed by brute force search in the space of sets of enabled transitions. Apart from inefficient, a brute force search for admissible sets suffers from another problem. It is not known upfront which internal events will be generated and which plausible transitions will become enabled by these internal events, without actually executing the transitions that may generate the events. For this reason we introduce compatibility as an approximation of admissibility.

Compatibility. Compatibility allows the incremental computation of sets of transitions that are guaranteed to be admissible. Compatibility is a monotone property.

Definition 5.6. We say that two scopes are independent if no state in any of them is a prefix of a state in the other. Two transitions are *compatible* if their scopes are independent. A set of transitions is compatible if all its transitions are pair-wise compatible. A transition is compatible with a set of transitions if it is compatible with all transitions in the set. We use $s_1 \# s_2$ to represent that scopes s_1 and s_2 are independent.

In Ex. 5.2 the only compatible sets are the singletons. One implication of compatibility is that the union of effects $(A, R) \sqcup (A', R')$ is equal to $(A \cup A', R \cup R')$, because the sets are not hierarchically related. Moreover, $\sqcup$ becomes associative for compatible sets.

LEMMA 5.7. *Let τ_1 and τ_2 be two compatible transitions, with (A_1, R_1) and (A_2, R_2) their add and remove sets. Then $scope(\tau_1) \cup scope(\tau_2) = scope(A_1 \cup A_2, R_1 \cup R_2)$.*

PROOF. First, note that both $scope(\tau_1) \cup scope(\tau_2)$ and $scope(A_1 \cup A_2, R_1 \cup R_2)$ are covers of $(A_1 \cup A_2, R_1 \cup R_2)$. We only need to show that every s in $scope(\tau_1) \cup scope(\tau_2)$ is also in $scope(A_1 \cup A_2, R_1 \cup R_2)$. Consider an arbitrary $a \in A_1$ state, and let s be the state in $scope(\tau_1)$ covering A_1, and let $r \in R_1$ an arbitrary state under s. The existence of s and r are guaranteed by the definition of scope. Let e_2 be an arbitrary state of A_2 or R_2. This element e_2 cannot be under s, because then (by s_2 covering e_2 in $scope(\tau_2)$), e_2 would be hierarchically related with s. Hence, there can be no s' in $scope(A_1 \cup A_2, R_1 \cup R_2)$ under s, because it must be covering some a from $A_1 \cup A_2$ and r from $R_1 \cup R_2$, but no such element from A_2 and R_2 exists and then s' would be a better choice than s for $scope(\tau)$. □

COROLLARY 5.8. *Let I be a compatible set of transitions and let τ be a transition compatible with I. Then, $scope(\tau) \cup scope(I) = scope(A_\tau \cup \bigcup_{i \in I} A_i, R_\tau \cup \bigcup_{i \in I} R_i)$.*

LEMMA 5.9. *Let I be a compatible set and τ be a transition. Then τ is incompatible with I if and only if there is a τ' in I such that $scope(\tau) \# scope(\tau')$ does not hold.*

PROOF. Let I be incompatible with τ. Then, it is not the case that $scope(\tau) \# scope(I)$, so there must exist $s \in scope(\tau)$ and $s' \in scope(I)$ that are hierarchically related. Since $scope(I)$ is $\cup_{\tau' \in I} scope(\tau')$ there is a $\tau' \in I$ with $s' \in scope(\tau')$. It follows that $scope(\tau) \# scope(\tau')$ does not hold, because these scopes contain hierarchically related states. For the other direction, assume there is a $\tau' \in I$ violating $scope(\tau) \# scope(\tau')$, so there are s in $scope(\tau)$ and s' in $scope(\tau')$ that are hierarchically related. Since I is a compatible set, $s' \in I$ and hence τ is incompatible with I. □

Finally, the following theorem relates compatibility and admissibility.

THEOREM 5.10. *If a set I is a compatible set then I is admissible.*

PROOF. Let σ be a configuration, and I a compatible set whose transitions are all enabled in σ. Let σ' be such that $Act_I(\sigma, \sigma')$. By contradiction, let σ' be an inconsistent state (not a configuration). Then, there must be an or-state that has either none or more than one child, or an and-state that does not have all children. First, assume that

there is an or-state s with no child in σ. This implies that R_I contains some descendant r of s and hence there is a transition τ with $scope(\tau, r) = s$. But then, by compatibility, no other transition contains a scope that conflicts with s. It follows that all the changes under s from σ to σ' are due to R_τ and A_τ. It follows that τ is not admissible, because from σ, τ produces a configuration with s but not child of s. The other cases follow similarly. □

Theorem 5.10 justifies the use of compatibility as an incremental approximation of admissibility. Lemma 5.9 implies that compatibility is monotone (subsets of compatible sets are compatible sets). This lemma also allows the offline calculation of incompatible pairs of transitions. This calculation enables the use of bit vectors to encode the set of transitions that are compatible with a given set R.

5.3. The Reaction Loop

The reaction loop computes a maximal set of transitions that run in reaction to an external event. This computation is performed incrementally because some transitions are triggered by internal events generated by other transitions. This algorithm is synthesized as the body of the method `external_event_process()`. The reaction loop assumes that there is a given total order on transitions such that there exists always a most prioritary transition to choose from a given set of enabled transitions. We leave the construction of this total order outside the scope of this paper; this involves priority conventions such as (1) manually declared priority; (2) transitions which modify chart states that are higher in the hierarchy are more prioritary (like in STATEMATE [Eshuis 2009]); (3) external events have precedence over internal and *empty* events.

The reaction loop must guarantee that the final running set of transitions leaves the chart in a configuration. At all instants the reaction loop partitions the set of transitions into four subsets:

— R: the set of transitions that run in the reaction.
— C: the set of candidate transitions, which have met all the enabling conditions, and will run unless they conflict with some transition in R.
— D: the set of discarded transitions, that will not run.
— P: the set of plausible transitions, for which there is not enough information yet to decide whether they are candidates (C) or discarded (D).

These sets are pairwise disjoint, and $R \cup C \cup P \cup D = \mathcal{T}$. Initially, all transitions are declared plausible: $P = \mathcal{T}$ and $R = C = D = \emptyset$. These sets are not represented explicitly except for the candidate set, which is maintained in a priority queue, ordered by the order of transitions: `PriorityQueue<Transition> the_candidates;`

The algorithm proceeds in 3 phases:

Phase 1. Checking the Chart and Data Enabling Condition. This phase consists in filtering out those transitions that are not ready to run, either because of the chart state, or because of the object or global data values. The `when` clause of a transition expresses whether the transition is ready to be fired with respect to the current state of the chart. Apart from modifying the chart state, transition bodies also inspect and change internal data so either there is a mechanism to freeze the values of the data before any transition runs, or the enabling conditions for all plausible transitions must be checked before any transition starts running. We check the enabling condition for all transitions upfront. Hence, for each transition, the reaction loop checks the when clause against the current chart state and invokes `is_enabled()`. If either of these checks fail, the flag `discarded` is set to `true`. This corresponds to moving those transitions that are not ready to the discarded set. The rest of the transitions are still in P.

Phase 2. External Events. The reaction loop checks whether the received event triggers those transitions that depend on external events, by invoking `is_triggered_by` on each. Again, those transitions that return false are discarded by setting `discarded` to `false`. However, those transitions for which `is_triggered_by` returns true are inserted in the candidate queue with the following code:

```
if (check_transition_event(t1,e)) {
   t1.ready = true;
   the_candidates.add(t1);        }
```

Inserting a transition in the queue `the_candidates` corresponds to moving the transition to the C set. After phase 2, all transitions that depend on external events are either discarded or candidates.

Phase 3. Running a Maximal Set. The next step of the reaction loop is to actually execute a set of transitions. The reaction loop will incrementally compute a *maximal* compatible set of transitions (see Section 5.2 above). The idea is to repeatedly choose the maximum transition m in C according to the established total order. Transition m is run if it is chart compatible with the previously run transitions. The bit vector that encodes the compatibility of further transitions is updated by marking those transitions which are incompatible with m. As a result of running m a set of internal events may be generated which, in turn, may move some transitions from P to C. The set of transitions is maximal when C is empty, at which point, the algorithm terminates. Termination is guaranteed because each transition can be inserted in C at most once, which is enforced using the flag `ready`. A `ready` transition is not re-inserted in the priority queue that respresents C.

6. CONCLUSIONS AND FUTURE WORK

This paper has presented the FUSE language with formal semantics and a translation into Java. A prototype implementation is under development and will be released as future work. We identify several other directions for future work:

Additional Language Features. In self-loops in a chart, the introduction of a `keep` construct would make the specifications cleaner, since removing and adding the same state currently in FUSE resets substates to the default states. Moreover, sometimes there are restrictions that must apply within a statechart, such as ordering transitions which may otherwise execute concurrently. FUSE could be extended to specify *constraints*, i.e. what relationships *must* hold or are *forbidden* between chart-states. For example, in `CVM` we must enter money before we can make our drink selection:

```
constraint CoinCtrl(Coin c, Control ctrl) {not(ctrl == cafe && c == empty)}
```

Static Guarantees. Currently FUSE does not provide an explicit language to describe event parameters and constraints on events, but delegates event-based enabling conditions to the Java programmer (as methods with Boolean return type). The alternative is to introduce an *event language*. The main advantage of a language approach is that a simple, event-specific language with clean semantics would enable to determine statically which transitions will execute in a reaction loop. FUSE can be extended with one such language to allow the expression of *contracts*.

Specifically, a contract associated with a method may define the following types of information: (1) the precondition—via `triggered by` (that includes simple predicates over events) and `when`, (2) the set of all internal events that the function *must* and *may* generate—via a `generates` construct (that may include the relation between the values in the event and values in the data at the beginning of the reaction or in the triggering event), (3) the post-condition—via the `add`, `remove`, and a new `ensures` construct, (4) the set of variables `modified` by the method, and (5) the set of variables that are `accessed` by the transition. Contracts would enable the effective static checking of properties.

Compound Events and Concurrency. FUSE can be easily extended to support compound events using features of the target language Java. First, the `external_event_process()` method should be extended to receive compound events (for example event sets). The reaction loop only needs to be adapted in phase 2 to invoke several times the `is_triggered_by` function, one per event in the compound event. Alternatively `is_triggered_by` could be extended to receive event sets. To optimize the execution of the reaction loop we may allow transitions to execute concurrently. Apart from the need to synchronize the shared data, a transition at the head of the priority queue cannot simply be run unless there exists the guarantee that the currently running transitions will not generate an internal event enabling a higher priority transition. This check can be performed using the `generates` construct in the contract. At the language level, FUSE may be extended with *asynchronous methods* to model *asynchronous events* in Statecharts.

ACKNOWLEDGMENTS

This work was partially funded by the MICINN project TIN2010-16497, *Input/Output Scalable Techniques for distributed and high-performance computing environments*, by the EU project FET IST-231620 *HATS*, MICINN project TIN-2008-05624 *DOVES*, CAM project S2009TIC-1465 *PROMETIDOS*, and by the COST Action IC0901 *Rich ModelToolkit-An Infrastructure for Reliable Computer Systems.*

REFERENCES

ALI, J. AND TANAKA, J. 1999. Converting Statecharts into Java code. In *Proc. Fourth World Conf. on Integrated Design and Process Technology (IDPT'99)*.

BALARIN, F., CHIODO, M., GIUSTO, P., HSIEH, H., JURECSKA, A., LAVAGNO, L., SANGIOVANNI-VINCENTELLI, A. L., SENTOVICH, E., AND SUZUKI, K. 1999. Synthesis of software programs for embedded control applications. *IEEE Trans. on CAD of Integrated Circuits and Systems 18,* 6, 834–849.

BASU, A., BOZGA, M., AND SIFAKIS, J. 2006. Modeling heterogenous real-time components in BIP. In *4th IEEE Int'l Conf. on Soft. Eng. and Formal Methods (SEFM'06)*. IEEE, 3–12.

BENEVISTE, A., CAILLAUD, B., AND GUERNIC, P. L. 1999. From synchrony to asynchrony. In *Proc. of the 10th Int'l Conf. in Concurrency Theory (CONCUR'99)*. LNCS Series, vol. 1664. Springer, 162–177.

BERRY, G. 2000. *The Foundations of Esterel*. MIT Press.

BJÖRKLUND, D., LILIUS, J., AND PORRES, I. 2001. Towards efficient code synthesis from statecharts. In *Proc. of Practical UML-Based Rigorous Development Methods (pUML'01)*. Lecture Notes in Informatics Series, vol. 7. German Informatics Society, 29–41.

BOUSSINOT, F. 1991. Reactive C: an extension of C to program reactive systems. *Software Practice and Experience 21,* 4, 401–428.

BOUSSINOT, F. 2006. FairThreads: mixing cooperative and preemptive threads in C. *Concurrency and Computation: practice and experience 18,* 5, 445–469.

BUCK, J., HA, S., LEE, E. A., AND MESSERSCHMITT, D. 1994. Ptolemy: A framework for simulating and prototyping heterogeneous systems. *Intl. Journal in Computer Simulation 4,* 2, 155–182.

CHAMBERS, C. 1993. Predicate classes. In *Proc. of the 7th European Conf. on Object-Oriented Programming (ECOOP'03)*. LNCS Series, vol. 707. Springer, 268–296.

CHAMBERS, C. 1997. The Cecil language specification and rationale, Version 2.1. Tech. rep., CSE, University of Washington. `http://www.cs.washington.edu/research/projects/cecil/pubs/cecil-spec.html`.

CHEONG, E., LIEBMAN, J., LIU, J., AND ZHAO, F. 2003. TinyGALS: a programming model for event-driven embedded systems,. In *Proc. of the 2003 ACM Symp. on Applied Computing (SAC'03)*. ACM, 698–704.

CHEONG, E. AND LIU, J. 2005. galsC: a language for event-driven embedded systems. In *Design, Automation and Test in Europe Conf. (DATE'05)*. IEEE Computer Society, 1050–1055.

CHOW, T. 1978. Testing software design modeled by finite-state machines. In *Trans. on Soft. Eng.* IEEE, 178–187.

EDWARDS, S. A. AND ZENG, J. 2007. Code generation in the Columbia Esterel compiler. In *EURASIP Journal on Embedded Systems*. Vol. 2007.

ERNST, M. D., KAPLAN, C. S., AND CHAMBERS, C. 1998. Predicate dispatching: a unified theory of dispatch. In *Proc. of the 12th European Conf. on Object-Oriented Programming (ECOOP'98)*. LNCS Series, vol. 1445. Springer, 186–211.

ESHUIS, R. 2009. Reconciling Statechart semantics. *Sci. of Computer Programming 74,* 3, 65–99.

GAMMA, E., HELM, R., JOHNSON, R., AND VLISSIDES, J. 1995. *Design Patterns: elements of reusable object-oriented software*. Addison-Wesley.

GAY, D., LEVIS, P., VON BEHREN, J. R., WELSH, M., BREWER, E. A., AND CULLER, D. E. 2003. The nesC language: a holistic approach to networked embedded systems. In *Proc. of the ACM SIGPLAN 2003 Conf. on Programming Language Design and Implementation (PLDI'03)*. ACM, 1–11.

GERY, E., HAREL, D., AND PALACHI, E. 2002. Rhapsody: a complete life-cycle model-based development system. In *3rd Int'l Conf. on Integr. Formal Meth. (IFM'02)*. LNCS Series, vol. 2335. Springer, 1–10.

GÖSSLER, G. AND SANGIOVANNI-VINCENTELLI, A. 2002. Compositional modeling in Metropolis. In *Proc. of the 2nd Int'l Conf. on Embedded Software (EMSOFT'02)*. LNCS Series, vol. 2491. Springer, 93–107.

GRUIAN, F., ROOP, P. S., SALCIC, Z. A., AND RADOJEVIC, I. 2006. The SystemJ approach to system-level design. In *4th Intl. Conf. on Formal Methods & Models for Co-Design (MEMOCODE'06)*. IEEE, 149–158.

HAREL, D. 1987. Statecharts: a visual formalism for complex systems. *Sci. of Comp. Progr. 8,* 3, 231–274.

KNAPP, A. AND MERZ, S. 2002. Model checking and code generation for UML state machines and collaborations. In *Proc. of the 7th Intl. Symp. on Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT'02)*. LNCS Series, vol. 2469. Springer, 395–416.

KÖHLER, H. J., NICKEL, U., NIERE, J., AND ZÜNDORF, A. 2000. Integrating UML diagrams for production control systems. In *Proc. of the 22nd Intl. Conf. on on Software Engineering (ICSE'00)*. ACM, 241–251.

LAVAGNO, L. AND SENTOVICH, E. 1999. ECL: A specification environment for system-level design. In *Proc. of the 36th Conf. on Design Automation (DAC'99)*. ACM, 511–516.

LEE, E. A. AND ZHENG, H. 2007. Leveraging synchronous language principles for heterogeneous modeling and design of embedded systems. In *Proc. of the 7th ACM & IEEE Intl. Conf. on Embedded Software (EMSOFT'07)*. ACM, 114–123.

LISKOV, B. AND WING, J. M. 1994. A behavioral notion of subtyping. In *ACM Trans. Program. Lang. and Syst.* Vol. 16. ACM, 1811–1841.

MIKK, E., LAKHNECH, Y., SIEGEL, M., AND HOLZMANN, G. J. 1998. Implementing statecharts in Promela/Spin. In *Workshop of Industrial-Strength Formal Spec. Techniques (WIFT'98)*. IEEE, 90–101.

MILLSTEIN, T. D. 2004. Practical predicate dispatch. In *Proc. of the 19th Annual ACM SIGPLAN Conference. on Object-Oriented Progr., Systems, Langs., and Applications (OOPSLA'04)*. ACM, 345–364.

NIAZ, I. A. AND TANAKA, J. 2004. Mapping UML Statecharts to Java code. In *Proc. of the IASTED Intl. Conf. on Software Engineering*. IASTED Acta Press, 111–116.

PNUELI, A. AND SHALEV, M. 1991. What is in a step: on the semantics of Statechars. In *Proc. of the Intl. Conf on Theoretical Aspects of Computer Software (TACS '91)*. LNCS Series, vol. 526. Springer, 244–264.

RAN, A. S. 1994. Modeling states as classes. In *Proc. of the Technology of Object-Oriented Langs. and Systems Conf.* Prentice-Hall.

ROOP, P. S., ANDALAM, S., VON HANXLEDEN, R., YUAN, S., AND TRAULSEN, C. 2009. Tight WCRT analysis of synchronous C programs. In *Proc. of the 2009 Intl. Conf. on Compilers, Architecture, and Synthesis for Embedded Systems (CASES'09)*. ACM, 205–214.

SAMEK, M. 2002. *Practical Statecharts in C/C++*. CMP Books.

SANE, A. AND CAMPBELL, R. H. 1995. Object-oriented state machines: subclassing, composition, delegation, and genericity. In *Proc. of the Tenth Annual Conf. on Object-Oriented Programming Systems, Langs., and Applications (OOPSLA'95)*. ACM.

SEKERINSKI, E. AND ZUROB, R. 2001. iState: a Statechart translator. In *Proc. of the 4th Intl. Conf. on the Unified Modeling Language (UML'01)*. LNCS Series, vol. 2185. Springer, 376–390.

SREEDHAR, V. C. AND MARINESCU, M.-C. 2005. From Statecharts to ESP: programming with events, states and predicates for embedded systems. In *5th Int'l Conf. on Embedded Soft. (EMSOFT'05)*. ACM, 48–51.

TARDIEU, O. AND EDWARDS, S. A. 2006. Scheduling-independent threads and exceptions in SHIM. In *Proc. of the 6th ACM Intl. Conf. on Embedded Software (EMSOFT'06)*. ACM, 142–151.

TOMURA, T., KANAI, S., UEHIRO, K., AND YAMAMOTO, S. 2001. Object-oriented design pattern approach for modelling and simulating open distributed control system. In *Proc. of the 2001 IEEE Intl. Conf. on Robotics and Automation (ICRA'01)*. IEEE, 211–216.

UML 2011. UML 2.0. `http://www.uml.org`.

VON DER BEECK, M. 1994. A comparison of Statechart variants. In *3rd Intl. Symp. on Formal Techniques in Real-Time and Fault-Tolerant Systs. (FTRTFT'94)*. LNCS Series, vol. 863. Springer, 128–148.

VON HANXLEDEN, R. 2009. SyncCharts in C – a proposal for light-weight, deterministic concurrency. In *Proc. of the ACM Intl. Conf. on Embedded Software (EMSOFT'09)*. ACM, 225–234.