Collective Specification and Verification of Behavioral Models and Object-oriented Implementations *

Qing Yi Jianwei Niu Anitha R. Marneni

University of Texas at San Antonio

Abstract. We present a finite-state-machine-based language, iFSM, to seamlessly integrate the behavioral logic and implementation strategies of object-oriented applications to prevent their design and implementation from being out-of-sync. The language allows developers to focus on higher-level abstractions to support software analysis and design instead of focusing on language or architecture specific details. To support the language, we provide a transformation engine which automatically translates iFSM specifications to lower-level C++/Java implementations of object-oriented classes. The auto-generated implementations from iFSM are similar in style to the manually written code, so that they are readable and easily integrable with the existing code. Our experiments show that their performance is similarly comparable to that of manually written programs. We have automatically verified that these implementations are consistent with their behavioral models by translating iFSM specifications into the input language of model checker NuSMV.

1 Introduction

A large collection of model-driven techniques and tools [1–3, 5, 22] can be used to automatically generate C++/Java code from models in various notations, such as class diagrams, message sequence charts, and statecharts. However, most of these tools require developers to manually provide code in object-oriented (OO) languages to complete the auto-generated code skeletons, where the manual implementations are maintained separately from their higher-level design and can easily become out-of-sync. Further, because compilers and other static analysis tools do not have access to the higher-level specifications, opportunities for performance optimizations and automatic detection of coding errors are missed.

We introduce a finite-state-machine based language, iFSM, to seamlessly integrate software design using FSM-based modeling notations such as HTS [26] with implementations using lower-level languages such as C++ and Java. A key contribution of the iFSM language is a concise mapping from the behavioral model of arbitrary C++/Java classes, expressed using FSMs, to the implementation of these classes, expressed using an implementation specification language that is independent of either C++ or Java. Such a mapping effectively unifies

* This research is funded by the NSF through award CCF0747357 and CNS-0964710.

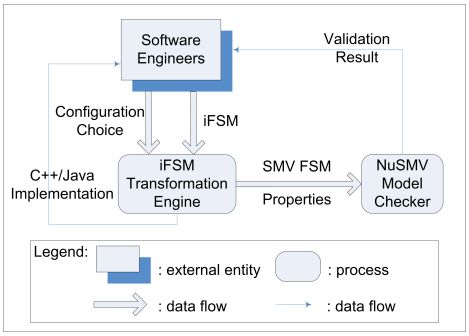

Fig. 1. The iFSM framework

the design and implementation of OO classes within a single language to provide the following benefits.

- The behavioral model and the implementation strategy of object-oriented classes are effectively unified and maintained together. The consistency between them can be automatically verified using model checking techniques.
- Detailed OO class implementations can be automatically generated from iFSM specifications without requiring developers to manually write C++/Java code. Since iFSM specifications can be easily translated to equivalent implementations in different programming languages, they are highly useful when building libraries to work with multi-lingual applications.
- The high-level behavioral model of C++/Java classes can be made available to compilers and other static source-code analysis tools without requiring expensive program analysis. Such domain-specific knowledge can potentially enable a large collection compiler optimizations to more easily overcome the barriers imposed by dynamically bound method calls and therefore be applied in a more aggressive fashion, which is the subject of our future work.

Figure 1 shows the work flow of our framework, where we use a general-purpose transformation engine to automatically translate iFSM specifications to C++/Java classes and to the input language of model checker NuSMV [9]. The auto-generated C++/Java classes are similar in style to manually written code. Therefore, they are readable and can be easily integrated with existing legacy code. Our experimental results have confirmed that their performance is similarly comparable. The unification of behavior and implementation notations within iFSM also allows us to automatically verify the consistency between design and implementation using model checking techniques.

Automatic verification of software implementations is extremely challenging due to the enormous state space required to verify properties of the unlimited amount of memory. The iFSM language aims to bridge the semantic gap between behavior specifications and the corresponding implementations to reduce the difficulty of proving their consistency. As a result, we are able to automatically discover potential errors in program implementations beyond what is supported via conventional approaches.

```
aFSM(Iterator,dstate(Started),tparam(T),inherit()) {
  States: Started,Ended;
  Events: Advance:()->(); Reset:()->(); ReachEnd:()->bool; Current:()->T;
  Trans:  Advance(): Started->(Started,Ended);
          Reset(): (Started,Ended)->Started;
          Current(): Ended->ERROR;
}
```

Fig. 2. Abstract FSM for an *Iterator* interface

2 The iFSM Language

Our iFSM language supports three main concepts, the abstract *FSM* (aFSM), which models the runtime behavior of an object-oriented class (see Section 2.1); the implementation FSM (iFSM), which extends the *aFSM* to additionally model implementation strategies (see Section 2.2); and *iFSM_class*, which specifies how to adapt the interface of automatically generated OO classes when translating iFSM to programming languages such as C++/Java (see Section 2.3).

2.1 The Abstract FSM

An aFSM models the expected behavior of an arbitrary object-oriented class. As illustrated by the *Iterator* interface in Figure 2, each aFSM has a default state (specified by the *dstate* attribute), a list of type parameters (specified by the *tparam* attribute), a list of base FSMs that the current FSM inherits (specified by the *inherit* attribute), and the following additional components.

- **A finite number of control states** which categorize the different values that a runtime object of the FSM may have. Each object must stay in exactly one of the declared states at any time. For example, an object of the *Iterator* FSM in Figure 2 can stay either in state *Started*, which means the object is in normal working condition, or in state *Ended*, which means the object is no longer in operation.
- **A list of events** which define the interface for a runtime object to communicate with the external world. Each event has a list of parameters and can optionally return a value. The *iterator* FSM in Figure 2 has four events, where *Advance* and *Reset* have no input parameter and return nothing; *ReachEnd* takes no parameter and returns a boolean value indicating whether the object is in the *Ended* state, and *Current* returns a value of type T.
- **A list of transitions** which model the change of states within an object as it responds to different events. Each transition has a triggering event, a set of source and destination states, and an optional boolean expression which specifies additional constraints required for the transition. For example, the *Advance* event in Figure 2 will trigger an *Iterator* object to transition from state *Started* to either *Started* or *Ended*, and the *Current* event will raise an exception (enter the *ERROR* state) if the object is in the *Ended* state.

In summary, each aFSM specifies the expected behavior of a C++ abstract class or a Java interface. They may optionally contain some partial implementation specifications (see Section 2.2) but mostly serve as pure interface specifications to support interactions between different software components.

```
1: aFSM(CloneFSM,dstate(),tparam(),inherit())  {Events:Clone:()->ref(fsm(CloneFSM));}
2: iFSM(CountRefHandle,dstate(objNULL),tparam(T:CloneFSM),inherit(),init(),delete(reset)){
3:   Vars:   obj:ref(T)=null; count:ref(int)=null;
4:   States: objNULL: (count==null) -> (obj==null);
5           objUnique: (count!=null&&val(count)==1) -> (obj!=null);
6:          objShared: (count!=null&&val(count)>1) -> (obj!=null);
7:   Events: build:(t:T)->(); const_ref:()->obj; modify:()->obj; reset:()->();
            copy:(that:fsm(CountRefHandle,T))->();
8:          EQ: (that:fsm(CountRefHandle,T))->(obj==that.obj);
9:   Actions:share:(that:ref(fsm(CountRefHandle,T)))->()
                  {obj=that.obj;count=that.count;val(count)=val(count)+1;}
10          init:(t:ref(T))->() {obj=t;count=new(int,1);}
11:         destroy:()->() { delete(count,obj);}
12:  Trans:  build(t):objNULL->objUnique: {init(t.Clone());}
13:         build(t):objUnique->objUnique {destroy(); init(t.Clone());}
14:         build(t):objShared->objUnique {val(count)=val(count)-1;init(t.Clone());}
15:         reset():objUnique->objNULL {destroy();}
16:         reset():objShared->objNULL {val(count)=val(count)-1;obj=null;count=null;}
17:         copy(that) & !in_state(that,objNULL):  objNULL->objShared {share(that);}
18:         copy(that) & !in_state(that,objNULL): objUnique->objShared {destroy();share(that);}
19:         copy(that) & !in_state(that,objNULL): objShared->objShared
20:                     {val(count)=val(count)-1;share(that);}
21:         copy(that) & in_state(that,objNULL): objUnique->objNULL { destroy();}
22:         copy(that) & in_state(that,objNULL): objShared->objNULL
23:             {val(count)=val(count)-1,obj=null,count=null;}
24:         modify():objShared->objUnique {val(count)=val(count)-1;init(val(obj).Clone());}
25: iFSM_class(CountRefHandle) {
26:    constructors: build,copy;
27:    access:       modify:protected;
28:    binding:      reset:dynamic;
29:    extra:        copy=>"operator="; EQ=>"operator=="
30:  }
```

Fig. 3. iFSM for a reference counting C++ class

2.2 The Implementation FSM

As illustrated by Figure 3, each iFSM extends an aFSM to additionally specify implementation details including the following.

Object construction and deletion. For example, at line 2 of Figure 3, *init*() specifies that no parameter is required to build a *CountRefHandle* object, and *delete*(*reset*) specifies that the *reset* event should be triggered when deleting a *CountRefHandle* object.

Member variables of the iFSM. Each variable is declared with a name, a type and an initial value. For example, line 3 of Figure 3 declares two iFSM variables, *obj*, which points to an object of type T, and *count*, which points to an integer. Both variables are initialized with *null*.

The condition and implication of each control state. Each state s in an iFSM contains two boolean expressions, $condition\_of(s)$, which evaluates to true if and only if a runtime object is in the s state; and $imply(s)$, which is guaranteed to evaluate to true if a runtime object is in the s state (i.e., additional properties implied by state s). For example, line 4 of Figure 3 specifies that a $CountRefHandle$ object is in the *objNULL* state if and only if *count==null* is satisfied at runtime; that is, $condition\_of(objNULL)$ is *count==null*. Further, if an object is in the *objNULL* state, then *obj==null* is guaranteed to hold; that is, $imply(objNULL)$ is *obj==null*.

Local details of each event. Each event in an iFSM can include a collection of variables, control states, and transitions that are local to the event; that is.,

type expressions	
fsm(n,targs)	an aFSM/iFSM type with name n and type parameters $targs$
ref(t)	a reference (pointer) type to values of type t
array(t,s)	an array of size s and element type t
expressions	
+,-,*,/,%,	arithmetic operators
==,>=,<=,!=,>,<	comparison operators
&&, ||, !	boolean operators (and, or, and not)
$f(args)$	invoke action/event f using $args$ as parameters
$new(t, o)$	a new object of type t and initialized with value o
$new\_array(t, n, o)$	a new array of size n and item type t, each item initialized with o
$a[s]$	the element at subscript s of array a
$a.b$	the attribute b (e.g.,a variable or event) of an FSM/iFSM object a
val(p)	the value contained in the memory referenced by pointer expression p
ref(v)	the memory reference (address) associated with variable v
in_state(x,y)	whether the $aFSM/iFSM$ object x is in state y
statements	
$delete(p)$	free the memory referenced by p
$except(x)$	raise an exception x
$m = exp$	assign a new value exp to memory expression m.
$LOCAL(x : t = i)$	create a local variable x with type t and initial value i
$LOOP()$	iteratively evaluate autonomous transitions of the current event

Table 1. iFSM Expressions and Statements

each event can contain an embedded iFSM to model any complex algorithm triggered by the event. Note that events cannot be nested inside one another, so local transitions within an event no longer have any triggering event (these are called *autonomous* transitions). In Figure 3, lines 7-8 contain all event definitions, all of which are simple enough that no embedded iFSMs are required.

A list of local actions. Actions are essentially the same as events except that they are private members of the iFSM. Specifically, they are not part of the iFSM interface, cannot be used to as the trigger of any state transition, and cannot be invoked from outside of the iFSM. The iFSM in Figure 3 has three actions, *share*, *init*, and *destroy*, defined at lines 9-11.

The implementation of each transition. Each iFSM transition contains a sequence of statements which modify local variables to implement the transition. All transitions in Figure 3 are at lines 12-24 and are ordered by their triggering events so that the iFSM can be more easily correlated with the corresponding C++ code in Figure 4. The statements currently supported by iFSM are listed in Table 1. Note that transition implementations cannot contain any control-flow statements, so there are no if-conditionals or while-loops in Table 1. This restriction is necessary to simplify the verification of iFSM implementations.

A list of *autonomous* transitions (i.e. transitions not triggered by any event). Each autonomous transition t is controlled by a boolean attribute *repeat* which specifies whether t should be triggered only once or repetitively until the triggering condition no longer holds. If local to an event ev, an autonomous transition t can be triggered by three options, *pre*, where t is evaluated before evaluating any transition triggered by ev; *post*, where t is evaluated after evaluating all transitions triggered by ev; and *loop*, where t is evaluated as part of each *LOOP()* statement (see table 1) invoked by a transition triggered by ev.

Exception handling and debugging support. Each event, action, or transition can specify a list of exceptions together with a sequence of statements

```
template <class T> class CountRefHandle {
  private:
    int* count; T* obj;
    void share(const CountRefHandle<T>& that)
       { obj = that.obj; count = that.count; (*count) = (*count)+1; }
    void init(T* t) {obj=t; count=new int(1);}
    void destroy() { delete count; delete obj; obj = 0; count = 0; }
  protected:
    T* modify()
       { if (count!=0&&(*count)>1) {(*count)=(*count)-1; init((*obj).Clone());}  return obj; }
  public:
    CountRefHandle() : obj(0),count(0)  {}
    CountRefHandle(const T& t) {init(t.Clone());}
    CountRefHandle(const CountRefHandle<T>& that)
       { if (!(that.count==0)) share(that); else { count = 0;obj = 0; } }
    void build(const T& t) {
       if (count==0) init(t.Clone());
       else if (count!=0&&(*count)==1) { destroy(); init(t.Clone()); }
       else if (count!=0&&(*count)>1) { (*count) = (*count)-1; init(t.Clone()); }
     }
    const T& const_ref() const {return (*obj);}
    ......
}
```

Fig. 4. Auto-generated C++ code from Figure 3

iFSM components	C++/Java components
iFSM variables	private member variables in C++/Java classes
iFSM actions	private member functions in C++/Java classes
Nested iFSMs	inner classes nested inside C++/Java classes
iFSM events	public/protected member functions in C++/Java classes
event variables	local variables of the corresponding C++/Java member functions
event-triggered transitions	if-statements inside the corresponding C++/Java member functions
autonomous transitions	loops or if-statements inside the C++/Java class member functions

Table 2. Mapping iFSM to object-oriented C++/Java classes

to handle each exception. Further, each event or action can include a number of debugging declarations, where each debugging declaration specifies what information to print when entering or exiting the event/action.

2.3 Interface Adaptations for C++/Java

Each iFSM component is designed to correlate behavioral notations with the internal implementation details of object-oriented classes in C++/Java. The mapping of concepts between iFSMs and C++/Java classes is shown in Table 2. When translating iFSM specifications to C++/Java classes, actions and events are translated to *private* and *public* member functions respectively. All aFSM events are translated to dynamically-bound public methods. All iFSM actions and events are translated to statically-bound (i.e., non-virtual) methods in C++ but dynamically-bound (i.e., non-static) methods in Java, as these are the default method binding strategies in C++ and Java. The default access control and binding of each method can be overridden via the following interface adaptations, as illustrated by the `iFSM_class` specification at lines 25-30 of Figure 3.

– Extra constructors of the class. In Figure 3, line 2 specifies that the default constructor for the *CountRefHandle* iFSM takes no parameter. However, line 26 specifies that two extra constructors should be defined based on state transitions triggered by the *build* and *copy* events.

– Alternative access control of each event or action. In Figure 3, line 27 specifies that the event *modify* should be made *protected* instead of *public*.
– Alternative binding of each event or action. In Figure 3, line 28 specifies that the event *reset* should be dynamically bound.
– Extra member functions to be generated for a particular event. These extra methods provide alternative names for existing events and do not change the behavioral model or the implementation of an iFSM. In Figure 3, line 29 specifies that an extra name "operator=" should be used for event *copy*, and an extra name "operator==" should be used for event *EQ*.

The `iFSM_class` specification is used to provide users the flexibility of easily adapting the interface of an auto-generated C++/Java class for different needs, e.g., to integrate the auto-generated class with existing legacy code. Figure 4 shows a portion of the C++ code automatically generated from the iFSM specification in Figure 3. The code generation is discussed in more detail in Section 3.

2.4 Expressiveness of the Language

As it stands, our iFSM language serves as a proof-of-concept in collectively specifying and enforcing the consistency between software behavioral design and implementation. It currently supports the following OO concepts in C++/Java.

– C++ abstract classes and Java interfaces, which are modeled using aFSMs.
– Fully implemented C++/Java classes, which are modeled using iFSMs.
– C++ templates and Java generics, which are modeled via type parameters of aFSM/iFSMs.
– Subtyping and class inheritance. Both aFSM and iFSMs can inherit the behavior of other FSMs through the *inherit* attribute. Ambiguity of events, however, is disallowed (treated as errors) when multiple inheritance is present.
– Access control and dynamic/static binding of member functions, which are supported via interface adaption.
– Nested classes, which are supported by nesting iFSMs inside one another.

Table 1 shows the set of expressions and statements currently supported by the language, which are a small subset of those in C++/Java and are expected to be extended when used to model larger and more complex software systems beyond what we have studied.

While incomplete, we believe that the iFSM language has the potential to conveniently specify arbitrary general-purpose OO classes. To demonstrate this potential, we have used the language to fully specify a large and complex C++ class (the original manual implementation of the single class takes 490 lines in C++) that implements the parsing capability of a research compiler project (see Section 5.1). Although compilers and language interpreters are typically sequential and do not need to deal with concurrent evaluation of components, they are among the most challenging software to build, and their implementations typically feature extremely complex and delicate control logics that are easily broken when the code needs to be modified for maintenance (e.g., bug fixing)

```
GenClassImpl(ifsm_decls, adapt_decls)
  globalTable = TypeCheck(ifsm_decls);
  for each (ifsm,adapt) in adapt_decls
     symTab= lookup_symbol_table(globalTable, ifsm);  clsBody = empty; /*body of generated class*/
1.   generate_member_variable_decls(clsBody, symTab, ifsm);
2.   generate_private_actions(clsBody, symTab, ifsm);
3.   for (each inner iFSM m nested inside ifsm): gen_inner_class(clsBody,symTab,m,adapt);
4.   /* map event names to relevant info.*/
     accMap=map_event_to_accessCtrl(adapt);  bndMap=map_event_to_binding(adapt);
     trMap=map_event_to_transitions(ifsm);  autoMap=map_event_to_autoTransitions(ifsm);
5.   gen_default_constructor(clsBody, symTab,ifsm);
     for (each event ev in extra_constructors_of(adapt)):
        gen_constructor_from_event(clsBody,symTab,ev,trMap[ev],autoMap[ev],accMap[ev],bndMap[ev]);
6.   if (ifsm has a destructor event ev)
        gen_destructor_from_event(clsBody,symTab,ev,trMap[ev],autoMap[ev],accMap[ev],bndMap[ev]);
7.   for (each event ev defined in ifsm)
        gen_method_from_event(clsBody,symTab,ev,trMap[ev],autoMap[ev],accMap[ev],bndMap[ev]);
8.   for (each additional method wrapper (ev,name) in adapt)
        gen_method_wrapper(clsBody, ev, name);
9.   output_class_impl(name_of(ifsm),type_param_of(ifsm),inherit_by(ifsm),clsBody);
```

Fig. 5. Generating class implementations

or for functionality enhancement. We have found that by explicitly specifying the behavioral logic of a complex C++ class, the new generated code has better structure and is easier to read and understand. Further, since the behavioral logics of class implementations are made explicit, their consistency can be more easily verified. For more details, see Section 4.

3 Automatically Generating C++/Java Code

We use a transformation engine, shown in Figure 1, to automatically translate iFSM specifications to C++/Java class implementations. The transformation engine is implemented using POET [31], an interpreted program transformation language designed for building ad-hoc translators between arbitrary languages (e.g. C/C++, Java) as well as applying transformations to programs in these languages. Our iFSM translator can be configured via command-line parameters to dynamically produce output in C++, Java, or the input language of the NuSMV model checker, and allows variations of software implementations to be manufactured on demand based on different feature requirements. Figure 4 shows a portion of the C++ class automatically generated by our transformation engine from the iFSM specifications in Figures 3.

3.1 The Code Generation Algorithm

Our algorithm for translating iFSMs to C++/Java classes is shown in Figure 5. The algorithm takes two parameters, a list of aFSM/iFSM declarations (*ifsm_decls*) and a list of interface adaptations (*adapt_decls*). It first applies type checking to verify that all aFSM/iFSMs in *ifsm_decls* are properly defined and constructs a symbol table for each FSM in the process. The algorithm then takes each interface adaptation from *adapt_dcls*, finds the corresponding iFSM, and generates an object-oriented class accordingly.

The *GenClassImpl* routine in Figure 5 essentially follows the mapping rules shown in Table 2 to translate each iFSM to a corresponding C++/Java class. In particular, after setting up the symbol table properly, steps (1-2) of the algorithm translate iFSM variables and actions; step(3) recursively invokes the algorithm to translate nested iFSMs; steps(4-7) translate events and transitions into class member functions; and steps (8-9) post-process the class body (based on interface adaptations) and generate an internal representation of the class implementation, which is then later unparsed with proper syntax in either C++ or Java.

The main task of the algorithm is translating events to class member functions. Here step(4) pre-computes two associative maps for each event: $trMap$, which maps each event to the collection of state transitions triggered by it, and $autoMap$, which maps each event to the pertinent autonomous transitions that are defined either inside the event or inside the surrounding iFSM. Steps (5-7) then combine such information with additional interface adaptation information ($accMap$ and $bndMap$) to translate each event to a member function or a constructor/destructor of the class. Specifically, an if-else-branch is generated for each transition t triggered by an event ev, where the branch-condition considers both the source states of t and any additional constraints associated with t, the true-branch is generated from the statements associated with t, and the else branch includes implementations of other transitions triggered by ev.

Most iFSM statements can be translated to C++/Java statements in a straightforward fashion. A special case is the *LOOP* statement (see Table 1), which is translated to a sequence of loops and if-statements. In particular, for each *loop*-triggered autonomous transition t within the current event, a *while* loop is generated if the source and destination states of t are different or if the *repeat* attribute of t is set to true; otherwise, an if-statement is generated for t. Code generation for *pre*- and *post*-triggered autonomous transitions (see Section 2.2) are supported in a similar fashion, except that these transitions are evaluated at the beginning and exit of the corresponding event or action. Exception handling and debugging are similarly supported and are omitted in the algorithm due to space constraints.

3.2 Correctness of The Generated Code

Our code generation algorithm guarantees that the auto-generated C++/Java code will not incur any compilation error if the given iFSM specifications are properly defined (i.e., all required iFSM components are complete and properly typed); otherwise, the algorithm reports error and aborts. The algorithm loyally follows the translation rules shown in Table 2, which define the operational semantics of the iFSM specifications. Consequently, the generated code is guaranteed to be correct if the input is known to be correct. However, if an input iFSM contains a semantic error, e.g., dereferencing a null-pointer or accessing an array out-of-bound, the error appears accordingly in the C++/Java code. To alleviate this problem, we translate iFSM specifications to the input language of a model checker, NuSMV, and utilize model checking techniques to automatically detect semantic errors in iFSM specifications.

3.3 Quality of The Generated Code

The goal of our iFSM language is to raise the level of abstractions so that software developers can focus on the behavioral design of object-oriented classes and then explicitly map their design to concrete implementation strategies. The design and implementation are collectively specified and maintained together so that they never become out-of-sync. The auto-generated code needs to be easily understandable by both human and compilers, so that it can be seamlessly integrated with existing manually written code and can benefit from the same level of automatic performance optimization by compilers.

As illustrated by Figure 4, which shows a portion of the C++ code automatically produced by our transformation engine from the iFSM specifications in Figure 3, our auto-generated code is similar in style to manually written code, is readable, and can be easily integrated with existing code. Our experimental results show their performance is similarly comparable.

A potential benefit of using iFSM is that by making the behavioral specification of OO classes available to compilers and other program analysis tools, compilers can significantly improve their capabilities to understand the runtime behavior of different OO objects and therefore more effectively optimize interactions between these objects. In particular, since each FSM state can be associated with a collection of important constraints, including aliasing properties of pointer variables, the compiler can make extensive use of such information to enable better optimization of user applications, e.g., to move invariant code out of loops after proving the absence of pointer aliasing between different objects.

4 Verifying iFSM Specifications

A key objective of each iFSM is to unify the behavioral design and implementation details of an OO class so that their consistency can be readily verified. In particular, the control states of each iFSM categorize the different values of local memory references (represented using iFSM variables) that an iFSM object may experience at runtime. As the object goes through various modifications triggered by external event invocations, the modification of memory references must conform to the declared state transitions. We therefore need to verify that the implementation of each iFSM event indeed makes an arbitrary runtime object move from one of the declared source states to a declared destination state. To enable the verification, modifications to the entire collection of iFSM variables must be traced as different events are invoked and relevant state transitions are triggered. We apply model checking techniques to accomplish this purpose.

Model checking techniques [10] have long been used to automatically verify finite state systems, where the reachable state space is exhaustively explored (based on given state transitions) to determine if a given set of properties, typically expressed in terms of temporal logic, hold. In the case that a property fails to hold, the model checker produces a counterexample to show how the failure can arise. We choose to use the NuMSV [9] general-purpose model checker, a

```
    VAR
1:  state : {objNULL,objUnique,objShared}; obj : 0..3; count : 0..3; val_count : -21..20;
2:  copy_that_obj : 0..1; copy_that_count : 0..1; val_copy_that_count : -21..20;
3:  EQ: boolean; const_ref: boolean; reset: boolean; modify: boolean; copy: boolean; build: boolean;

    ASSIGN
4:  init(state) := objNULL; init(count) := 0; init(obj) := 0;
5:  next(state) := case
6:   build&(state=objNULL) : objUnique;  build&(state=objUnique) : objUnique;
7:   build&(state=objShared) : objUnique;  reset&(state=objUnique) : objNULL;
8:   reset&(state=objShared) : objNULL;  copy&(!(copy_that_count=0))&(state=objNULL) : objShared;
9:   copy&(!(copy_that_count=0)&(state=objUnique) : objShared;
10:  copy&(!(copy_that_count=0))&(state=objShared) : objShared;
11:  copy&(copy_that_count=0)&(state=objUnique) : objNULL;
12:  copy&(copy_that_count=0)&(state=objShared) : objNULL;
13:  modify&(state=objShared) : objUnique;  1 : state;
    esac;
14: next(obj) := case
15:  build&obj=0&count=0 : 1;  build&(obj!=0)&(count!=0)&(val_count=1) : 1;
16:  build&(obj!=0)&(count!=0)&(val_count>1) : 1;  reset&(obj!=0)&(count!=0)&(val_count=1) : 0;
17:  reset&(obj!=0)&(count!=0)&(val_count>1) : 0;  copy&(!(copy_that_count=0))&(count=0) : 2;
18:  copy&(!(copy_that_count=0))&(count!=0)&(val_count=1) : 2;
19:  copy&(copy_that_count=0)&(count!=0)&(val_count=1) : 0;
20:  copy&(!(copy_that_count=0))&(count!=0)&(val_count>1) : 2;
21:  copy&(copy_that_count=0)&(count!=0)&(val_count>1) : 0;
22:  modify&(obj!=0)&(count!=0)&(val_count>1):3;  1 : obj;
    esac;
23: next(val_count) := case ..... esac;
24: next(count) := case ..... esac;

25: LTLSPEC  G(state=objNULL -> obj=0&count=0)
26: LTLSPEC  G(obj=0&count=0 -> state=objNULL)
27: LTLSPEC  G(state=objUnique -> (obj!=0)&(count!=0)&val_count=1)
28: LTLSPEC  G((obj!=0)&(count!=0)&val_count=1 -> state=objUnique)
29: LTLSPEC  G(state=objShared -> (obj!=0)&(count!=0)&(val_count>1))
30: LTLSPEC  G((obj!=0)&(count!=0)&(val_count>1) -> state=objShared)
```

Fig. 6. Auto-generated NuSMV input from Fig. 3

BDD-based (symbolic) highly-optimized tool that can handle a relatively large state space. Figure 6 shows the result of translating the *CountRefHandle* iFSM in Figure 3 to the NuSMV input for verification.

4.1 Translating iFSM Specifications to NuSMV

In order to use the NuSMV model checker, we must first translate iFSM specifications to the NuSMV input and then specify properties to verify. The translation aims to simultaneously simulate the state transitions and the corresponding iFSM memory modifications so that their agreement can be verified using temporal logic properties. Figure 6 shows the translation result from the iFSM specification in Figure 3.

A key translation step from iFSM to NuSMV is the conversion of value types, where all non-integer types in iFSM must be converted to an integer type with explicit lower/upper bounds (the only type supported by NuSMV), so that NuSMV can check all values against the proposed properties. To accomplish the conversion, we associate a unique integer to each unknown external memory reference (e.g., an event parameter or the result of a *new* operator) whose value has been used to modify an non-integer-type (e.g., the pointer or array type) iFSM variable. We then use these integers as values for iFSM variables that have

non-integer types. For example, at line 15 of Figure 6, when responding to event *build*, the *next* value for *obj* is set to 1 because the expression $t.Clone()$ used to modify *obj* at lines 12-14 of Figure 3 has been associated with integer 1. For iFSM variables that already have an integer type but can hold an unlimited number of different values, we impose an artificial bound configured via command-line options. In Figure 6, this artificial bound is set to be 20. So both variables *val_count* and *val_copy_that_count* have a value range $-21..20$, where -21 is used to represent all values beyond $-20..20$. The translation approximation, as defined above, could potentially cause NuSVM to report failure with a counter example that does not exist in reality, thus making our verification conservative.

The rest of the translation simply maps each iFSM constituent, e.g., states, variables, events, actions, and transitions, to a corresponding NuSMV component. As illustrated by Figure 6, the resulting SMV code contains the following.

Variable declarations. At lines 1-3 of Figure 6, four different groups of NuSMV variables are created under the `VAR` section, including

- The *state* variable, which is of *enum* type and used to keep track of the control states of an iFSM object;
- The *internal* variables, each of which corresponds to a memory reference used in a boolean expression associated with an iFSM control state. In Figure 6, these variables are *obj*, *count* and *val_count*, where *val_count* corresponds to the memory reference *val*(*count*) used at lines 4-6 of Figure 3.
- The *external* variables, each of which corresponds to a memory reference used in evaluating modifications of the *internal* variables. In Figure 6, these variables are *copy_that_obj*, *copy_that_count*, and *val_copy_that_count*, which correspond to the memory references, *that.obj*, *that.count*, and *val*(*that.count*), used in state transitions triggered by the event *copy* at lines 17-23 of Figure 3.
- The *event* variables, each of which corresponds to an iFSM event and contains a boolean value to keep track of the random invocation of each event. In Figure 6, these variables are *EQ, const_ref, reset, modify, copy*, and *build*, which correspond to events declared in Figure 3.

Variable initializations. At line 4 of Figure 6, the *state* variable is initialized with the default state of the iFSM (i.e., *objNULL*), and the *internal* variables are initialized with their initial values defined in the iFSM. Note that *val*(*count*) is not initialized because *count* is *null*. The *external* and *event* variables are not initialized, so that NuSMV will enumerate all possible values for them.

State modification. At lines 5-13 of Figure 6, the *state* variable is modified based on the state transitions declared at lines 12-24 of Figure 3. In particular, the next value of the *state* variable depends on which event is currently being invoked, the values of event parameters (e.g., *copy_that_count* and *copy_that_obj* at lines 9-12), and the current value of the *state* variable.

Internal variable modifications. At lines 14-24 of Figure 6, the next value of each *internal* variable is computed based on the implementations of event-triggered transitions (i.e., how each transition modifies the *internal* variables).

LTL (Linear Temporal Logic) properties for the NuSMV model checker to verify. This is discussed in Section 4.3.

```
GenNuSMV(ifsm,symTab)
1.  traceThis = memory refs used in boolean expressions associated with states_of(ifsm);
2.  for (each pointer variable x in traceThis): /* compute alias info. of pointer variables */
      tracePtr[x] = stmts that modify x in transitions_of(ifsm);
      aliasMap[x] = memory refs aliased to x by stmts in tracePtr[x];
3.  for (each t in transitions_of(ifsm)): /*compute conditions and side effects of transitions*/
      condMap[t] = condition_of(t) && condition_of(src_states_of(t));
      for (each memory ref x in traceThis):
        modMap[t][x]=stmts in t that modify x or aliasMap[x]
4.  traceExt=empty; /*external memory references that need to be traced by SMV*/
    for (each t in transitions_of(ifsm)): traceExt∪ = external_memory_refs(condMap[t]);
    for (each ref x in traceThis): traceExt∪ = external_memory_refs(modMap[t][x]);
5.  /* generate SMV variable declarations*/
    for (each x in traceThis ∪ traceExt ∪ event_names(ifsm) ∪ {"state"}):
      gen_SMV_variable_declaration(symTab, ifsm, x);
6.  /* generate initialization of state and internal variables */
    for (each x in traceThis ∪ {"state"}) : gen_SMV_variable_initialization(symTab, ifsm, x));
7.  cases = empty; /* generate SMV state modification */
    for (each t in transitions_of(ifsm)): append_SMV_mod_case(cases, condMap[t], dest_states(t));
    gen_SMV_variable_modification("state", cases);
8.  for (each memory reference x in traceThis): /* generate internal variable modifications */
      cases = empty;
      for (each t in transitions_of(ifsm)): append_SMV_mod_case(cases, condMap[t], modMap[t][x]);
      gen_SMV_variable_modification(x, cases);
9.  for each s in states_of(ifsm) /* generate properties */
      gen_SMV_property(name_of(s), condition_of(s));
```

Fig. 7. Algorithm for generating SMV code

4.2 The Translation Algorithm

Figure 7 shows our algorithm for translating iFSM specifications to NuSMV input. The algorithm takes two parameters, *ifsm*, the input iFSM specification to verify, and *symTab*, the symbol table of *ifsm*. The translation process includes the following steps.

Step(1): Collect internal variables, which are variables used in boolean expressions associated with the iFSM states. Save the result to variable *traceThis*.

Step(2): Compute pointer aliasing information. For each pointer variable x in *traceThis*, collect all the values that have been assigned to x by the iFSM statements. Then, examine the collection to extract all the memory references that can be aliased with x.

Step(3): Pre-compute the triggering condition and side effects of each event-triggered transition. In particular, *condMap* maps each transition t to a boolean expression that controls its evaluation, and for each memory reference x in *traceThis*, $modMap[t][x]$ maps t to the new values it could assign to x. Note that *modMap* aims to collect only the last value assigned to each *internal* variable by an event-triggered transition; that is, it does not trace intermediate modifications within transitions (e.g., modifying x to be null before giving it another value).

Step(4): Collect external variables, which include all the external memory references that are used in expressions inside *condMap* or *modMap*. The collection is saved into the *traceExt* variable in Figure 7.

Step(5): Create SMV variable declarations. A variable is declared for each iFSM event, each memory reference in *traceThis* or *tranceExt*, and the special *state* variable which keeps track of the control states of an iFSM object.

Step(5-9): Generate SMV code for variable initialization, modification, and LTL properties, as discussed in Sections 4.1 and 4.3.

4.3 Verifying Properties of iFSM Specifications

Lines 25-30 of Figure 6 list the LTL properties that we verify for the iFSM in Figure 3. Specifically, we generate a property for each control state s to verify that at any time, the *state* variable equals to s if and only if the boolean expression associated with s in the original iFSM specification evaluates to true.

As shown in Figure 6, the auto-generated SMV code separately simulates the state transitions (i.e., the declared behavioral model) and the corresponding iFSM memory modifications (i.e., the model implementation) by tracing the values of the *state* variable and the *internal* variables respectively. The LTL properties essentially reconcile the results of both simulations, thereby verifying the consistency between the behavioral model and its implementation. Once the LTL properties are confirmed by the NuSMV model checker, the input iFSM is guaranteed to satisfy the following constraints.

1. The boolean expressions associated with different control states are mutually exclusive. Specifically, if more than one of the boolean constraints are simultaneously satisfied, the LTL properties would imply that the *state* variable have two different values simultaneously, which is a contradiction.
2. An iFSM object cannot possibly enter any runtime state beyond those explicitly declared in the iFSM. Specifically, if the object enters a state that does not satisfy any of the boolean expressions associated with the declared control states, since the SMV *state* variable always has a valid *enum* value, say s, the property relevant to $state = s$ would have failed the verification.
3. Immediately after initializing all iFSM member variables, an iFSM object is guaranteed to enter its declared default state. If this is violated, the LTL properties pertinent to the iFSM default state will fail.
4. After evaluating the implementation of each iFSM transition t, the resulting new values for the memory satisfy the boolean constraints associated with one of the declared destination states of t. If this is violated, the SMV verification will fail immediately as the value of the *state* variable no longer agrees with the values of the *internal* variables.

In summary, our verification algorithm will detect implementation errors that cause an iFSM object to violate its declared runtime behavior. For example, if *val_count* (dereference of the *count* variable) in Figure 6 becomes < 1 at any point, e.g., due to the programmer forgetting to check the value of $val(count)$ before decrementing it, the verification will detect the error and report the execution path that causes the value of *val_count* to become out-of-sync.

5 Evaluation

Our experimental evaluation aims to confirm two conclusions regarding the iFSM language: (1) the language has the potential to conveniently specify most general-purose C++/Java classes in real-world applications, and (2) the auto-generated

C++/Java class implementations are easily integrable with existing code and are comparable to manually written code in terms of readability and efficiency. To confirm each conclusion, we have taken a number of existing manually written C++ classes and manually generated iFSM specifications for them. We then use our iFSM transformation engine to automatically generate equivalent C++/Java implementations from the iFSM specifications. This approach enables us both to look into the expressiveness of the language in terms of specifying randomly chosen existing C++ classes and to compare the quality of the auto-generated code with existing manual software implementations.

5.1 A Use Case Study

To verify the expressiveness of our iFSM language, we have taken a large, complex C++ class from an open-source compiler project, POET [31], which supports the construction of translators between arbitrary languages (e.g. C/C++, Java) and was used to build our iFSM transformation engine in Figure 1. We choose the POET project because it is implemented in C++, its adaptive parser includes extremely complex control flow that was difficult to understand without documentation, and we can fully understand the code as we were involved in developing the parser. We concede that building iFSM specifications for existing legacy code is difficult, unless developers can fully understand the code. Our future work will look into automatically building partial iFSM specifications to assist program understanding of legacy code.

The single C++ class we took from POET is named *ParseMatchVisitor*, and its manual implementation contains 490 lines of C++ code. This class inherits from two base classes and uses the visitor pattern to dynamically match syntax descriptions of an arbitrary language with a given input token stream in a top-down recursive descent fashion. We have used iFSM to fully specify the behavioral logic of this class together with its implementation strategies, and have regenerated an equivalent class implementation from the iFSM. Essentially, we have reverse engineered the complex control flow within the original C++ code into higher-level behavioral design. The resulting auto-generated C++ code has 540 lines and has confirmed many of our expectations of the iFSM language.

First, when specifying transitions triggered by each iFSM event, we are required to consider both the overall side effects of the corresponding member method and the different runtime situations that may occur when invoking the method. Each situation is then specified using a separate state transition. Because different transitions cannot intermix, operations that are common to them must be re-factored into smaller actions. The separate definitions of transitions and the explicit specification of their source and destination states serve to clearly document the semantic intention of each transition. The end result is a successful unification of the implementation with its higher-level behavior design.

Second, we discovered that cross-cutting concerns are made much more explicit by the iFSM language. In particular, after the beginning and ending states of all transitions are explicitly defined, it becomes much easier to notice common traits of different events. For example, to properly support dynamic parsing, all

events of the *ParseMatchVisitor* iFSM must return empty if the input stream is empty, and when the leading input token is matched against a designated syntax, all events must properly advance the token pointer.

Third, we found that iFSM imposes certain constraints in programming styles which result in better coding structure. In particular, since each iFSM transition must contain a straight sequence of statements as its implementation, and *LOOP()* (see Table 1) is the only statement that can introduce additional implicit control flow, at most two levels of branches can be directly nested inside one another in the auto-generated code. If a deeper nesting of control-flow is required, additional methods (i.e., iFSM actions) must be introduced. Since complex nesting of control flow is a main source of confusion and significantly reduces the readability of programs, the extra member functions have made the auto-generated code much easier to understand than their original version.

In summary, we found that our auto-generated code is easy to read, trivial to integrate with legacy code, and can offer better coding structure than manually written code. When integrated within the POET project, the performance difference between the auto-generated *ParseMatchVisitor* class and the manually written one is indiscernible (below 0.01%). Although the coding structure difference did not have any performance impact, we believe the better support for analyzability by iFSM results in easier verification of program correctness and potentially better automated compiler optimization in the future.

5.2 Performance Comparison

Besides the use case study, we have used iFSM to specify a number of smaller C++ and Java classes, including two finite state machine classes that model the behavior of a heating system, the $CountRefHandle$ class shown in Figure 3, two iterator classes named *SingleIterator* (which supports the iterator interface in Figure 2 for a single item) and $MultiIterator$ (which unifies two iterator interfaces into a single one) respectively, a matrix container named $Matrix$, and a singly-linked list container named *SinglyLinkedList*. Both Java and C++ code are generated for each iFSM except for $CountRefHandle$, where only C++ code is generated because Java does not allow explicit memory deletion by developers.

We aim to confirm that our auto-generated class implementations in general perform just as well as alternative implementations manually written by professional developers. Five of the above mentioned classes, namely $CountRefHandle$, $SingleIterator$, $MultiIterator$, $Matrix$, and $SinglyLinkedList$, are generated based on existing manually written code taken from a C++ compiler project. Since all of our manual implementations are in C++, we compare the performance of auto-generated C++ code with their manually written counter parts. Figure 8 shows the result of comparison.

We have manually written a separate driver program in C++ to measure the performance of each C++ class. Each driver program builds a large number of objects and then invokes the public methods of each object a constant number of times. Therefore, the number of overall object construction and method invocation operations is proportional to the reconfigurable object container size, e.g.,

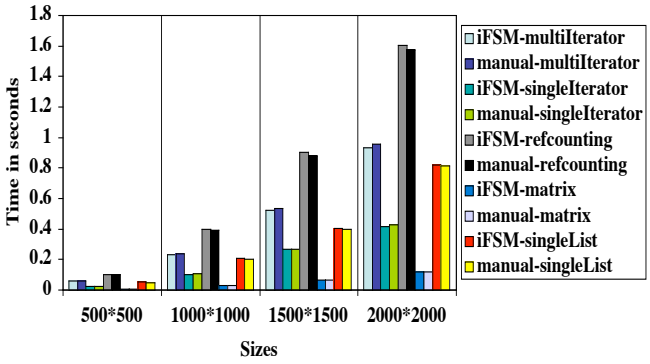

Fig. 8. Performance of C++ implementations using containers of different sizes

a matrix of size 500*500 or a singly-linked list that contains 500*500 items. We have compiled all the C++ code using g++ 4.2.0 with the -O2 compilation flag. The elapsed time of each evaluation is measured on an Intel 2.16 GHz Core2Duo processor with 1GB 667MHz memory and 4MB L2 cache. The performance results are shown in Figure 8.

From Figure 8, we see that all the auto-generated C++ classes from iFSM perform similarly as the manually written ones. In particular, the auto-generated iterator classes consistently performed slightly better than the manually written ones, while the auto-generated reference counting and singly-linked list classes performed slightly worse than the manually written ones. The auto-generated and manually written matrix classes have performed almost identically.

Since the differences in performance are minor between the iFSM-generated class implementations and the manually written ones, they are likely caused by random factors in the compiler. The main benefit to gain from the iFSM specifications is the automatic verification of consistency between implementation details and behavioral design, and the potential of compilers utilizing the behavioral information of object-oriented classes to improve the efficiency of their interactions. Our future work will integrate such information within our compiler infrastructure to significantly improve application performance.

6 Related Work

Model-driven development [20] captures important aspects of a software system through models [14, 15] before producing lower-level implementations of the system [1–3, 5, 22]. In particular, finite-state-machine-based notations have long been used in previous research to model the dynamic behavior of large reactive systems [16, 17], and many research projects have automatically produced code to simulate the behavior of various state machine models [21, 25, 28, 30].

Runtime behavior modeling, however, is only one aspect of software development. Unless the behavioral notations are correlated with other aspects of software implementation, e.g., data structures, algorithms, memory management,

and integration of different software components, the behavioral notations are merely artifacts of the software design phase and have to be kept separate from complete implementations of software systems. Consequently, although many software development tools exist to automatically generate C++/Java code from various aspects of software design, the auto-generated code either is used for software design only or requires developers to provide additional implementation details by manually modifying auto-generated code skeletons. Since manual implementations are maintained separately from higher-level modeling notations, the models and implementations can easily become out-of-sync. Our iFSM language offers a way to unify behavioral modeling notations and implementation strategies so that they can be maintained together, and their agreement can be automatically verified.

By unifying behavioral modeling with domain-specific implementation specifications, previous research has produced efficient finite-state-machine implementations in some specialized domains, e.g., embedded systems [29] and lexer/parser generation [23]. In essence, our iFSM language is in some aspects similar to how Lex/Yacc [23] correlates the syntax specifications of input languages with concrete C/C++ implementations to automatically generate high-performance lexers/parsers for compilers. We share the goal of automatically producing highly efficient full-blown software implementations. Our work is different from these domain-specific code generators in that we target general-purpose object-oriented C++/Java code, and we automatically verify the consistency between the implementation specifications and the corresponding behavioral design.

Program transformation tools have long been used to analyze and modify existing software implementations, including re-documenting/re-implementing code, reverse engineering, changing APIs, and porting to new platforms [6, 13]. Several general-purpose transformation languages and systems have been developed [18, 12, 7, 4] and some have been widely adopted [7, 4]. These tools and systems mostly rely on pattern-based transformation rules coupled with application strategies. They do not use modeling notations and are typically not concerned with the consistency between software design and implementations. We focus on combining program transformation with software verification technology to better support both the correctness and the efficiency of generated code.

Formal methods have been widely used to verify the correctness of both software design and implementations [10, 27, 8, 24, 11, 19]. In particular, a number of projects can effectively verify important properties of software implementations from analyzing the source code [11, 8]. However, these projects must apply sophisticated decomposition, abstraction, or program slicing techniques to prune the enormous modeling space required for verifying whole user applications. While more appealing, directly verifying low-level software implementations is in general extremely challenging due to the unlimited memory references dynamically modified by user applications. Our iFSM language explicitly categorizes the unlimited memory modifications using a finite number of runtime state transitions and ensures that each transition is implemented with minimal control flow. As a result we can much more readily bridge the semantic gap

between the behavior properties and the implementation details. For example, if a variable x is modified within a transition, a small set of unique expressions can be determined to be the new values for x, which is typically not possible when directly verifying programs in lower-level languages such as C++/Java. It may be possible to embed iFSM annotations inside existing implementations to enable more effective verification, but this belongs to our future work.

7 Conclusions

This paper presents a high-level specification language, iFSM, to effectively unify the behavioral design of object-oriented classes with detailed implementation strategies, so that efficient and complete C++/Java code can be automatically generated, and the consistency between design and implementation can be automatically verified. Our experimental results confirm that the auto-generated classes are comparable in performance to manually written code. The conformance between the implementations and their behavioral models have been automatically verified using model checking techniques.

Our iFSM language has the potential to significantly improve both the error detection and runtime efficiency of object-oriented applications in C++ and Java, without sacrificing software readability or modularity. In particular, the unification of behavioral and implementation notations has made software implementation details much easier to reason about and verify, and their higher-level behavioral model can be made readily available to compilers and other source-code analysis tools. Such knowledge can potentially enable a large collection of compiler optimizations to be applied in a much more aggressive fashion, which is the subject of our future work.

8 Acknowledgement

We would like to thank Dr. Macneil Shonle at UTSA for offering insightful suggestions to improve our paper.

References

1. Implementing UML statechart diagrams. www.PathfinderMDA.com.
2. Metamill 4.0. www.metamill.com.
3. Umodel. Altova Inc. http://www.altova.com/.
4. O. S. Bagge and et. al. Design of the CodeBoost transformation system for domain-specific optimisation of C++ programs. In *SCAM*, pages 65–75, 2003.
5. K. Balasubramanian and et. al. Applying model-driven development to distributed real-time and embedded avionics systems. *International Journal of Embedded Systems. Special issue on Design and Verification of Real-time Embedded Software*, April 2005.
6. I. Baxter, P. Pidgeon, and M. Mehlich. Dms: Program transformations for practical scalable software evolution. In *ICSE*, 2004.

7. M. Bravenboer and et. al. Stratego/XT 0.17. A language and toolset for program transformation. *Science of Computer Programming*, 2008.
8. S. Chaki, E. Clarke, and A. Groce. Modular verification of software components in c. *Transactions of Software Engineering*, 1(8), Sept. 2004.
9. A. Cimatti and et. al. NuSMV Version 2: An OpenSource Tool for Symbolic Model Checking. In *CAV*, volume 2404 of *LNCS*, July 2002.
10. E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 1999.
11. M. Das and et. al. Esp: path-sensitive program verification in polynomial time. In *PLDI '02*, pages 57–68, 2002.
12. M. Erwig and D. Ren. A rule-based language for programming software updates. *SIGPLAN Not.*, 37(12):88–97, 2002.
13. Y. Futamura, Z. Konishi, and R. Glück. WSDFU: program transformation system based on generalized partial computation. *The essence of computation: complexity, analysis, transformation*, 2002.
14. J. A. Goguen and R. M. Burstall. Institutions: abstract model theory for specification and programming. *J. ACM*, 39(1):95–146, 1992.
15. J. Gray, T. Bapty, and S. Neema. Handling crosscutting constraints in domain-specific modeling. In *CACM*, pages 87–93, October 2001.
16. D. Harel. Statecharts: A visual formalism for complex systems. *Science of Comp. Prog.*, 8(3), 1987.
17. D. Harel and A. Naamad. The statemate semantics of statecharts. *ACM Trans. Softw. Eng. Methodol.*, 5(4):293–333, 1996.
18. S. S. Huang, D. Zook, and Y. Smaragdakis. Statically safe program generation with safegen. In *Generative Programming and Component Engineering*, 2005.
19. M. Kawaguchi, P. Rondon, and R. Jhala. Type-based data structure verification. In *PLDI '09*, pages 304–315, 2009.
20. A. Kleppe, J. Warmer, and W. Bast. *MDA Explained: The Model Driven Architecture Practice and Promise*. Addison Wesley, 2003.
21. A. Knapp and S. Merz. Model checking and code generation for uml state machines and collaborations. In *Proc. 5th Wsh. Tools for System Design and Verification*, pages 59–64, 2002.
22. A. Kogekar and et. al. Model-driven generative techniques for scalable performability analysis of distributed systems. In *NSF NGS Workshop of IPDPS*, 2006.
23. J. R. Levine, T. Mason, and D. Brown. *Lex & Yacc*. O'Reilly & Associates, 1992.
24. G. C. Necula. Proof-carrying code. In *POPL'97*, pages 106–119, 1997.
25. I. A. Niaz and J. Tanaka. Mapping uml statecharts to java code. In *IASTED International Conference on Software Engineering*, pages 111–116, 2004.
26. J. Niu, J. M. Atlee, and N. A. Day. Template semantics for model-based notations. *IEEE Transactions on Software Engineering*, 29(10):866–882, October 2003.
27. S. Owre, J. Rushby, and N. Shankar. PVS: A prototype verification system. In *CADE*, 1992.
28. A. Prout, J. M. Atlee, N. A. Day, and P. Shaker. Semantically configurable code generation. In *MoDELS*, pages 705–720, 2008.
29. A. Wasowski. On efficient program synthesis from statecharts. In *LCTES*, pages 163–170, 2003.
30. M. W. Whalen. High-integrity code generation for state-based formalisms. In *ICSE '00: Proceedings of the 22nd international conference on Software engineering*, pages 725–727, New York, NY, USA, 2000. ACM.
31. Q. Yi, K. Seymour, H. You, R. Vuduc, and D. Quinlan. POET: Parameterized optimizations for empirical tuning. In *Workshop on Performance Optimization for High-Level Languages and Libraries*, Mar 2007.