Automatic Generation of Program Families by Model Restrictions

Andrzej Wąsowski

Department of Innovation
IT University of Copenhagen
2300 Copenhagen S, Denmark
`wasowski@itu.dk`

Abstract. We study the generative development of control programs for families of embedded devices. A software family is described by a single common model and restriction specifications for each of the family members. The model and the specifications are used in the automatic generation of restricted programs. We describe an application of the process of modeling reactive systems with statecharts. A trace inclusion refinement relation is established for automatically generated family members, inducing a behavioral inheritance hierarchy over the generated programs.

1 Introduction

Manufacturers commonly face the problem of developing multiple versions of control programs for similar electronic devices. The programming languages employed in the development of such program families—typically assembly languages, C, and state-machine-based formalisms—poorly support code reuse and extension, while the object-oriented technology is hardly accepted in the development of resource-constrained systems, because of its size and speed overheads. As a consequence, maintaining even three versions of a simple product, such as home appliances or mobile devices, remains a troublesome task.

Generative programming [4] has been approaching the very same problem for some years now, but usually for bigger systems, focusing on highly customizable, component-based sophisticated products like cars and satellites (see an example in the work of Czarnecki and associates [3]). We shall discuss a class of program families, suitable for describing simple products and propose a suitable development strategy based on model restriction. The main assumption is that all family members can be generated from one model by means of simple transformations (restrictions). The process of restriction is driven by short terse specifications that, themselves, can be arranged in a behavioral hierarchy similar to a class hierarchy of object-oriented programming. The ultimate gain is that the production of many new family members and maintenance of the whole family become significantly cheaper than in the classical top-down approach.

The paper is structured as follows. Section 2 introduces the basic concepts. Section 3 describes the technical details of restrictions using the language of statecharts. Section 4 sketches the formal semantics, while Sections 5 and 6 introduce restriction transformations and their correctness in a sense of behavioral refinement. An extension of the framework—namely support for more diverse program families—is discussed in section 7. Finally, Sections 8 and 9 discuss related work, conclusions, and future work.

2 Development by Restriction

Consider product families for which a total ordering can be introduced such that the simplest member of the family p_1 is the least element and the most advanced p_n is the greatest one:[1]

$$p_1 \preceq p_2 \preceq \ldots \preceq p_n$$

For each non-greatest element p_i, its successor p_{i+1} is a product that can be obtained solely by adding features to p_i, that is, by extension of p_i. No modifications of features already existing in p_i are allowed. The software for such a product family may be developed using incremental object-oriented programming. The simplest device p_1 is implemented first as the base class c_1. Then, additions are made, building an inheritance hierarchy with more advanced devices implemented by subclasses. Each class c_i adds new features to the implementation. The topology of the hierarchy strongly resembles the topology introduced on the product family. Since the ordering on classes in this example is total, the class hierarchy will actually be linear:

$$c_1 \leftarrow\!\!\lhd\; c_2 \leftarrow\!\!\lhd\; \ldots \leftarrow\!\!\lhd\; c_n .$$

This can be generalized easily to program families structured in lattices, where arbitrary hierarchies emerge.

Consider the dual of the extension relation—the reversed relation of *restriction.* One can view the ordering over the family in an alternative way: a predecessor p_{i-1} of each non-least element p_i would be obtained solely by *removal* of features from p_i. This yields a reversed implementation strategy: start with the development for the most advanced device p_n and obtain the rest by stripping parts away (see Fig. 1). This idea, although initially nonintuitive, exhibits a number of advantages attractive for the production of embedded software.

The main reason for adopting restriction as a program refinement operator is the possibility of automating the development process. The design of the greatest product and selection of restriction steps are the only necessarily manual steps. The production of restricted family members can be fully automated. This cannot possibly be achieved for program extension. An addition of a new family member in an object-oriented architecture demands at least an implementation of a new class. In the restriction approach, the generation of a new product can be done automatically, if the product is simpler than already implemented. If the product needs to incorporate genuinely new features, the addition is not much

[1] We shall consequently use the words *least* and *greatest* only in the sense of ordering over the products and not in the sense of any physical properties.

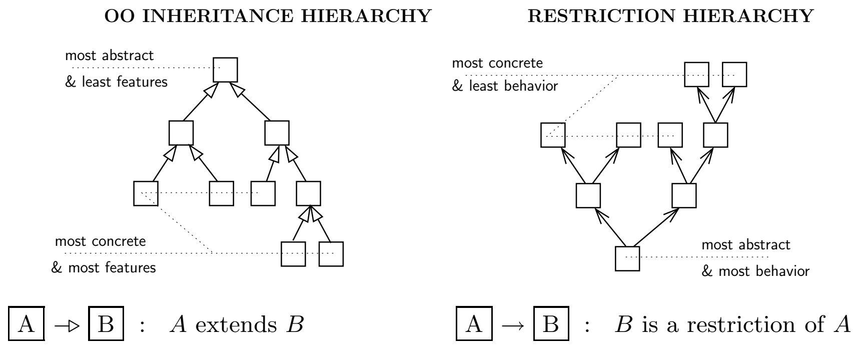


Fig. 1. A comparison of extension and restriction hierarchies. Left: the hierarchy of classes in incremental programming. The simplest class is the root. The most able classes are leaves. Right: the hierarchy of programs in decremental programming. The most able program down with leaves being the simplified versions on top

harder than in the object-oriented case. Automation makes prototyping new family members extremely cheap, which substantially aids business and technical decisions. A product manager can use a prototype of a new family member for the evaluation of its usability and merchantability. An engineer can use fast prototyping to select a set of features that most efficiently utilize resources of the hardware platform available for the specific version of the product.

The main arguments in favor of restrictions—efficiency and correctness—are also crucial in the development of resource-constrained embedded systems. Efficiency requirements (memory consumption mostly) still prevent the application of object-oriented programming and partial evaluation in this area. In contrast, restriction introduces neither runtime overheads nor any code growth and may be seen as an optimization pass before the code generation. Restriction addresses correctness issues, too, guaranteeing that behavioral refinement holds between related members of the family. If p_1 is a simpler derivative of p_2 ($p_1 \preceq p_2$), p_1 and p_2 behave in exactly the same way if run in the environment of p_1. Similar properties cannot be established easily for programming by extension.

We shall follow Parnas [16] in distinguishing a *product family* denoting a set of similar hardware devices and an isomorphic *program family* implementing control algorithms for these devices. In addition to that, a layer of *model families* is considered. Models are executable abstractions of programs that can be used for automatic synthesis. Members of the model family are generated from a single manually designed greatest model. The generation is driven by *restriction specifications*—a set of constraints imposed on the original model defining the expected configuration of features. Figure 2 illustrates the process for a simple example: a line of three compact disk (CD) players.

As we shall see later on, a restriction specification is a simple model (or type) of the environment in which the restricted software will be operating. It can include user behaviors, but, in most applications, it will describe the hardware

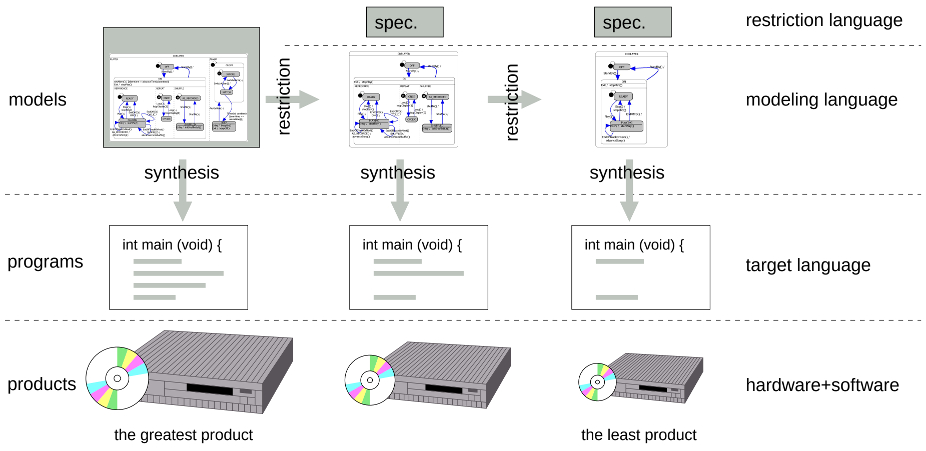


Fig. 2. Layers of development process for a simple family of three CD players. Gray shaded rectangles represent manually developed parts. Arrows represent automatically performed steps

in which the program will be embedded. The hardware will limit users' behaviors indirectly (i.e. the users cannot press a nonexistent button or observe an output of a nonexistent actuator).

In principle, there is no requirement that the hardware architecture is identical in all members of the product family. As long as a compiler is available for all the hardware platforms, it is still possible to support them within the same family. However, the gain is significantly higher for more homogeneous families, based on microcontrollers from the same family. In such cases, not only the control code but also the drivers for sensors and actuators can be reused. Homogeneous families are of our primary interest.

3 Restriction for the Development of Reactive Systems

Embedded appliances are often described as reactive synchronous systems [8], namely programs that continuously receive inputs and react by producing outputs. All outputs are emitted before the arrival of the subsequent input (so called *synchrony hypothesis* [1]). The inputs are environment events transmitted by sensors. The outputs are commands to influence the environment by means of connected actuators. A reactive system maintains an internal state during operation. Emission of the outputs and changes to the internal state depend solely on incoming events and the previous state of the system.

Reactive systems are conveniently described in highly imperative modeling languages with explicit control flow [6, 1], usually exhibiting a close coupling of functional features and their implementations. Entities like inputs and outputs can be closely related to features they activate or perform. Let us consider a simple example of a CD player to illustrate this. The fundamental functionality of a CD player (see Fig. 3) is controlled by play and stop buttons. Additionally, the player has a built-in alarm clock that is related to the two buttons controlling the setup and the two actuators presenting the state of the alarm—a beeper and a small display indicator.

A set of inputs can be used to describe an implementation of a feature. The two events corresponding to the two buttons used to control the alarm clock define a part of the model (a *slice*) that implements the setup of the alarm clock. Removal of this slice leads to a simpler model that does not allow any control over the alarm clock. Similarly, the slice for the alarm state display and the beeper constitute the part of the model implementing the entire alarm clock feature. If the display's and the beeper's influence on the environment can be ignored, the alarm's entire implementation can be eliminated from the device without any loss. The slices defined by various inputs and outputs may overlap depending on the actual model. A sum of several slices may comprise the implementation of more sophisticated features.

3.1 Modeling Language

We shall study the implementation of restriction for the language of *statecharts* [6, 15], the most popular language used in modeling discrete control. Several distinctive features of the language can be indicated on Fig. 3: nesting of states (hierarchy), explicit control-flow (transitions), and concurrency. Note that some inputs come from the environment (*Stop, Play, Next, StandBy*), and some come from the hardware itself (e.g., *EndOfTrack*). Despite this difference, all kinds of events are considered external inputs from the software's point of view.

The particular version of statecharts studied here [20, 11] uses C as a host language.[2] It allows only simple integer and vector types for variables of the model, disallowing pointer arithmetic and aliasing. Transition guards follow a simple grammar:

$$\begin{aligned} \textit{guard} \longrightarrow{}& (\mathsf{event}\ (\texttt{"}\vee\texttt{"}\ \mathsf{event})^*)\ \texttt{"}\wedge\texttt{"}\ \mathsf{state}\ (\texttt{"}\wedge\texttt{"}\ \mathsf{state})^* \\ & (\texttt{"}\wedge\texttt{"}\ \texttt{"}\neg\texttt{"}\mathsf{state})^*\ (\texttt{"}\wedge\texttt{"}\ \textit{pure-C-cond})? \end{aligned}$$

where event stands for an event identifier, state for a state identifier, and pure-C-cond for any side-effect-free arithmetic expression written in C. Transition actions are assumed to be relatively pure with respect to each other, which means that the order of transition firing is insignificant.

Statecharts incorporate model variables and external function calls. Both can be used to model communication with the external world. Unfortunately, both can be used to model output and input as well. A variable may be bound to

[2] The choice of C as the host language is rather arbitrary. It has been enforced by the modeling tool IAR visualSTATE, which we use in the project. Other host languages could be used, including specification languages like Object Constraint Language (OCL). The host language does not have to be identical with the target language, but such a situation is obviously most convenient for tool developers.

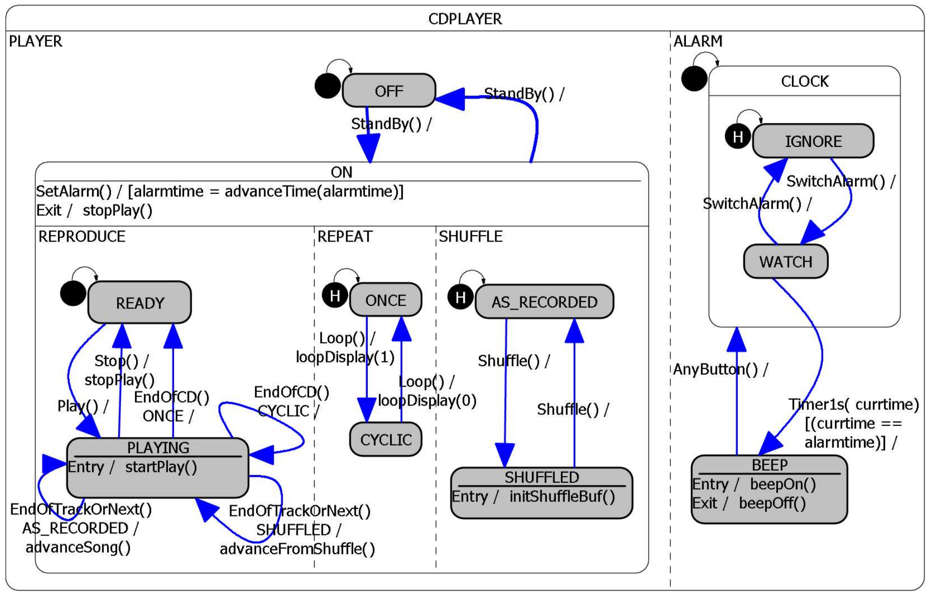


Fig. 3. A CD player modeled in IAR visualSTATE. Transitions are labeled by events, guards and outputs. Guards are optional and range over states and variables. Events *StandBy, SetAlarm, Stop, Play, Loop, Shuffle, SwitchAlarm, Next* correspond to buttons of the user interface. Events *EndOfCD*, *EndOfTrack* and *Timer1s* are internally produced by the hardware. *AnyButton* is in fact a disjunction of all button events. *EndOfTrackOrNext* is a disjunction of *Next* and *EndOfTrack*. Outputs are optional and preceded by slash. They correspond to driver calls: *advanceTime, stopPlay, startPlay, advanceSong, advanceFromShuffle, loopDisplay, initShuffleBuf, beepOn* and *beepOff*

a hardware port representing a sensor or the command entry of an actuator. Similarly, a function call can be used to obtain values of sensors, or to control actuators. Furthermore, some variables and functions participate only in the internal computation of the model. Consequently, the syntactic categories can hardly be used to discover the meaning of a variable or a function. To remedy the confusion, we shall assume that events, variables, and functions are divided in three disjoint classes: (1) sensors (modeled by events, read-only variables, and pure functions), (2) actuators (modeled by write-only variables and non-pure functions), and (3) elements of internal logics of the model.

3.2 Restriction Language

We shall now consider restricted versions of the CD player and describe them in Restriction Language (RL), the language used to express restriction specifications. The fundamental concepts of RL are constraints on variables, functions, and events; hierarchical restrictions of interfaces; and the binding of interfaces to models in the family.

Assume that e is an event in the statechart model. Then, the meaning of an *impossibility constraint*

```
impossible e;
```

is that event e is not relevant for the restricted model and, thus, may be ignored. A version of the CD player without the shuffle feature can be specified by

```
impossible Shuffle;
```

A *purity constraint* may be written for a function f:

```
pure void f();
```

The meaning is that calls to f do not cause side effects (are not observable) and can be removed from the restricted model. The purity constraint is, thus, useful to eliminate outputs of the program in the absence of actuators connected to these outputs.

Groups of constraints are combined together in interface restrictions. A somewhat handicapped CD player with built-in alarm, but without any beeper can be described as follows:

```
restriction NoBeep {
        pure void beepOn();
        pure void beepOff();
};
```

New members of a model family are defined through the binding of model names to restrictions:

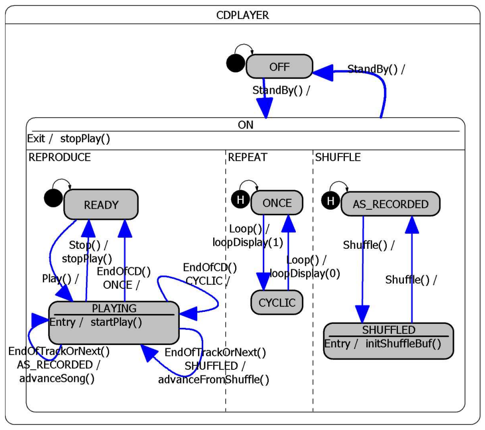
```
InterfaceName ModelName;
```

Figure 4 shows the restriction specification and the corresponding model of a CD player without any functionality related to the alarm clock.

Interface restrictions may be arranged in hierarchies by means of restriction inheritance. Any interface restriction can inherit declarations from one or more previously defined restrictions:

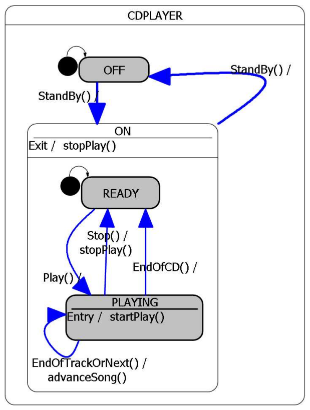
```
restriction Name restricts ancest_1, ..., ancest_n
{ ... };
```

Figure 5 presents the very simplest player with no available alarm, shuffle, or looping features. Both the interface restriction and the model are obtained by further restricting the specification and the model of Figure 4. Transitions fired by Loop and Shuffle have been removed. Reachability analysis removes states CYCLIC and SHUFFLED, which allows the removal of more transitions in the REPRODUCE state. Finally, states ONCE and AS_RECORDED can be removed because they don't affect transitions or have any embedded side effects.



```
restriction WithoutAlarm {
    impossible SetAlarm();
    impossible SwitchAlarm();
};
WithoutAlarm CDPLAYER;
```

Fig. 4. CD player without the alarm clock: a restriction specification (right) and the corresponding model obtained by restriction (left)



```
restriction Least restricts WithoutAlarm {
        impossible Loop();
        impossible Shuffle();
};

Least CDPLAYER;
```

Fig. 5. Simple CD player with neither the alarm clock nor the shuffle, and continuous play functions: a restriction specification and an automatically generated model

```
       start  ⟶  (restriction  | binding)*
     binding  ⟶  restriction-id system-id ;
 restriction  ⟶  "restriction" restriction-id
                         ("restricts" restriction-id ("," restriction-id)*)?
                 "{" constraint* "}" ";"
  constraint  ⟶  "impossible" event-id ";"
               | "const" type-id var-id "=" value ";"
               | "const" type-id fun-id "(" param-types ")" "=" value ";"
               | "dead" type-id var-id  ";"
               | "pure" type-id fun-id "(" param-types ")" ";"
 param-types  ⟶  type-id ("," type-id)*
```

Fig. 6. Grammar of RL specifications

For any given variable v, one can consider a value constraint and a liveness constraint:

```
const int v = 1; /* value constraint */
dead int v;      /* liveness constraint */
```

A value constraint substitutes a variable with a constant. A liveness constraint marks a variable as dead, which, in turn, means that assignments to that variable can be discarded. A useful application of a value constraint is to limit the value of a sensor. A liveness constraint may be used to indicate the absence of an actuator bound to the port represented by v. There is a similar value constraint for functions:

```
const int f(int) = 4; /* value constraint for function */
```

This specification states that f always returns 4 in the restricted model. Note that it implicitly says that f is a pure function. Value constraints should only be used for sensors, which must always be pure by the semantics of statecharts.

Figure 6 summarizes the description of RL, giving its grammar.

4 Semantics of RL

We shall model the semantics of statecharts using execution traces [10]. Traces are finite sequences of events. If $e_1, \ldots, e_n$ denote events, $\langle e_1, \ldots, e_n \rangle$ is a trace consisting of their occurrences. If t is a trace and A is a set of events, $t \upharpoonright A$ denotes t after restricting it to members of A (all nonmember events are omitted). Concatenation on traces is written $s\hat{\ }t$. Finally, the inclusion of traces is defined to be

$$s \ \mathbf{in}\ t \ \equiv\ (\exists p, q.\ t = p\hat{\ }s\hat{\ }q)$$

Let us assume that the semantics of model m is given by a set of all its execution traces over an alphabet αm:

$$\mathit{traces}(m) \;\subseteq\; (\alpha m)^*$$

The alphabet includes all the input events (discrete events and readings of sensors) and all the output events (control commands for actuators) of m. Discrete events of m are represented directly as members of αm. All reads from and all writes to a variable v are visible in the trace, written $v.l$, where l is the value read or written. Calls to a function f are members of αm, too. A call to f that returns a value l is written $f.l$. Finally, all the events belonging to internal logics of the model—changes in local variables, triggering local signals, and calls to local computations—are excluded (concealed) from αm.

Our models are reactive, so they never generate actions autonomously. Formally, it means that all traces start with an input event. Each input event is followed by a possibly empty subsequence of output events.

Knowing that, we can give the semantics of restriction. First, the restriction hierarchy shall be flattened. An interface restriction `A` that further restricts `B` is equivalent to all constraints of `B` being included in `A`. The flattening is applied transitively to all interface restrictions defined in the specification. Simple consistency checks are carried out simultaneously to flattening:

i. The types of events, variables, and functions must be identical in all constraints of a given element (invariance).
ii. If there are multiple value constraints for the same variable or function, all constraint values must be identical (no contradiction).

The semantics of a flattened interface restriction r is defined with respect to a model m, which is syntactically consistent with r (m defines all the elements referred to in r with the same types as in r). We shall only give the semantics for restrictions over events and over variables and functions that describe sensors and actuators. The semantics is given as a function on sets of execution traces:

$$RL[\![\ \cdot\ ]\!]_m \;:\; \mathcal{P}(\mathit{traces}(m)) \longrightarrow \mathcal{P}(\mathit{traces}(m))$$

Impossibility constraints only allow execution traces without impossible events:

$$RL[\![\ \texttt{impossible e;}\ ]\!]_m \;=\; \lambda S.\ \{\ t \mid t \in S \;\wedge\; \neg(\langle e\rangle\ \mathbf{in}\ t)\ \}$$

The two types of constraints over sensors only allow traces with sensor reads returning the actual value of a constraint:

$$RL[\![\ \texttt{const T v = k;}\ ]\!]_m \;=\; \lambda S.\ \{\ t \mid t \in S \;\wedge\; \forall\langle v.l\rangle\ \mathbf{in}\ t.\ l = k\ \}$$
$$RL[\![\ \texttt{const T f() = k;}\ ]\!]_m \;=\; \lambda S.\ \{\ t \mid t \in S \;\wedge\; \forall\langle f.l\rangle\ \mathbf{in}\ t.\ l = k\ \}$$

Constraints over actuators do not remove entire traces. They remove single calls to actuators instead:

$$RL[\![\ \texttt{dead T v;}\ ]\!]_m \;=\; \lambda S.\ \{\ t \upharpoonright (\alpha m \setminus \{v.l \mid \forall l \in \mathit{values}(T)\})\ \mid t \in S\ \}$$
$$RL[\![\ \texttt{pure T f();}\ ]\!]_m \;=\; \lambda S.\ \{\ t \upharpoonright (\alpha m \setminus \{f.l \mid \forall l \in \mathit{values}(T)\})\ \mid t \in S\ \}$$

Finally, several constraints may be combined using function composition:

$$RL[\![\ c_1\ \ c_2\ ]\!]_m \ = \ RL[\![\ c_2\ ]\!]_m \cdot RL[\![\ c_1\ ]\!]_m$$

The composition of constraints is commutative, since we have allowed value constraints only for sensors, and purity and liveness constraints only for actuators.

As an example, consider three extreme restriction specifications. ***`Universal`*** is an empty specification. It does not modify the original model at all:

$$\forall m.\ RL[\![\ \texttt{Universal}\ ]\!]_m traces(m) = traces(m)$$

`Inactive` models an environment, which never generates any inputs (all inputs are `impossible`). It can be easily observed that

$$\forall m.\ RL[\![\ \texttt{Inactive}\ ]\!]_m traces(m) = \emptyset$$

Finally, ***`Blind`*** models an environment that does not care about any outputs of the system (all outputs are `pure`):

$$\forall m.\ RL[\![\ \texttt{Blind}\ ]\!]_m traces(m) = traces(m) \restriction (\alpha m \setminus Out_m),$$

where Out_m denotes all possible output actions of m. Note that both ***`Inactive`*** and ***`Blind`*** allow an empty model as a result of restriction over any statechart.

The semantics of RL has been given only for nonparameterized events and outputs. The parameterized events and outputs are not a problematic extension. They have been avoided in RL only for clarity of the presentation.

5 Model Transformations

Model restriction can be implemented as part of the framework for model-based development. A code generator can be divided into two parts: (1) the specializer of models and (2) the synthesizer of programs. The technique is being developed as part of the SCOPE [17] tool, which also incorporates a full C-code generator for statecharts [19].

In principle, any model transformation that preserves the semantics under conditions of restriction specification can be used as a restriction function. However, for the purpose of the current investigation, we limit ourselves to functions that indeed decrease the model in a syntactical sense (i.e., the only transformations they perform are the elimination of model elements and substitutions of variables and function calls by constants). We relax this requirement by allowing an optimization pass to be performed over the resulting restricted model:

$$\omega : Model \longrightarrow Model$$

The type of the optimizations used depends on the application domain. If the reaction time is the main concern, speed optimizations should be applied. Size optimizations should be used for embedded systems with limited memory. We focus on the latter case here, which limits the range of possible optimizations to the static evaluation of subexpressions, constant propagation, and dead-code elimination.

Assume that m_2 is the original model, and m_1 is a restricted version of m_2 (so $m_1 \preceq m_2$). The specialization step is considered correct if it does not affect the execution of the model—the semantics of m_2 is exactly the same as those of m_2 executed under conditions expressed in restriction specification. If restriction is based solely on dead-code elimination, the correctness condition is even simpler: the restriction is correct if it never removes any code that is live under specification conditions.

Consider a simple specialization step. Transitions fired by impossible events are removed. Read references (rvalues) of a constrained variable are substituted by the variable's value. Assignments to dead variables are removed (preserving the right-hand side or side effects). Read accesses to dead variables can be substituted by any value. Constrained function calls are replaced with their value.

The optimization step is somewhat more complicated and leaves more choices. We propose the iteration of basic optimization steps until a fixpoint is reached:

i. Statically evaluate all parts of expressions that can be evaluated (constant propagation).
ii. Remove pure expressions whose value is never used.
iii. Remove transitions whose guards are proven to be false.
iv. Remove pure state machines that do not produce any outputs and are not referred from other machines.
v. Remove the triggering of internal events that do not fire any transitions.
vi. Remove sibling-less basic states that contain no actions.
vii. Remove nonreachable components (nonreachable states and the transitions targeting them).

All the optimizations but the last one are fairly simple. The quality of reachability analysis [13, 14] depends strongly on the level of abstraction it takes. Unfortunately the exact computation of reachable state space is not feasible for arbitrarily big systems. Actually even over-approximations applied in the verification of safety properties are too expensive to apply in a code generation tool. A radical over-approximation can be achieved by the reduction of reachability analysis to the easily solvable reachability problem in a hierarchical directed graph without labels on the edges. This approximation is not so weak as it may seem: the control flow in statecharts is explicit, and simple restrictions like `impossible e` easily break it into nonreachable pieces.

6 Refinement

The inheritance hierarchy of interface restrictions is accompanied by an isomorphic hierarchy of restricted models. In this section, we shall study the meaning of the hierarchical relation between models.

Definition 1 (Trace Inclusion). *Consider two statechart models m_1 and m_2. We say that m_1 refines m_2 with respect to the alphabet α, written $m_1 \lesssim_\alpha m_2$, iff*

$$\forall t_1 \in \mathit{traces}(m_1).\ \exists t_2 \in \mathit{traces}(m_2).\ t_1 = t_2 \restriction \alpha\ \ .$$

Intuitively, m_1 refines m_2 with respect to αm_1 if it can be executed with the same trace, modulo some m_2-specific events. Let m_1 be the least CD player of Figure 5. One of its execution traces might be

$$t\ =\ \langle \mathit{StandBy,\ Play,\ startPlay(),\ Stop} \rangle$$

This trace is included in the following trace of the greatest CD player of Figure 3:

$$t'\ =\ \langle \mathit{StandBy,\ SwitchAlarm,\ SwitchAlarm,\ Play,\ startPlay(),\ Stop} \rangle$$

Since each execution trace of m_1 can be included in some trace of m_2, we can say that m_1 will emulate m_2 if run under the conditions assumed in the restriction for m_1 (so only events from αm_1 can occur). This can be formalized for all models obtained by automatic restriction:

Theorem 2 (Soundness of Restriction). *If m_2 is a statechart model of a reactive system and m_1 has been obtained from m_2 by restriction with respect to the RL specification S, then m_1 refines m_2 with respect to αm_1. More precisely*

$$m_1\ \preceq_S\ m_2\ \Rightarrow\ m_1\ \lesssim_{\alpha m_1}\ m_2.$$

Theorem 3 (Completeness). *If m_2 is a statechart model and m_1 has been obtained from m_2 by restriction with respect to the RL specification S, then*

$$\forall t_2 \in RL[\![\ S\ ]\!]_{m_2}\mathit{traces}(m_2).\ \exists t_1 \in \mathit{traces}(m_1).\ t1 = t2$$

or briefly $m_1\ \preceq_S\ m_2\ \Rightarrow\ m_2 \lesssim_{\alpha m_1} m_1$.

These results follow directly from the conservative construction of specialization transformations, based solely on dead-code elimination.

Note that if the restrictions on internal logics (nonsensors and nonactuators) were allowed, the above results would not hold. Conservative dead-code elimination would not be a viable specialization algorithm any more. One could modify the behavior of the model arbitrarily by forcing some internal variables to behave like constants, where they are not constants indeed. Restrictions over sensors and actuators avoid this: sensors are never changed inside the model, and actuators are never read inside the model.

7 Mutually Exclusive Features

The greatest member of a model family can be understood as a description of the domain, from which all family members are selected by restriction. It seems, that since model elements can only be removed, the approach does not

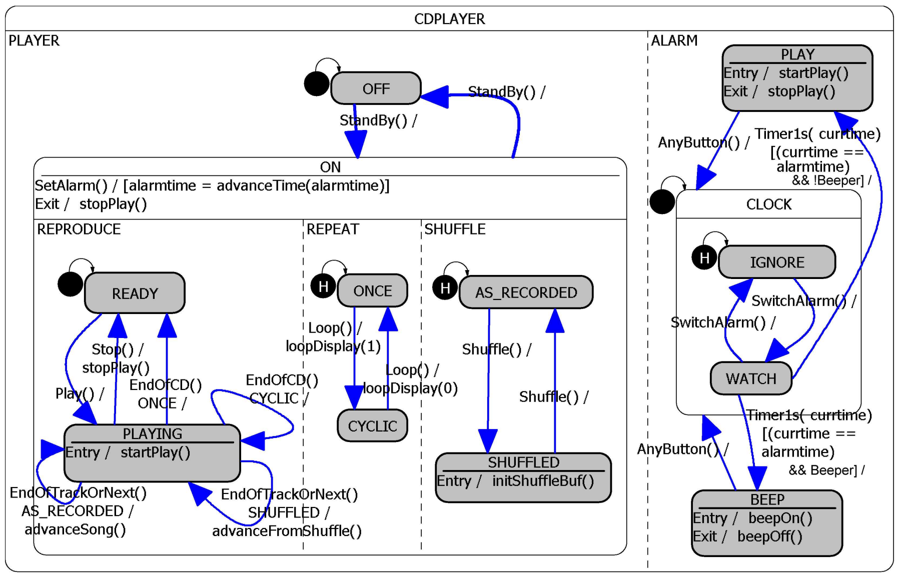


Fig. 7. A CD player model containing conflicting modes of waking up: a beeper and an automatic start-up of a CD. Note the determinizing use of a compile-time sensor *Beeper* on the two transitions leaving the *WATCH* state

support selection from sets of mutually exclusive features. We propose the use of *compile-time sensors* to deal with this problem.

Compile-time sensors are input variables that should be fixed to constants in restriction specifications. More precisely, we relax the condition that the greatest model must describe a working product. Conflicting behaviors should be determinized (guarded) by a compile-time sensor. Entire replaceable system components could be defined in this fashion by guarding their conditions with proper sensor values.

Figure 7 presents a more general CD player model. This model contains two possible reactions to an alarm timeout event. One CD player will presumably turn on the music; when the alarm time outs, the other should just start beeping.

Actual models describing physical devices can be created from generalized models by restricting enough until all compile-time sensors are specialized out. In fact, nothing prevents the introduction of several levels of restriction before the model becomes executable. We can speak of a hierarchy part that is closer to the root as of abstract models—models that describe narrower, less customizable families than the greatest model, but still cannot be instantiated into concrete running programs. Only models sufficiently deep in the restriction hierarchy can be used successfully for program synthesis. Abstract models correspond to abstract classes, while concrete deterministic models correspond to concrete classes.

However, the notion of being abstract or concrete relates to behaviors, not to the structure of entities as assumed in object-oriented programming.

As a side effect, compile-time sensors can also be propagated forward to the C preprocessor to allow the control of driver code in the traditional fashion conditional compilation.

8 Related Work

The notion of refinement between two statecharts has been studied for Unified Modeling Language (UML) models, however, mostly in the context of consistency verification rather than in the context of transformation. A recent account of such work is given by Harel and Kupferman [7], proposing a formal notion of inheritance relation between statecharts. Noteworthy they argue that establishing a refinement relation for a top-down design is a hard problem. This refinement is obtained by the construction in the bottom-up approach presented here. Engels and associates [5] discuss a problem of formal consistency between diagrams produced at various stages of a modeling process. As an example, a distinction is made between the *invocable* and *observable* behaviors. The same difference manifests itself in our semantics of RL, where constraints on inputs remove entire traces while constraints on outputs restrict the alphabet.

A consistency checker similar to that described by Engels and associates [5] could be used to extend our framework to support multiple development cycles. In such cases, restriction would be used to generate family members, while extension could be used to create the most-general model of a family of a new generation of products. We refer the reader to both of the above mentioned texts for extensive references on behavioral refinement for statecharts.

Ziadi and associates [21] describe a framework for the derivation of a product line from a single general UML model. First, a model of the entire variability domain is created. Then, a systematic way of specification and derivation of a single family member is proposed. Their approach is appealing for including consistency constraints into the model, restricting legal configurations. Unfortunately, they do not explain how the consistency is validated. Our contribution and the work of Ziadi and associates [21] are complimentary. They are concerned solely with structural modeling (class diagrams), while we only consider behavioral modeling (statecharts). In particular, [21] does not seem to consider program derivation, only model derivation for family members. It remains a question how the two problems should be solved generatively in a single framework.

The methodology presented here is an example of generative programming [4]. The ideas have been inspired strongly by program slicing [18] and partial evaluation [9, 12]. Indeed, the restriction mechanism performs well-known tasks in both techniques. Restricting by inputs (forward restriction) resembles partial evaluation, while restricting by outputs (backward restriction) is similar to the removal of program slices. While partial evaluation and slicing are applied typically to programs, we have proposed using them on the model level, which has an advantage of access to explicit control-flow information. This information is lost in the generated code, where a model is encoded typically in big integer arrays, with packed fields, interpreted in a noncompositional way. Such programs are very hard, if not infeasible, for analysis with general-purpose specializers.

In program slicing, one usually computes slices because they are of interest. Here we compute slices to remove them (the complementary slice is of interest). Partial evaluation, especially polyvariant partial evaluation [2], usually replicates pieces of code for several values of a variable to uncover the control hidden in data structures. Consequently, a partial evaluation is likely to produce bigger but faster programs. My tool only decreases the models. The discovery of control in data is not crucial in a language with an explicit control flow like statecharts.

9 Conclusion and Future work

We have presented a software development methodology for product families based on the automatic generation of restricted programs from the model of the most complex one. The methodology has been demonstrated for the language of statecharts using a combination of intuitions from program slicing and partial evaluation. Provided the restriction algorithm is sound, a behavioral refinement relation is established between members of the family. Finally, a way of handling mutually exclusive features has been discussed.

Development with restrictions is not a silver bullet. The idea of removing features to generate new members might be not very useful for big and diverse families. The development of the single greatest model may not be feasible for them. We claim, though, that restriction can significantly simplify the development process of well-defined, relatively small program families met in the production of simple home appliances, ranging from kitchen equipment through Hi-Fi systems to
mobile phones.

The class of restriction specification chosen here seems to match very well the industry needs. We have undertaken projects with two partners from the embedded systems industry in Denmark, aiming at the practical evaluation of our approach. We shall report on the experiences as soon as the case studies are over. So far, our partners are very positive about the proposed extensions to the modeling language. Nevertheless, we believe that extending the restriction language even more, giving it the behavioral power of true models of environments, will multiply the benefits of the approach. We are currently working on providing a theoretic framework for this.

10 Acknowledgments

The author would like to thank Peter Sestoft, Kåre Kristoffersen, Kasper Østerbye, Henrik Hulgaard, and Krzysztof Kaczmarski for discussions on the above material. The visualSTATE division of IAR Systems and the cooling division of

Danfoss A/S provided practical evidence on shortcomings in the present development process. Last, but not least we would like to thank all three anonymous reviewers for suggesting some important improvements.

References

1. Gérard Berry. The foundations of Esterel. In Gordon Plotkin, Colin Stirling, and Mads Tofte, editors, *Proof, Language and Interaction. Essays in Honour of Robin Milner*, Foundations of Computing Series, pages 425–454. The MIT Press, Cambridge, Massachusetts, 2000.
2. Mikhail A. Bulyonkov. Polyvariant Mixed Computation for Analyzer Programs. *Acta Informatica*, 21(5):473–484, December 1984.
3. Krzysztof Czarnecki, Thomas Bednasch, Peter Unger, and Ulrich Eisenecker. Generative programming for embedded software: An industrial experience report. In Don Batory, C. Consel, and Walid Taha, editors, *Generative Programming and Component Engineering (GPCE)*, volume 2487 of *Lecture Notes in Computer Science*, pages 156–172, Pittsburgh, PA, USA, October 2002. Springer-Verlag.
4. Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming: Methods, Tools, and Applications.* Addison-Wesley, 2000.
5. Gregor Engels, Reiko Heckel, and Jochen Malte Küster. Rule-based specification of behavioral consistency based on the UML meta-model. In Martin Gogolla and Cris Kobryn, editors, *4th International UML Conference – The Unified Modeling Language, Modeling Languages, Concepts, and Tools*, volume 2185 of *Lecture Notes in Computer Science*, pages 272–286, Toronto, Canada, October 2001. Springer-Verlag.
6. David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.
7. David Harel and Orna Kupferman. On object systems and behavioral inheritance. *IEEE Transactions on Software Engineering*, 28(9):889–903, September 2002.
8. David Harel and Amir Pnueli. On the development of reactive systems. In Krzysztof R. Apt, editor, *Logic and Model of Concurrent Systems*, volume 13 of *NATO ASI*, pages 477–498. Springer-Verlag, October 1985.
9. John Hatcliff, Torben Æ. Mogensen, and Peter Thiemann, editors. *Partial Evaluation: Practice and Theory. DIKU 1998 International Summer School*, volume 1706 of *Lecture Notes in Computer Science*. Springer-Verlag, Copenhagen, Denmark, 1999.
10. C.A.R. Hoare. *Communicating Sequential Processes.* International Series in Computer Science. Prentice Hall, 1985.
11. IAR Inc. IAR visualSTATE®. `http://www.iar.com/Products/VS/`.
12. Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation.* International Series in Computer Science. Prentice Hall, 1993. `http://www.dina.kvl.dk/~sestoft/pebook`.
13. J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. In *Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 1–33, Washington, D.C., 1990. IEEE Computer Society Press.
14. Jørn Bo Lind Nielsen. *Verification of Large/State Event Systems.* PhD thesis, Technical University of Denmark, April 2000.

15. Object Management Group. OMG Unified Modelling Language specification, 1999. `http://www.omg.org`.
16. David L. Parnas. On the design and development of program families. *IEEE Transactions on Software Engineering*, Vol. SE-2(No. 1):1–9, March 1976.
17. SCOPE: A statechart compiler, 2003. `http://www.mini.pw.edu.pl/~wasowski/scope`.
18. Frank Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3):121–189, September 1995.
19. Andrzej Wąsowski. On Efficient Program Synthesis from Statecharts. In *ACM SIGPLAN Languages, Compilers, and Tools for Embedded Systems (LCTES)*, San Diego, USA, June 2003. ACM Press.
20. Andrzej Wąsowski and Peter Sestoft. On the formal semantics of visualSTATE statecharts. Technical Report TR-2002-19, IT University of Copenhagen, September 2002.
21. Tewfik Ziadi, Jean-Marc Jézéquel, and Frédéric Fondement. Product line derivation with UML. In *Software Variability Management Workshop*, University of Groningen Departement of Mathematics and Computing Science, February 2003.