Model Projection: Simplifying Models in Response to Restricting the Environment

Kelly Androutsopoulos[1] David Binkley[2] David Clark[1] Nicolas Gold[1]
Mark Harman[1] Kevin Lano[3] Zheng Li[1]

[1] University College London, Malet Place, London, WC1E 6BT, UK.
[2]Loyola University Maryland, Baltimore, MD 21210-2699, USA.
[3]King's College London, Strand, London, WC2R 2LS, UK.

ABSTRACT

This paper introduces *Model Projection*. Finite state models such as Extended Finite State Machines are being used in an ever increasing number of software engineering activities. Model projection facilitates model development by specializing models for a specific operating environment. A projection is useful in many design-level applications including specification reuse and property verification.

The applicability of model projection rests upon three critical concerns: correctness, effectiveness, and efficiency, all of which are addressed in this paper. We introduce four related algorithms for model projection and prove each correct. We also present an empirical study of effectiveness and efficiency using ten models, including widely–studied benchmarks as well as industrial models. Results show that a typical projection includes about half of the states and a third of the transitions from the original model.

1. INTRODUCTION

Both automatically and manually generated state-based models find many applications in software engineering. For example, they are often constructed by model checkers [7] and built by engineers in the telecommunications and embedded systems sectors [8, 9, 23]. They are also used in modelling notations such as the UML [26], which draws heavily on state-based modelling [25], and are widely used for specifying discrete-event control devices, such as automated manufacturing systems (AMS) [28].

A great deal of engineering effort is directed towards the design of an organisation's models. As a result, it is common for models to be reused. One such scenario arises when control models are reused within a device that offers a restricted operational environment. For example, the 'basic model' car air conditioning system may not offer the climate control functionality found in more luxurious models. It would be wasteful to specify different models for each potential car configuration. Rather, there is typically a single controller for the entire product line [2]. In this situation, model projections, specialized to the particular environment, are useful as they avoid producing bloated controllers that implement unused functionality.

The design of models typically occurs at an earlier stage of the overall development process than coding where the cost of errors is, in general, higher. This makes it important to have powerful and flexible tools for model analysis. Model projection facilitates such analysis in much the same way that techniques such as program transformation and program slicing assist with code level analysis [27, 14, 30].

Figure 1 presents an example of model projection in which a vending machine allows a user to insert coins and vends tea with options of large cups, small cups, sugar, and double sugar. Consider redeploying this controller in an environment where the 'sugar' button is disabled, and thus in an environment in which the event *Sugar()* never occurs. The projection removes all states and transitions (shown in dashed grey) unreachable by event sequences in the restricted environment. This causes the state *Sugar* to become unreachable and subsequently the state *Double Sugar*, because it is only reachable from the state *Sugar*. When these states and their associated transitions are projected out of the model, only the black elements remain.

Model projection can also be used to facilitate property verification by reducing the complexity of a model and thus making analysis more efficient. Consider the *production cell* example used in the industrial case study of a German metal processing plant [24]. In this system, metal blanks enter the system on a feed belt and are conveyed via a table by robot arms to one of two presses. Assume that we want to prove that the feed belt motor is off if there is a blank on the belt and a blank on the elevating rotating table. Call this property P. It turns out that Property P only concerns two sensors and one actuator of the model. To prove that Property P holds in the original model requires consideration of a state space having over 3,000 states. However, only four events that affect P (s1on, s1off, s3on, s3off). A model projection produced by ignoring all other events has only four states and eight transitions. This substantial reduction in controller size means that P can be verified to hold in all states directly. As shown in Section 4, this implies that P holds in the original model.

With these applications and others, the applicability of our approach depends upon both correctness and the degree of reduction achievable in reasonable time. Therefore, in this paper we are concerned with the theoretical and empirical underpinnings of model projection as formalized in Section 2. Section 3 then introduces four progressively more precise (and more expensive) model projection algorithms. This is followed in Section 4 by proofs of correctness for the four algorithms. Finally, Section 5 investigate the effectiveness and efficiency of our approach in terms of the reduction achieved on a set of ten models (including standard benchmarks and real world models).

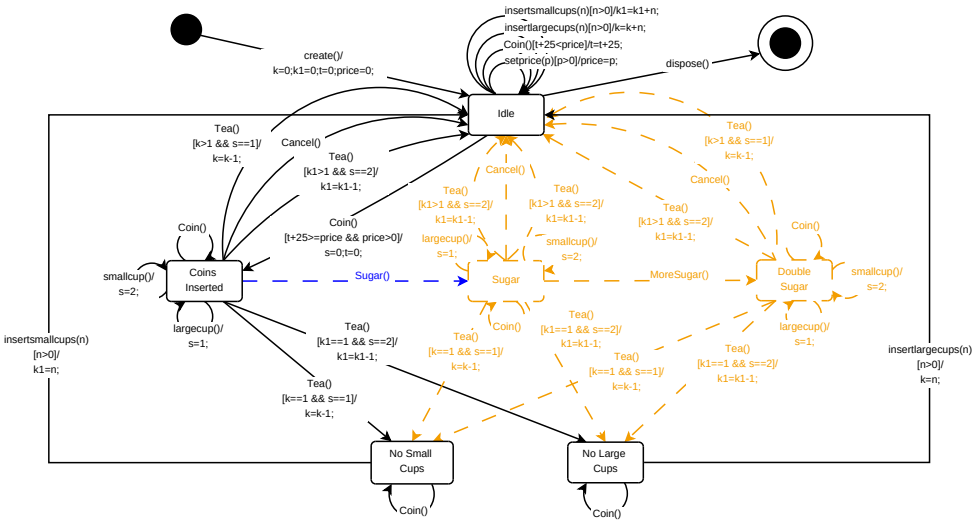

Figure 1: Illustrative example: model projection for a simple vending machine. The complete vending machine includes both the dashed grey and black elements. The projection, for reuse in an environment where sugar is unavailable (the 'sugar' button is disabled), includes only the black elements of the model. The states and transitions concerned with providing sugar are projected out of the model. The projection is faithful to all other (sugar free) interactions with its environment.

2. MODEL PROJECTION DEFINITION

This section first introduces some notation and terminology related to the particular models focused on in this paper: Extended Finite State Machines (EFSMs). It then formalizes the notion of a projection for EFSMs.

Definition 1 (Extended Finite State Machines (EFSMs))
An EFSM M is a tuple (S, T, A, S_i) where S is a set of states, T is a set of state to state transitions, A is an event alphabet (a set of events), and S_i is the set of initial states. A transition $t \in T$ has a label $lbl(t)$ of the form $e()[g]/a$, where $e \in A$, g is a boolean guard, written in an unspecified condition language, and a is an action, written in an unspecified command language. All parts of the label are optional. Only when the guard evaluates to True can the transition be taken. An omitted guard is assumed to be True. For transition t, source(t), target(t), guard(t), trigger(t), and action(t) are used to refer to t's source state, target state, boolean guard, trigger event, and action, respectively.

An EFSM M takes as input a sequence of events $E \in A^*$. Each event includes, in addition to the event name, a list of formal parameters that are passed values from the external environment with the event. A transition's action commonly updates the store and may produce output events, which need not come from A. Finally, if two transitions share the same source state, the same trigger event, and their guards can be simultaneously *True*, then M is a non-deterministic EFSM; otherwise, it is deterministic.

A completely specified EFSM includes, for every state, a transition triggered by every event in the event alphabet A. Since completeness can often obscure the true nature of an EFSM, as a convention EFSMs are often presented with certain transitions left implicit. A common interpretation for implicit transitions is a self transition. In effect the event is consumed by having the state machine stutter (repeat) the same state without any effect other than consuming the event. The following definition formalizes the event sequences that do not require stuttering.

Definition 2 (Implicit Transition Free Event Sequence)
For EFSM $M = (S, T, A, S_i)$, an input event sequence $E \in A^*$ is referred to as implicit transition free *if M takes no implicit transitions when presented with E. Such an event sequence is also referred to as* stutter-free. *For a set of event sequences X, itf(M, X) denotes the implicit transition free subset of X:*

$$itf(M, X) = \{E \in X \mid M \text{ takes no implicit transitions when processing } E\}.$$

basicProjection(EFSM $M = (S, T, A, S_i)$, Ignore Event Set $\mathcal{I}$)
Let $M' = (S', T', A, S_i)$ where
$$S' = \{s' \in S \mid \exists \text{ path } p = s \cdots s' \in T^* \ \wedge \ s \in S_i\} \\ \wedge \forall t \in p, trigger(t) \notin \mathcal{I}$$
$$T' = \{t \in T \mid source(t) \in S' \wedge target(t) \in S'\}$$
in
$$M' = (S', T', A - \mathcal{I}, S_i)$$

Figure 2: The basic model projection.

Given a set of events to ignore, denoted $\mathcal{I}$, the projection of EFSM M preserves M's behavior when placed in an environment that does not generate events from $\mathcal{I}$. The resulting set of event sequences, formalized by the following filter, is then used to define *model projection*.

Definition 3 (Event Sequence Set Filter)
Sequence-set Filter function fi$(X, \mathcal{I}) = \{E \in X \mid E \cap \mathcal{I} = \varnothing\}$ where $E \cap \mathcal{I}$ denotes the intersection of the set of events in E with $\mathcal{I}$.

Definition 4 (Model Projection)
A Model Projection *of EFSM $M = (S, T, A, S_i)$ for ignore set $\mathcal{I}$ is a reduced EFSM M' that is semantically indistinguishable from M on all event sequences from fi$(A^*, \mathcal{I})$.*

For a projection to be semantically indistinguishable as required by the definition, it must preserve the behavior of the original machine. There are two aspects to this semantic requirement: what behavior is preserved and over what set of event sequences it is preserved. The first of these, "what behavior is preserved" is formalized by Definition 9 in Section 4. Informally, the behavior includes the variable values computed at each state.

The second aspect of the semantic requirement considers the set of event sequences over which the semantics is preserved. Two options are considered: the *strong semantic requirement* and the *weak semantic requirement*. Under the strong requirement, behavior is preserved on all sequences in fi$(A^*, \mathcal{I})$, while for the weak requirement, behavior is preserved only for the stutter free sequences of fi(itf$(M, A^*), \mathcal{I})$. Thus the weak requirement does not require matching behavior on event sequences for which the EFSM implicitly consumes an event. Rather, it restricts this requirement to event sequences where the behavior of M is explicitly defined.

Figure 2 presents the definition of a basic projection of an EFSM, which is illustrated by the example in Figure 1. In Section 4 we prove that the implementation of basicProjection shown in Figure 3 satisfies the strong semantic requirement. Furthermore this algorithms removal of states and transitions can facilitate further simplifications. Thus three additional algorithms go beyond the basic projection and consider simplifying the state machine using constant propagation and state merging. The most aggressive of the state merging algorithms satisfies only the weak semantic requirement.

3. ALGORITHMS

This section describes four model projection algorithms that represent differing performance/precision tradeoffs. Model projection has several possible implementation strategies. For example, one could build a projection by adding necessary elements to an initially empty model. Alternatively, one could remove elements from the model being projected. The initial implementation follows the later approach. To do so, it recasts the notion of program slicing [32] to the EFSM domain. However, this recasting differs from previous work on slicing state-based models [15, 19, 21, 22, 31], which transplant concepts from the program slicing [32] in an attempt to incrementally adapt program-slicing concepts and techniques. These differences are considered further in Sections 5.3 and 6.

For much of its history, dependence analysis has typically been applied to programs, but there has been a recent increase in the theory and practice of dependence analysis for models, and in particular state-based models [**?**]. Indeed, it has been argued [**?**] that all descriptions, whether they be specifications, models, or other descriptions, will have a tendency to become executable over time, making such descriptions more likely to become the subject of techniques associated with program analysis, such as dependence analysis and slicing. The recent interest in dependence analysis for state-based models makes it timely to consider whether notions of slicing, hitherto defined for programs, remain appropriate when transplanted into the paradigm of models. This paper argues that the notion of slicing needs to be generalised to one of model projection for considering models both EFSMs and otherwise. Specifically, the 'slicing criterion' and algorithms for slicing will need to take account of model-specific characteristics.

The recasting of slicing used to implement model projection provides a more natural slicing approach by providing a more natural way to specify the slicing criterion: an *environment-based slice* is taken with respect to an ignore set $\mathcal{I}$. As shown in Sections 4 and 5, environment-based slicing correctly and effectively implements model projection. This achieves a long standing goal in program slicing of defining the slicing criteria in terms of higher level concepts [13, 32].

The remainder of this section first introduces the slicing algorithm basicSlice and then three extensions that incorporate constant propagation and 2 different state mergings. Figure 3 presents basicSlice. Given as a slicing criterion an ignore set $\mathcal{I}$, basicSlice computes a slice by deleting all transitions whose trigger event corresponds to events in the ignore set. Then it removes all states and their transitions that are no longer reachable from an initial state. The reduced model is a basicProjection as defined in Figure 2. Figure 1 shows the basicSlice produced for the Vending Machine example using the ignore set $\mathcal{I} = \{\mathsf{Sugar}()\}$.

The first extension to basicSlice, Algorithm A_2 is shown in Figure 4, incorporates constant propagation. Algorithm A_2 further reduces the projection's size by replacing a constant-valued variable by its value throughout the machine. When a guard is updated, it may simplify to *False*; thus the corresponding transition (and perhaps its target state) can be removed. The algorithm implements

basicSlice(EFSM $M = (S, T, A)$, Ignore Event Set $\mathcal{I}$)
{
  $M' \leftarrow M$
  delete from T' all transitions t where *trigger*$(t) \in \mathcal{I}$
  while $\exists$ state s with no incoming transitions do
    remove s and all transitions t where *source*$(t) = s$
  return M'
}

Figure 3: Algorithm A_1: basicSlice. Basic algorithm for computing an environment-based slice.

a simple flow-insensitive constant propagation algorithm in which variables are mapped to values in a flat lattice [11]. Initially variables that appear as formal parameters of an event labelling a transition are assigned the value $\perp$ (non-constant) because they receive an unknown value from the environment. All other variables v are initialized to $\top$ (uninitialized). The assignments that label the transitions are then taken into account. If the right-hand side of the assignment evaluates to a constant then the value of the left-hand side variable is replaced with the meet of the constant and the variable's present value. Otherwise, the value is replaced with $\perp$.

In addition to constant propagation, two state merging algorithms are used to further reduce the size of a projection by combining groups of states that have identical semantics. The right-equivalent

Slice$_{CP}$(EFSM M, Ignore Event Set $\mathcal{I}$)
{
  $M' =$ basicSlice$(M, \mathcal{I})$
  constantPropagation(M')
  return M'
}

constantPropagation(EFSM $M = (S, T, A, S_i)$)
{
  $\bar{\sigma} = \lambda\, v.$ { $\perp$ if v is used as a formal parameter of an input event in a transition label; $\top$ otherwise }
  foreach $t \in T$
    foreach assignment "$v = exp$" $\in$ *action*(t)
      let x = result of evaluating exp using $\bar{\sigma}$
      in **update**$(\bar{\sigma}, v, x)$
  deleteUnTriggerableTransitions$(M', \bar{\sigma})$
}

deleteUnTriggerableTransitions(EFSM $M = (S, T, A, S_i)$, abstract store $\bar{\sigma}$)
{
  while $\exists\ t \in T$ where *guard*(t) evaluates to *False* using $\bar{\sigma}$
    remove t from T
    while $\exists\ s \in S$ with no incoming transitions do
      remove s and all transitions $\bar{t}$ where *source*$(\bar{t}) = s$
}
where
  update$(\bar{\sigma}, v, x)$ replaces v's value in $\bar{\sigma}$ with $\bar{\sigma}(v) \sqcap x$
  using the following meet operation:
  $\top \sqcap \top = \top$ $\top \sqcap C = C$ $\top \sqcap \perp = \perp$
  $C \sqcap C = C$ $C \sqcap C' = \perp (C \neq C')$
  $C \sqcap \perp = \perp$ $\perp \sqcap \perp = \perp$

Figure 4: Algorithm A_2: Constant Propagation removes transitions whose guard is always *False*.

Slice$_{Merge}$(EFSM M, Ignore Event Set $\mathcal{I}$)
{
  $M' =$ basicSlice$(M, \mathcal{I})$
  constantPropagation(M')
  R-mergeEquivalentStates(M')
  if only satisfying the weak semantic requirement then
    G-mergeEquivalentStates(M')
}

R-mergeEquivalentStates(EFSM $M = (S, T, A, S_i)$)
{
  while $\exists$ States $s_1, s_2 \in S$ such that *R-equivalent*(s_1, s_2)
    replace s_1 and s_2 with s, where s has as its outgoing (and self-) transitions a copy of those of s_1,
    and incoming transitions the union of those of s_1 and s_2.
}

R-equivalent(States s_1, s_2)
{
  if *finalState*$(s_1) \wedge$ *not finalState*(s_2) or *not finalState*$(s_1) \wedge$ *finalState*(s_2)
    return false
  else return true iff
    $\forall\, t \in T \cdot$ *source*$(t) = s_1 \Rightarrow \exists\, t' : T \cdot$ *source*$(t') = s_2 \wedge$ *trigger*$(t') =$ *trigger*$(t) \wedge$ *action*$(t') =$ *action*(t)
      $\wedge$ *guard*$(t') \equiv$ *guard*$(t) \wedge$ *target*$(t') \sim$ *target*(t)
    $\forall\, t \in T \cdot$ *source*$(t) = s_2 \Rightarrow \exists\, t' : T \cdot$ *source*$(t') = s_1 \wedge$ *trigger*$(t') =$ *trigger*$(t) \wedge$ *action*$(t') =$ *action*(t)
      $\wedge$ *guard*$(t') \equiv$ *guard*$(t) \wedge$ *target*$(t') \sim$ *target*(t)
    // Where $s_1 \sim s_2$ means the targets are the same state, or one is s_1 and the other s_2.
}

G-mergeEquivalentStates(EFSM $M = (S, T, A, S_i)$)
{
  For a set of transitions T, *triggers*$[T]$ denotes the set of triggers of the transitions of T.
  Function $metric(g)$ denotes the number of states removed by merging groups in g into single states: $\Sigma_{s \in g}(|\, s\,| - 1)$
  where $|\, s\,|$ denotes the size of set s.
  possibleGroups $= \{ss \in 2^S \mid\ |ss| \geq\ 2 \wedge$
    $(\forall\, t \in T \cdot$ *source*$(t) \in ss \wedge$ *target*$(t) \in ss \Rightarrow$ *action*$(t) =$ *skip*$)$
    $\wedge$ ((*triggers*$[\{t \in T \mid$ *source*$(t) \in ss \wedge$ *target*$(t) \in ss\}]$
      $\cap$ *triggers*$[\{t \in T \mid$ *source*$(t) \in ss \wedge$ *target*$(t) \notin ss\}] = \varnothing)\}$
  if *possibleGroups* $\neq \varnothing$ then
    mergeGroups $= \{pg \in 2^{possibleGroups} \mid pg \neq \varnothing \wedge \forall$ *state sets* $ss_1, ss_2 \in pg \cdot ss_1 \neq ss_2 \Rightarrow ss_1 \cap ss_2 = \varnothing\}$
    bestGroup $= g \in$ *mergeGroups* such that $metric(g)$ is maximal.
    collapseStates$(M,$ *bestGroup*$)$
}

Figure 5: Algorithms A_3 and A_4: Two State Merging algorithms

merging algorithm extends an algorithm of Ilie et al. [17] to EFSMs. R-equivalence equates to bisimulation for deterministic state machines. It has a counterpart that merges left-equivalent states. However, this merge is only relevant for non-deterministic EFSMs and is not presently implemented. Merging both R-equivalent and L-equivalent states forms a bisimulation for general EFSMs. The approach, implemented by the function R-mergeEquivalentStates, shown in Figure 5, repeatedly merges pairs of states provided that the two states of the pair are *R-equivalent*, until no further merging is possible. States s_1 and s_2 are *R-equivalent* if, for every transition from s_1, there is a corresponding transition from s_2 and vice versa. That is, for each trigger event, the outgoing transitions from the two states are identical in their guards and actions, and have equivalent target states after the merge (i.e., s_1 and s_2 are considered to be the same state). This transformation preserves the regular expression language (i.e., the set of stutter-free event sequences) accepted by the state machine.

The second merging algorithm, G-mergeEquivalentStates, shown in Figure 5, is a more aggressive merge that can yield greater simplification and, as a tradeoff, only preserves the weak semantic requirement. In this function the set *possibleGroups* includes all groups (subsets of S) that are eligible to be merged into single states. A group ss can be merged if it has size two or more, all transitions within the group have no actions, and the set of *internal triggering events* is disjoint from the set of *exiting triggering events*. The set of internal triggering events includes those events that trigger transitions from a state in ss to another state in ss while the set of exiting triggering events includes those events that trigger transitions from a state in ss to a state outside of ss. While unimplemented, it is possible to relax this requirement and allow the merge, provided that for all events E, the guard conditions of each internal transition triggered by E are logically inconsistent with the guard conditions of all of the exiting transitions triggered by E.

If the set *possibleGroups* is non-empty, then some merging is possible. To determine the best merge, a set of all possibilities *mergeGroups* is defined. This set includes subsets of *possibleGroups* that are pairwise disjoint and can thus be merged simultaneously. Finally, the subset that maximizes the number of states removed is chosen.

Because this algorithm is exponential in the number of states (which is likely to be the case for any algorithm which considers all possible candidates for state merging) future work includes considering heuristics for applying G-mergeEquivalentStates. However the empirical study reported in Section 5.3 indicates that, in practice, the run times are reasonable.

Figure 6 illustrates the impact of constant propagation and state merging. EFSM A is sliced with respect to ignore set $\mathcal{I} = \{a\}$. Slicing removes the two assignments to x when it deletes the transitions from $s_3 \rightarrow s_4$ and $s_5 \rightarrow s_4$. This leaves x with the constant value 1. Constant propagation rewrites the guard $x < 2$ on the transition from s_4 to the final state to "*True*", which in turn enables R-merging of states s_4 and s_5 leading to EFSM B. EFSM C shows the result of R-merging states s_1 and s_3. This merge does not require constant propagation; thus the example illustrates both that some merging is possible without performing constant propagation and that constant propagation can enable (additional) merging. If only stutter-free sequences are acceptable then G-mergeEquivalentStates can be used. This merges s_1, s_3 and s_4, s_5 yielding EFSM D.

4. ALGORITHM CORRECTNESS

In what follows we show that basic slicing Algorithm A_1 is correct with respect to the strong semantic requirement. Moreover, we show that the constant propagation and right-equivalent merging algorithms, A_2 and A_3, are also correct with respect to this requirement. Finally, the group merging algorithm is shown to be correct with respect to the weak semantic requirement. Before we present the proofs of correctness, we describe the semantics of EFSMs formally.

Definition 5 (Configurations)
Let Σ be the set of all possible stores for an EFSM. For EFSM $M = (S, T, A, S_i)$, a configuration is a pair (s, σ) where $s \in S$ is a state and $\sigma \in \Sigma$ is a store.

Definition 6 (Status)
For EFSM $M = (S, T, A, S_i)$, a status is a pair (C, E) where C is a configuration and $E \in A^*$ is an event sequence. A status is an initial status if the state contained its configuration is an initial state.

Definition 7 (Enabled Transitions)
For Status $St = (C, E)$ of an EFSM $M = (S, T, A, S_i)$ where $C = (s, \sigma)$, transition t is enabled iff source$(t) = s$, guard(t) *evaluates to True when evaluated using σ, and* trigger$(t) = head(E)$. *The set Enabled(St) denotes the set of all transitions enabled in Status St.*

The semantics of an EFSM M, is formalized as a Labeled Transition System (LTS). States of the LTS are the statuses of M and transitions of the LTS have the same labels as the transitions of M but extended with an additional label for implicit transitions (see the operational semantics definition below) . The LTS has a labeled step rule which defines legal transitions. A closed system assumption is made in which the status is only changed explicitly by the step rule. The semantics of EFSM M taking enabled transition t is given as an LTS move from Status St_1 to Status St_2, denoted $M \vdash St_1 \xrightarrow[LTS]{t} St_2$. In doing so the store is updated using the store update function denoted by $[\![action(t)]\!]$. This single step rule is formalized as follows:

Definition 8 (LTS Step $\xrightarrow[LTS]{}$)
For Status $St = ((s, \sigma), E)$ of EFSM M, taking enabled transition t produces the next status:

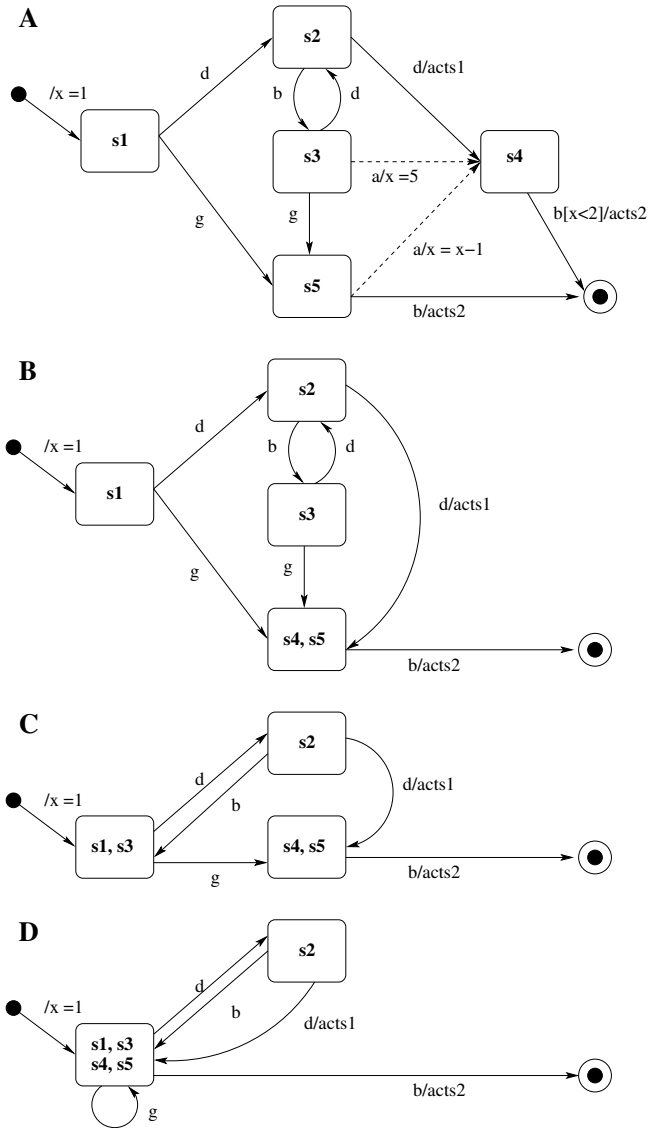

Figure 6: This example illustrates the roll of constant propagation and state merging in the current implementation of model projection. Applying basicSlice to EFSM A using ignore set $\mathcal{I} = \{a\}$ removes the two dashed transitions from EFSM A. EFSM B shows the results of constant propagation, which replaces the guard $x < 2$ with "*True*," and then the R-merging of states s_4 and s_5, which is enabled by the constant propagation. EFSM C shows the result of the additional R-merging of states s_1 and s_3. Finally, EFSM D shows the result of applying G-merge, combining s_1, s_3 and s_4, s_5.

$$\frac{t \in \mathit{Enabled}(St) \ \wedge \ \sigma' = [\![\text{action}(t)]\!]\sigma}{M \vdash ((\mathit{source}(t), \sigma), E) \xrightarrow[LTS]{t} ((\mathit{target}(t), \sigma'), tail(E))}$$

Definition 9 (Operational Semantics)
The operational semantics defines the meaning of EFSM $M = (S, T, A, S_i)$ as the LTS $(ST, ST_i, L, \xrightarrow[LTS]{})$ where ST and ST_i are sets of statuses and initial statuses, respectively, the label set L is $T \cup \{\varepsilon\}$, where the special label ε is used to model implicit transitions, and $\xrightarrow[LTS]{}$ is the LTS step rule from Definition 8.

Definition 10 (Stuttering Rule)
For a Status $St = (C, E)$, if no transitions are enabled, then the head of the event sequence is consumed without changing C.

$$\frac{\textit{Enabled(St)} = \varnothing}{M \vdash (C, E) \xrightarrow[LTS]{\varepsilon} (C, tail(E))}$$

Definition 11 (Reachable Statuses Set)
For EFSM $M = (S, T, A, S_i)$ having LTS $(St, St_i, L, \xrightarrow[LTS]{})$ and a restricted event alphabet A_R, the set $Statuses(M, A_R)$ is the set of statuses reachable in zero or more steps from an initial status in an environment that produces only events found in A_R:

$$Statuses(M, A_R) \triangleq \{\, st \in St \mid st_i \in St_i \wedge M \vdash st_i \xrightarrow[LTS]{t}{}^{*} st \text{ where } \textit{trigger}(l) \in A_R\}$$

Definition 12 (Status Congruence $\cong$)
Statuses $((s_1, \sigma_1), E_1)$ and $((s_2, \sigma_2), E_2)$ are congruent, written $((s_1, \sigma_1), E_1) \cong ((s_2, \sigma_2), E_2)$ iff $\sigma_1 = \sigma_2$ and $E_1 = E_2$.

In the following proofs, M' continues to denote a slice of EFSM M. Subscripts are added to denote a slice produced by a particular slicing algorithm; thus, a slice of M produced by A_1 (Figure 3) is denote M'_1. The main theorem uses *simulation* to establish the correctness of Algorithm A_1 by showing that for all event sequences $E \in \mathit{fi}(A^*, \mathcal{I})$ M and its slice M'_1 produce corresponding statuses. That is, whatever M can do using only the restricted event set M' also can do. This result is then extended to show that the simulation still holds for A_2 slices. Finally, it is shown that A_3 produces slices who's simulation is homomorphic to those for slices produced by A_2 and, for event sequences in $\mathit{fi}(\mathit{itf}(M, A^*), \mathcal{I})$, this results extends to slices produced by A_4.

Simulation is a relation between statuses in the operational semantics of a pair of EFSMs. We write $St_1 \rhd St_2$ to mean St_2 simulates St_1. We mildly abuse the word simulation when saying M' simulates M for a given set of event sequences $\mathcal{E}$. The intuition is that if M' simulates M for $\mathcal{E}$ then the two machines have the same behavior when placed in an environment that can only produces event sequences in $\mathcal{E}$. First we give the general definition of a simulation between the semantics of EFSMs and then the definition for the meaning of M' simulates M for $\mathcal{E}$.

Definition 13 (Simulation relation $\rhd$ for the operational semantics of EFSMs)
Let EFSMs $M = (S, T, A, S_i)$ and (its slice) $M' = (S', T', A', S'_i)$ have statuses (C_1, E_1) and (C'_1, E_1) respectively.
Then $(C_1, E_1) \rhd (C'_1, E_1)$ iff whenever

$(C_1, E_1) \cong (C'_1, E_1)$ *and* $M \vdash (C_1, E_1) \xrightarrow[LTS]{t} (C_2, E_2)$

there exists a status (C'_2, E_2) such that

$M' \vdash (C'_1, E_1) \xrightarrow[LTS]{t} (C'_2, E_2)$ *and* $(C_2, E_2) \rhd (C'_2, E_2)$

where E_2 denotes $tail(E_1)$.

Definition 14 (EFSM simulation)
EFSM $M' = (S', T', A', S'_i)$ **simulates** *EFSM $M = (S, T, A, S_i)$ for $\mathcal{E}$, written $M \rhd_{\mathcal{E}} M'$, if*

- *there is a simulation relation, $\rhd$, between the two semantics whose range includes all statuses in the semantics of M' for event sequences in $\mathcal{E}$ and*
- *for every start state $s'_i \in S'_i$ for M' there is a corresponding start state $s_i \in S_i$ for M so that any event sequence E in $\mathcal{E}$ and store σ, $((s_i, \sigma), E) \rhd ((s'_i, \sigma), E)$*

The first theorem states that projecting the behavior of an EFSM via an ignore set is safe as the resulting EFSM simulates the original with respect to the reduced environment.

Theorem 1 *Given EFSM $M = (S, T, A, S_i)$ and its slice $M'_1 = (S', T', A', S_i)$ produced by algorithm A_1 using ignore set $\mathcal{I}$ $M \rhd_{\mathcal{E}} M'_1$ for $\mathcal{E} = (\mathcal{A} - \mathcal{I})^*$ (M'_1 simulates M for $\mathcal{E} = (\mathcal{A} - \mathcal{I})^*$).*

Proof. We first prove that a non-empty simulation relation exists between the semantics of M'_1 and the semantics of M. If M'_1 is produced from M by Algorithm A_1 using ignore set $\mathcal{I}$, then M'_1 is a sub machine of M (its states, transitions and actions are contained in the states transitions and actions of M). Without loss of generality we can assume that the set of possible stores for each machine is the same. Then each status of M'_1 is a status of M. If M has status $((s, \sigma), E)$ and M'_1 has status $((s', \sigma'), E)$ where $s = s'$, $\sigma = \sigma'$, and $E \in \mathcal{E} = (\mathcal{A} - \mathcal{I})^*$ then we can show that they satisfy a simulation relation.

Because the statuses are identical $((s, \sigma), E) \cong ((s', \sigma'), E)$. Now assume that $M \vdash ((s, \sigma), E) \xrightarrow[LTS]{t} ((s_1, \sigma_1), tail(E))$ where $head(E) \in A'$. We have that $M'_1 \vdash ((s', \sigma'), E) = ((s, \sigma), E) \xrightarrow[LTS]{t} ((s_1, \sigma_1), tail(E))$ as well. If that were not the case, since $\xrightarrow[LTS]{t}$ corresponds to a transition $\xrightarrow{t}$ in M and $\xrightarrow{t}$ is triggered by an event in A', the failure to be in M'_1 can only be because the transition is deleted during construction of M'_1 by A_1 because it is unreachable. However since its source state is in M'_1 (by arbitrary choice of the status for M'_1), it must be reachable. Thus the resulting pair of configurations are identical and in turn must satisfy the simulation relation. We take $\rhd$ as the largest simulation relation between the configurations of the two semantics that both includes the identical pairs from the semantics of M and M'_1 and satisfy simulation as defined in Definition 13. Finally, by construction $\rhd$ includes a pair of identical statuses for every initial status in M'_1 because A_1 includes all of M's initial states in M'_1. □

We next show that there is an EFSM simulation between the original machine and the slice produced by A_2, which extends A_1 by applying constant propagation.

Proposition 1 *Given EFSM $M = (S, T, A, S_i)$ and its slice $M'_2 = (S'_2, T'_2, A'_2, S_i)$ produced by A_2 using ignore set $\mathcal{I}$, $M \rhd_{\mathcal{E}} M'_2$ where $\mathcal{E} = (\mathcal{A} - \mathcal{I})^*$.*

Proof. Let $M'_1 = (S'_1, T'_1, A'_1, S^i_1)$ be the result of applying A_1 to M using ignore set $\mathcal{I}$. We show that M'_2 simulates M'_1 for $\mathcal{E}$, that is there is a simulation $\rhd^2$ such that $M'_1 \rhd^2_{\mathcal{E}} M'_2$ and then rely on closure of simulations under composition for the existence of a simulation of M by M'_2 for $\mathcal{E}$, i.e. $M(\rhd;\ \rhd^2)_{\mathcal{E}} M'_2$.

First observe that A_2 produces M'_1 and then performs constant propagation. The effect of constant propagation on M'_1 is that some transitions whose guards are constantly false in the semantics of M'_1 are missing in M'_2 (*some* rather than *all* because the correctness of the constant propagation algorithm implies it is conservative). Furthermore transitions in M'_1 only reachable via transitions whose guards are constantly false are missing M'_2. So M'_2 is a sub machine of M'_1.

We show that there is a nonempty simulation of the semantics of M'_1 by the semantics of M'_2. Let $St_2 = ((s, \sigma), E)$ be a status of M'_2. Since M'_2 is a sub machine of M'_1 St_2 is also a status of M'_1 and the identical pair satisfy $\cong$. Now assume that $M'_1 \vdash ((s, \sigma), E) \xrightarrow[LTS]{t} ((s_1, \sigma_1), tail(E))$. $M'_2 \vdash ((s, \sigma), E) \xrightarrow[LTS]{t} ((s_1, \sigma_1), tail(E))$ will hold unless the transition $t \in T'_1$ does not

exist in T_2'. This could be for one of two reasons: either the guard of t is constantly false or t is only reachable via transitions whose guards are constantly false. In the first case $M_1' \vdash ((s,\sigma),E) \xrightarrow[LTS]{t} ((s_1,\sigma_1),tail(E))$ is not possible, in the second St is not a reachable configuration of M_1', so neither is it a reachable configuration of M_2'. Therefore $M_2' \vdash ((s,\sigma),E) \xrightarrow[LTS]{t} ((s_1,\sigma_1),tail(E))$ holds and the resulting configurations are identical, satisfying the simulation relation. Again $\rhd_2$ is defined similarly to $\rhd$ and the argument for the initial configurations of M_2' satisfying the relation is similar. □

We now show that the simulation produced by Algorithm A_3 is homomorphic to that produced by Algorithm A_2.

Proposition 2 *Given EFSM $M = (S,T,A,S_i)$ and its slice $M_3' = (S_3',T_3',A_3',S_i)$ produced by A_3 using ignore set $\mathcal{I}$, $M \rhd_{\mathcal{E}} M_3'$ where $\mathcal{E} = (\mathcal{A}-\mathcal{I})^*$.*

Proof. Let M_2' be the slice of M produced by A_2 using $\mathcal{I}$. We show that there exists a (model) homomorphism, h, from M_2' to M_3'. We then argue that h naturally extends to a homomorphism between the simulations of M_2' and M_3'; thus, $M \rhd_{\mathcal{E}} M_3'$ follows from $M \rhd_{\mathcal{E}} M_2'$.

Assume that M_2' includes states s_1, s_2 where s_1 and s_2 have identical outgoing transitions for events in $A-\mathcal{I}$. In more detail, for each transition t_1 where *source*$(t_1) = s_1$ and $lbl(t_1) = e[G]/acts$ and *target*$(t_1) = x$, there is a transition t_2 where *source*$(t_2) = s_2$, $lbl(t_2) = e[G]/acts$, and *target*$(t_2) = x'$ where $x = x'$ or $\{x,x'\} = \{s_1,s_2\}$. Model M_3' is identical to M_2' except that s_1 and s_2 are replaced by a single state s, each transition t targeting s_1 or s_2 is replaced be a transition targeting s, Matched transitions t_1 and t_2 in M_2' as above are replaced in M_3' by a single transition t with *source*$(t) = s$, $lbl(t) = e[G]/acts$ and *target*$(t) = z$ where $z = x$ if $x \neq s_1$ and $x \neq s_2$, or $z = s$ otherwise. Therefore, M_3' is clearly homomorphic to M_2'.

This model homomorphism induces a simulation homomorphism that maps status for M_2' to the statuses for M_3'. In particular, the statuses for s_1 and s_2 are mapped those for s. Since the associated transitions t_1 and t_2 have the same label, the resulting configurations are not affected. Consequently, $M \rhd_{\mathcal{E}} M_2'$ implies $M \rhd_{\mathcal{E}} M_3'$. □

Finally we show that applying A_4 also induces a simulation between the semantics of the original machine and the resulting slice, but only with some further restriction on the type of input event sequences that can be considered in the semantics.

Proposition 3 *Given EFSM $M = (S,T,A,S_i)$ and its slice $M_4' = (S_4',T_4',A_4',S_i)$ produced by A_4 using ignore set $\mathcal{I}$, $M \rhd^4_{\mathcal{E}'} M_4'$ where $\mathcal{E}' = fi(itf(\mathcal{M},\mathcal{A}^*),\mathcal{I})$, the set of restricted sequences which are stutter free on M.*

Proof sketch. Let M_3' be the slice of EFSM M produced by applying algorithm A_3. Let h denote the model homomorphism the exists between M_3' to M_4'. Proposition 2 shows that there is a simulation $\rhd^3_{\mathcal{E}'}$ such that $M \rhd^3_{\mathcal{E}'} M_3'$. (The proposition actually proves the stronger result for $\mathcal{E} = fi(\mathcal{A}^*,\mathcal{I}) = (\mathcal{A}-\mathcal{I})^*$ which is a superset of $\mathcal{E}'$.) The remainder of the proof shows that using h this relationship extends to M_4' (i.e., that $M\ (h(\rhd^3))_{\mathcal{E}'}\ M_4'$).

Model homomorphism h induces a homomorphism between the LTSs that are the semantics of M_3' and M_4' respectively but only if the input event sequences for M_3' are stutter free. Consider a status $((s,\sigma),E)$ in the semantics of M_3'. Its image in the semantics of M_4' under the induced homomorphism (also called h) is $((h(s),\sigma),E)$. Now suppose $M_3' \vdash ((s,\sigma),E) \xrightarrow[LTS]{t} ((s',\sigma'),E')$ where $E' = tail(E)$. Since E is stutter free for M_3' the label on t has $trigger(t) = head(E)$. We have two cases to consider. In the first case, $head(E)$ is *internal* and M_4' takes a self-loop: $M_4' \vdash ((h(s),\sigma),E) \xrightarrow[LTS]{h(t)} ((h(s'),\sigma'),E')$ and $h(s) = h(s')$ and $\sigma' = \sigma$ (because internal labels have no actions). Alternatively in the second case $head(E)$ is *exiting* and $M_4' \vdash ((h(s),\sigma),E) \xrightarrow[LTS]{h(t)} ((s',\sigma'),E')$ because G-mergeEquivalentStates preserves exiting transitions. Note that if E is not stutter free on M_3' this breaks down. It is possible that M_3' stutters: $M_3' \vdash ((s,\sigma),E) \xrightarrow[LTS]{t} ((s,\sigma),E')$ whereas because of the merging of states there exist an exiting transition from $h(s)$ in M_4' that is triggered by $head(E)$.

Thus, in conclusion, homomorphism h induces a homomorphism on simulations such that for event sequences in $fi(itf(M,A^*),\mathcal{I})$, $M_3'\ (h(\rhd^3))_{\mathcal{E}'}\ M_4')$. Combined with the result from Proposition 2 that $M \rhd^3 M_3'$, We conclude that $M\ (h(\rhd^3))_{\mathcal{E}'}\ M_4'$). □

5. EMPIRICAL EVALUATION

For environment-based slicing to be of practical use, it is important to empirically consider the reduction obtained. This section presents results from implementations of the four model slicing algorithms A_1-A_4. The empirical study addresses the following three research questions:

- RQ_1 What is the impact on model size of slicing with respect to small ignore sets?
- RQ_2 What is the impact on model size of slicing with respect to all large ignore sets?
- RQ_3 What is the performance of the slicer?

The ten EFSM models that were used in the experiment are shown in Table 1 where each model's size is given in terms of the number of states ($\#S$) and transitions ($\#T$) and the number of events in its input alphabet ($\#Event$). The first six models were used by Korel et al. [18, 19] in studies of dependence-based slicing. The last four are the production systems INRES [5] and DoorControl [29] from previous model-based studies, and TCP [33] and TCSbin. TCSbin, provided by Motorola, was originally written in SDL [4]. Since SDL specifications are richer than those currently handled by our tool, this model was transformed to meaning preserving equivalent (e.g., by removing the need for 'history' and 'All' state types).

Table 1: Experimental Models.

EFSM Model	*#S*	*#T*	*#Event*
ATM [19]	9	23	12
Cashier [20]	12	21	16
CruiseControl [18]	5	17	10
FuelPump [18]	13	25	16
PrintToken [20]	11	89	11
VendingMachine [20]	8	37	13
DoorController [29]	6	12	11
INRES protocol [5]	8	18	8
TCP [33]	12	57	6
TCSbin[Motorola]	24	65	39
Total	108	364	142

5.1 Small Ignore Sets

The first study exhaustively considers the impact of slicing using small ignore sets. Smaller ignore sets should produce larger slices, and thus provides a 'worst case' for the reductions to be expected from environment-based slicing. In addition, slice size is a proxy for the impact of an event: a larger slice size indicates a small impact while a small slice size indicates an event having a large impact. Thus, computing all possible slices can identify interesting events.

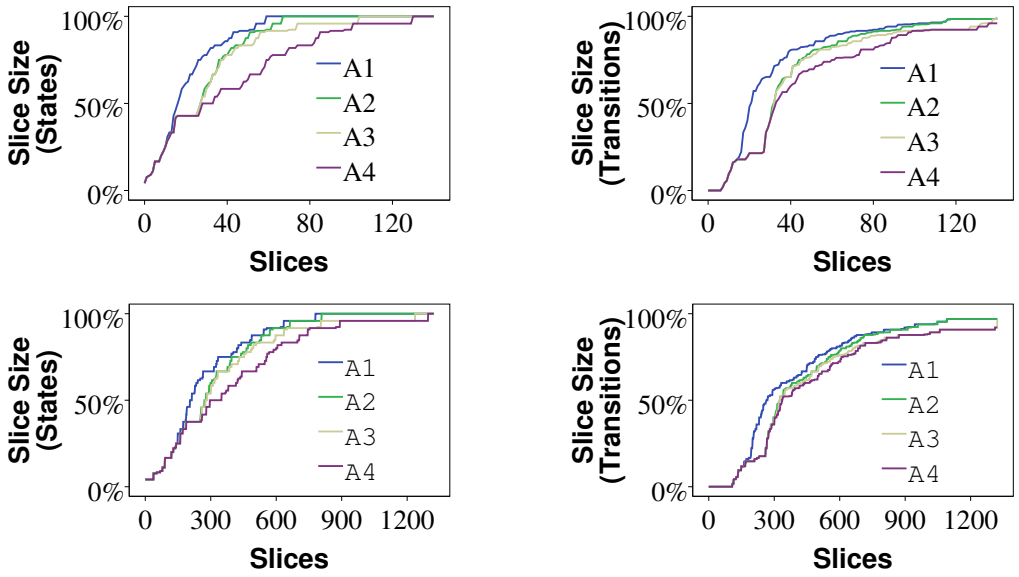

Figure 7: All slice sizes for Algorithms A_1, A_2, A_3 and A_4. The upper two figures show all slice sizes when the ignore set $\mathcal{I}$ is set to each of the 142 events and the lower two graphs show all slice sizes when the ignore set $\mathcal{I}$ is set to each of the 1323 pairs of events.

The experiment used all ignore sets of size 1, 2, and 4. Table 2 presents the average slice size computed over slices from all ten subjects separately for each of the four algorithms. Overall, the reduction in the number of states and the number of transitions tell a similar story. In addition as expected, the more expensive algorithms provide consistently smaller slices. Considering that at $n = 4$ only a third of the events are, on average, being ignored, the reductions are considerable: A_4 reduces the number of states by 60% and the number of transitions by 70%. For expensive applications, such as model checking, this reduction is especially significant.

Table 2: Average slice size when ignoring n events.

	States			Transitions		
	n=1	n=2	n=4	n=1	n=2	n=4
A_1	85.3%	78.8%	49.3%	77.5%	70.3%	33.5%
A_2	80.5%	77.2%	47.1%	71.3%	67.1%	31.1%
A_3	78.6%	74.6%	45.2%	69.2%	64.1%	30.3%
A_4	71.2%	70.7%	41.1%	66.1%	63.2%	29.8%

Figure 7 shows graphs of all slice sizes for all ignore sets of size $n = 1$ and $n = 2$ using all four algorithms. In each graph, the x-axis shows the slices in monotonically increasing order of slice size. Normalized to a percentage, the y-axis shows slice size as a percentage of number of states or the number of transitions. The data is individually sorted, and thus should not be used to compare a particular slice. In all graphs, the four lines show similar trends. Considering the extremes, for Algorithm A_1 and singleton event ignore-sets, there are no states removed in approx 80 of the 142 slices (indicated by the straight line on the top right of the diagram). For these slices, the reductions in the number of transitions is small (less than 10%). At the other end of the spectrum, in seven cases, the slices have zero transitions. This occurs when a *key* event is used as the slicing criterion. Further investigation reveals that these events often occur on the transitions close to an initial state where the triggering event determines if further processing by the model is needed. For example, ignoring the Card() event in the ATM model corresponds to not inserting an ATM card, and thus the ATM remains in its idle state.

5.2 Large Ignore Sets

The second research question considers larger ignore sets. For A_1, using ignore sets of size $n = 8$, a larger size than considered in Table 2, the average slice includes 27.6% of the states and 10.4% of the transitions. The sizes for A_2, A_3, and A_4 follow the same pattern as seen in Table 2. At the extreme, ignoring all but one event produces an average slice size of 12.7% states and 1.1% transitions. Figure 8 show graphs of these slices for all ten models where only a single line is visible for all four algorithms because they all produce very similar sized slices.

One motivation for this research question was to support a comparison with dependence-based EFSM slicing techniques and with the slicing of programs. Dependence-based techniques extract the part of an EFSM that affects one or more elements (states, transitions, or variables used in transition labels) from the EFSM. In the following comparison these elements are assumed to be transitions. Previous studies of such slices using the same ten subjects report an average slice size of between 38.4% and 68.0% of the transitions using a variety of definitions of "dependence" [1]. From a program P, a common variant of program slicing extracts a semantically meaning 'subprogram' that captures the execution of a particular statement from P.

It is difficult to make a direct comparison of environment-based slicing with the slicing of programs or dependence-based techniques for EFSMs because the slicing criteria are so different: environment-based slicing starts with one or more events to ignore, while the others start with a particular transition or statement to preserve. The average program slice size is about one third of a program [3]. This is comparable to the reduction in transitions for a model using an ignore set of size 4.

Comparing with EFSM dependence-based techniques, one possible comparison clearly favors environment-based slicing: given that dependence-based approaches preserve the behavior of a single transition, one might compare them with an ignore set that includes all but one event. Considering *all but one* event is similar to dependence-based slicing's seeking to preserve a *single* transition from the model. In this case A_4's average slice size of 1.0% of the transitions compares quite favorably. A similar result might

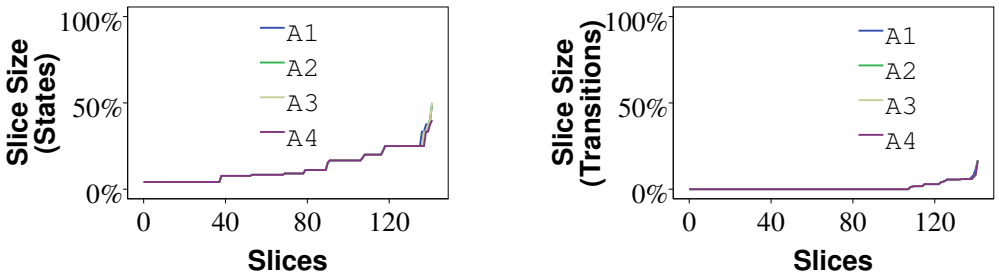

Figure 8: All slice sizes for Algorithms A_1, A_2, A_3 and A_4 using all possible ignore sets containing all but one $(n-1)$ events.

be expected from comparing the use of ignore sets of size one with dependence-based slices taken with respect to all transitions not using an ignored event. As this slicing criterion (starting point) is a substantial portion of the model, this comparison too favors environment-based slicing. Finally, consider a like-for-like comparison using a single transition for dependence-based slicing and an ignore set of size one for environment-based slice. Here A_4's average slice size of 66.1% of the transitions, while toward the large end of the range for dependence-based slicing (28.4% – 68%), still provides a comparable reduction.

5.3 Runtime Efficiency

The final research question considers the runtime efficiency of the slicer. Slices were computed on a 2.4GHz Intel(R)Core Duo CPU having 4GB memory. For ignore sets of size one, the results show that A_1 and A_2 are extremely fast with maximal times of $0.016s$ and $0.042s$, respectively. In contrast, the execution time for A_3 and A_4 reflect the PSPACE complexity of minimizing an EFSM (and in general an NFA) [16, 12]). The average times were $27s$ and $36s$ and the maximal times were $94s$ and $753s$, respectively. All times drop with larger ignore sets. For example, the worst-case times for the four algorithms using all ignore sets of size 4 are $0.0012s$, $0.025s$, $29s$, and $126s$, respectively. While all four show a large percentage drop, the impact is more meaningful for A_3 and A_4, where the worst case run time for A_3 and A_4 drop by 70% and 83% respectively.

6. RELATED WORK

Our model projections are related to model dependence analysis and slicing. However, previous work on slicing state-based formalisms [15, 19, 21, 22, 31] has concentrated exclusively on a white-box view of a state machine, in which the machine is much like a special case of a program; its states loosely play the role of statements and its transitions merely mimic control flow. This has allowed concepts from program slicing to be adapted for state-based models. Though valuable, such approaches miss the ability to talk about sub-models extracted from an original model in terms of the environment in which the model operates. In this section we describe the relationship between model projection and previous work on dependence analysis and slicing of state-based models.

Korel et al. [19] describe the increased complexity of the proof obligation for model-based slicing, arising from the presence of environmental considerations, while Sanchez et al. [28] advocate a clear separation between equipment control activities and product manufacturing procedures. However, neither provide algorithms that yield model projections defined for restricted environments.

Cheung and Kramer [6] consider a form of reachability analysis in terms of events. Such an analysis is a pre-requisite of slicing, which we use in our model projection. However, Cheung and Kramer use their approach to simplify interfaces, not to simplify the models themselves. The two complement each other as model projection computes the implications of the simplified interfaces on the models themselves.

The previous work that most closely resembles model projection is that described by Eshuis and Wieringa [10]. Their approach was developed to handle the state-space explosion problem when model checking workflows are modelled as UML activity diagrams. Rather than being based on dependence analysis, their approach consists of four reduction rules based on activity hypergraphs. Each of their four rules is related to both model-based slicing and aspects of model projection. The first of the four rules reflects changes made to the machine during the overall rewrite process back to its environment. This is similar to our notion of implicit-transition-free event sequences (Definition 2). The second rule insists that no two named events can occur at the same time (i.e., they can only be interleaved). This is similar to our requirement that only one event is dealt with at a time. The third rule defines when a variable can be removed, and is similar to a data dependence analysis that is a prerequisite of dependence-based slicing. The fourth rule, which resembles a restricted form of slicing, defines how `wait` and `activity` states can be removed from UML activity hypergraphs In comparison, with our approach, the environmental restriction is specified by the designer, whereas in the approach of Eshuis and Wieringa, the iterative application of the four rules can have the effect of removing some environmental elements.

Model projection introduces the possibility of slicing on the events in which a model participates to produce a reduced model specialized to a specific environment. Model projection therefore presents new challenges and opportunities for model-based slicing, in which the environment essentially becomes a first class citizen in the slicing criteria.

7. CONCLUSION

We introduced model projection in order to extend to the design level benefits that slicing offers at the coding level. However, modeling languages are not simply higher-level programming languages and so we found that we needed to re-think the way in which the simplification inherent in slicing is defined and implemented. In particular, we focused on the model's environment as a criterion for simplification, rather than the internal white box structure of the model itself.

As with any new approach there is an obligation to demonstrate correctness, effectiveness, and efficiency and these have therefore formed the primary contributions of this paper. We have introduced and proved correct a set of four algorithms for model projection that trade effectiveness (in terms of projected model size) for efficiency (in terms of projection construction time). The implementations of our four algorithms are available from the SLIM project website at `http://slim.dcs.kcl.ac.uk`.

8. REFERENCES

[1] K. Androutsopoulos, N. Gold, M. Harman, Z. Li, and L. Tratt. A theoretical and empirical study of EFSM dependence. In

International Conference on Software Maintenance (ICSM), pages 287–296, Edmonton, Canada, 2009.

[2] A. Baresel, H. Pohlheim, and S. Sadeghipour. Structural and functional sequence test of dynamic and state-based software with evolutionary algorithms. In *Genetic and Evolutionary Computation Conference (GECCO '03)*, pages 2428–2441, Berlin, 2003. Springer-Verlag.

[3] D. Binkley, N. Gold, and M. Harman. An empirical study of static program slice size. *ACM Transactions on Software Engineering and Methodology*, 16(2), 2007.

[4] C. Bourhfir, R. Dssouli, E. Aboulhamid, and N. Rico. Specification and description language (SDL),WebPro Forum Tutorial, Int. Eng. Consortium.

[5] C. Bourhfir, R. Dssouli, E. Aboulhamid, and N. Rico. Automatic executable test case generation for extended finite state machine protocols. In *IWTCS'97*, pages 75–90, 1997.

[6] S. Cheung and J. Kramer. *Context Constraints for Compositional Reachability Analysis*. *ACM Transactions on Software Engineering and Methodology*, 5(4), October 1996.

[7] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Pasareanu, Robby, and H. Zheng. Bandera: Extracting finite-state models from Java source code. In 22^{nd} *International Conference on Software Engineering (ICSE'2000)*, pages 439–448, Los Alamitos, California, USA, June 2000. IEEE Computer Society Press.

[8] D. Drusinsky. *Modeling and Verification Using UML Statecharts: A Working Guide to Reactive System Design, Runtime Monitoring and Execution-based Model Checking*. Elsevier Inc, 2006.

[9] B. Dutertre and V. Stavridou. Avionics systems requirements: A comparison of RSML and SCR. In *Irish Signals and Systems Conference*, Dublin Institute of Technology, 1998.

[10] R. Eshuis and R. Wieringa. Tool support for verifying UML activity diagrams. *IEEE Transactions on Software Engineering*, 30(7):437–447, 2004.

[11] C. N. Fischer and R. J. LeBlanc. *Crafting a Compiler*. Benjamin/Cummings Series in Computer Science. Benjamin/Cummings Publishing Company, Menlo Park, CA, 1988.

[12] M. R. Garey and D. S. Johnson. *Computers and Intractability*. W. H. Freeman and Company, 1979.

[13] N. Gold, M. Harman, D. Binkley, and R. M. Hierons. Unifying program slicing and concept assignment for higher-level executable source code extraction. *Software Practice and Experience*, 35(10):977–1006, 2005.

[14] M. Harman and S. Danicic. Using program slicing to simplify testing. *Software Testing, Verification and Reliability*, 5(3):143–162, Sept. 1995.

[15] M. P. E. Heimdahl and M. W. Whalen. Reduction and slicing of hierarchical state machines. In *Proc. Fifth ACM SIGSOFT Symposium on the Foundations of Software Engineering*, Zurich, Switzerland, 1997. Springer–Verlag.

[16] J. E. Hopcroft and J. D. Ullman. *Formal Languages and their Relation to Automata*. Addison-Wesley, Reading, MA, 1969.

[17] L. Ilie, R. Solis-Oba, and S. Yu. Reducing the size of NFAs by using equivalences and preorders. In *CPM*, volume 3537 of *LNCS*. Springer-Verlag, 2005.

[18] B. Korel, G. Koutsogiannakis, and L. H. Tahat. Model-based test prioritization heuristic methods and their evaluation. In *A-MOST '07: Proceedings of the 3rd international workshop on Advances in model-based testing*, pages 34–43, USA, 2007. ACM.

[19] B. Korel, I. Singh, L. Tahat, and B. Vaysburg. Slicing of state based models. In *IEEE International Conference on Software Maintenance (ICSM'03)*, pages 34–43, Los Alamitos, California, USA, Sept. 2003. IEEE Computer Society Press.

[20] B. Korell. Private communication, 2009.

[21] S. Labbé and J.-P. Gallois. Slicing communicating automata specifications: polynomial algorithms for model reduction. *Formal Aspects of Computing*, 20(6):563–595, 2008.

[22] S. V. Langenhove and A. Hoogewijs. SV_tL: System verification through logic tool support for verifying sliced hierarchical statecharts. In *Lecture Notes in Computer Science, Recent Trends in Algebraic Development Techniques*, pages 142–155, Berlin, 2007. Springer.

[23] D. Lee, K. K. Ramakrishnan, W. M. Moh, and U. Shankar. Protocol specification using parameterized communicating extended finite state machines - a case study of the ATM ABR rate control scheme. In *Proceedings of the International Conference on Network Protocols (ICNP '96)*, page 208, Washington, DC, USA, 1996. IEEE Computer Society.

[24] T. Maibaum, P. Kan, and K. Lano. Systematising reactive system design. In *Algebraic Methodology and Software Technology*, volume 1548 of *LNCS*. Springer-Verlag, 1999.

[25] W. E. McUmber and B. H. C. Cheng. A general framework for formalizing UML with formal languages. In *ICSE '01: Proceedings of the 23rd International Conference on Software Engineering*, pages 433–442, Washington, DC, USA, 2001. IEEE Computer Society.

[26] OMG. OMG unified modeling language specification 2.2, Feb. 2009. `http://www.omg.org/cgi-bin/doc?formal/2009-02-02`.

[27] H. A. Partsch. *The Specification and Transformation of Programs: A Formal Approach to Software Development*. Springer, 1990.

[28] A. Sanchez, E. Aranda-Bricaire, F. Jaimes, E. Hernandez, and A. Nava. Synthesis of product-driven coordination controllers for a class of discrete-event manufacturing systems. *Robotics and Computer-Integrated Manufacturing*, 26(4), August 2010.

[29] F. Strobl and A. Wisspeintner. Specification of an elevator control system – an autofocus case study. Technical Report TUM-I9906, Technische Universität München, 1999.

[30] F. Tip. A survey of program slicing techniques. Technical Report CS-R9438, Centrum voor Wiskunde en Informatica, Amsterdam, 1994.

[31] J. Wang, W. Dong, and Z.-C. Qi. Slicing hierarchical automata for model checking UML statecharts. In *Proceedings of the 4th International Conference on Formal Engineering Methods (ICFEM)*, pages 435–446, UK, 2002. Springer-Verlag.

[32] M. Weiser. Program slicing. In 5^{th} *International Conference on Software Engineering*, pages 439–449, San Diego, CA, Mar. 1981.

[33] R. Y. Zaghal and J. I. Khan. EFSM/SDL modeling of the original TCP standard (RFC793) and the congestion control mechanism of TCP Reno. Technical Report TR2005-07-22, Internetworking and Media Communications Research Laboratories, Department of Computer Science, Kent State University, 2005.