A General Model Based Slicing Framework

Tony Clark

September 23, 2011

Abstract

Slicing is used to reduce the size of programs by removing those statements that do not contribute to the values of specified variables at a given program location. Slicing aids program understanding, debugging and verification. Slicing could be a useful technique to address problems arising from the size and complexity of industrial scale models; however there is no precise definition that can be used to specify a *model slice*. Model slices are achieved using model transformations, and since models are usually instances of multiple heterogeneous meta-models, model slicing must involve the composition of multiple transformations. This paper proposes a framework that can be used to define both program and model slicing. The framework is used to construct slices of a simple model written in a UML-like language

1 Program Slicing

Slicing is a transformation technique that can be used to reduce the size of a program by focusing on a particular aspect of interest. There are different types of traditional slicing [7, 6]: *static* for all potential program executions; *dynamic* for one particular program execution; *conditioned* for a sub-set of executions. Traditional slicing does not change the program in any way, i.e. statements can only be deleted.

Slicing methods aim to support program comprehension, debugging, measurement verification (particularly using tools such as model checkers and theorem provers that cannot handle the state spaces produced by standard programs [4]), re-engineering, comparison [3], and testing.

A traditional *backward slice* [20, 18] involves an original program, a slicing criterion and a sliced program. Each program statement is uniquely identified and the slicing criterion is a collection of variable names and a statement identifier. The resulting slice is the smallest well-formed sub-program of the original where the two programs agree on the values of the variables at the given statement.

The following examples are taken from [6] (**Example 1**):

Original	Criterion	Slice
`x = 42`		
`d = 23`	`d`	`d = 23`

If variables depend on each other, the sub-program requirement forces earlier statements to be maintained (**Example 2**):

`a = 42`		`a = 42`
`x = 2`		
`b = 23 + a`		`b = 23 + a`
`y = 3`		
`c = b + 2`	`c`	`c = b + 2`

Control structures must be preserved providing that they contribute to the state defined by the slicing criterion. In the following example the number of times the loop is executed depends on the value read in for `i`, however since all possible input must be considered, the loop is part of the slice. The value of `prod` does not depend on the value of `sum` therefore that can be elided (**Example 3**):

```
sum=0
prod=1                                                        prod = 1
read(i)                                                       read(i)
while(i<11) {                                                 while (i<11) {
 sum=sum+i
 prod=prod*i                                                   prod=prod*i
 i=i+1                                                         i=i+1
}                 prod                                        }
```

Forward Slicing, using an original program and a criterion, produces a slice that contains the statements that are affected by the value of the variables in the criterion at the indicated statement. This seems to have no counterpart in modelling since there is no *control flow*.

Dynamic Slicing of programs works like backward slicing except that only a particular collection of values for inputs and variables are considered. Therefore, if the input is `11`, then the slice of **Example 3** is (**Example 4**):

```
sum=0
prod=1
read(i)                                                            prod = 1
while(i<11) {                                                      read(i)
 sum=sum+i
 prod=prod*i
 i=i+1
}                 prod
```

Conditioned Slicing allows program slicing to be viewed as a spectrum since the values of variables and inputs are specified using predicates on their values at given statements. For example, if `pos(next(input))=true` (**Example 5**):

```
read(a)                                                                  read(a)
if(a<0)
 a=-a
x=1/a       x                                                            x=1/a
```

Like programs, models contain variables and therefore we can make the distinction between backward (static), dynamic and conditioned slicing.

All program slicing techniques outlined above involve structural projections on the original program. *Amorphous Slicing* [7] allows the slice to change the program in any way in order to meet the slicing criterion. Whilst models can also be changed and retain the original semantics, amorphous slicing seems to introduce a new category of sophistication and we will not consider it further.

Unions of program slices are introduced in [5] as a means to combine dynamic slices. The union of multiple slices is the smallest program that can be sliced to produce all the individual component slices. Constructing the union of multiple slices involved the composition of program transformations.

2 Model Slicing

Recently, slicing has been applied to state machines [1] where similar benefits as those listed above for program slicing are claimed. State machine slicing is an example of applying slicing to a model of a system rather than to the system implementation. Although models of systems are, by definition, less complex than the systems they model, they can still become large and complex, contain errors, provide useful properties that can be measured, require maintenance and testing, and need to be compared. Therefore the potential benefits claimed for program slicing ought also to apply to system models.

It is not clear what it means to slice a model. It is not even clear what we mean by the term *model*. Given that we are claiming benefits in terms of large models, the rest of this paper will use UML as the language that is representative of industrially-relevant large-scale modelling languages. As described in [1] a number of definitions are possible for state machine slicing. However, system models such as those represented in terms of the UML-family of languages are much more complex than state machines (and contain state machine sub-languages).

Several attempts have been made at slicing UML class diagrams. The approach in [8] describe context-free slicing of UML class models where the issue of *context* is defined to be object location, which is a dynamic property; therefore context-free slicing is a static slice of a structural model. As noted in [8] the criteria used for slicing a model is more complex than that used in program or state machine slicing since there are more types of elements and relationships; they note that OCL should be used to express the slicing criteria. A similar approach is used to modularize the UML meta-model into collections of components that are relevant to the different UML diagram types in [2] although the predicate used to determine the slicing criteria is fixed in terms of traversing the meta-model elements starting with a collection of supplied classes. Class models are sliced together with OCL invariants in [17] thereby reducing the state-space explosion that would otherwise occur when using a model-checker (in this case Alloy) to verify a class-model.

UML statecharts can be sliced as described in [19, 13, 11] although these approaches do not generalize the results to include other parts of the UML language family. Both static and dynamic aspects of UML can be combined and sliced as described in [9, 10] where class and sequence diagrams are merged into a single representation (a model dependency graph MDG) that can be subsequently sliced to show partial dynamic and structural information resulting from criteria containing both structural and dynamic constraints. Slicing UML sequence diagrams in order to generate test cases is described in [16, 15].

UML sequence diagrams (or scenarios) are essentially underspecified executions of a program, they can be used as slicing criteria for programs as described in [14] where the scenario significantly limits the scope of the slice.

The approach described in [12] uses the logic-based sub-language of UML: UML-RSDS. Since this language is formal, it has a semantics as advocated by the framework in this paper. Although it does not introduce the notion of a projection, least slice, or a slicing criterion explicitly represented as a semantic domain object, the UML-RSDS approach could support the general slicing framework that is proposed here.

All of the approaches to model slicing outlined above are limited in some regard, either in terms of a specific set of model types (usually a sub-set of UML) or in terms of requiring the use of specific technologies or algorithms. What does it mean to specify a model-slice? Can we do so without resorting to specific technologies or being limited to specific types of slice? The rest of this paper proposes a general framework for slicing that can be used to specify slices of both programs and models. Section 3 describes the framework, section 4 shows how the framework applies to the examples in section 1, section shows how the framework supports model-slicing by defining a simple modelling language and applying the framework to a simple example model in the language.

3 General Slicing Framework

Figure 1 shows the proposed GSF. Syntax elements are of type Γ. An element has a semantic domain Σ and the semantic interpretation ϕ maps a syntax element γ to a semantic element σ. The mappings $p : \Gamma \rightarrow \Gamma$ are *syntactic projections*. The definition of a projection depends on the specific instance of the framework, for example a homogeneous projection of a program can remove abstract syntax sub-trees providing certain syntax well-formedness rules are satisfied; whereas, an amorphous projection of a program holds between programs when they have semantic elements that are related via a semantic slice.

$$\begin{tikzcd}
\gamma : \Gamma \arrow[rrr, "\phi"] \arrow[dr, "p'"] \arrow[dd, "p"'] & & & \sigma : \Sigma \arrow[dl, "q'"'] \arrow[dr, "x"] \arrow[dd] & \\
& \gamma'' : \Gamma \arrow[r, "\phi"] \arrow[dl, "p''"'] & \sigma'' : \Sigma \arrow[r] \arrow[dr, "q''"] & q \arrow[r, "z"] & \omega : \Sigma \\
\gamma' : \Gamma \arrow[rrr, "\phi"'] & & & \sigma' : \Sigma \arrow[ur, "y"'] &
\end{tikzcd}$$

Figure 1: General Slicing Framework (GSF)

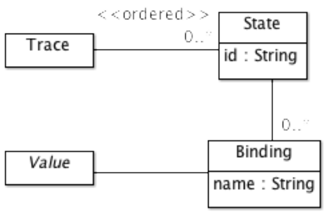


Figure 2: A Simple Semantic Domain for Programming

The mappings $q : \Sigma \to \Sigma$ are *semantic projections*. Again, the definition of a semantic projection depends on the instance of the framework. For any element there exists the identity projection. Any element can be projected uniquely to an *empty* or *terminal* element.

A slice of an original element γ with respect to a semantics ϕ and a slicing criterion (ω, x), is a pair (γ', y) such that the diagram shown in figure 1 commutes and for any other slice (γ'', z) there are projections p'' and q''. The latter condition ensures that the chosen slice is a *smallest* slice.

Since both the syntactic and semantic domain have projection mappings, they have *product* elements such that if γ_1 and γ_2 are syntax structures then $\gamma_1 \times \gamma_2$ is a syntax structure that can be projected onto *both* γ_1 and γ_2. Note that product elements are not guaranteed to exist, but if they do then they are likely to be unique up to isomorphism.

The importance of products is that, given a syntax structure γ with a slicing criterion (ω_1, x_1) leading to a slice γ_1' and a second slicing criterion (ω_2, x_2) leading to a slice γ_2' then products can be used to find a slice $\gamma_1' \times \gamma_2'$. If the component slices have been characterized using tranformations p_1 and p_2 then there is a combined transformation $p_1 \times p_2$.

Therefore to set up the framework for a particular application domain you will need: **syntax** (Γ) The things you want to slice; **semantics** (Σ, ϕ) The meaning of the syntax defined as a semantic domain and a semantic mapping; **projections** (p, q) Relationships that hold between syntactic and semantic elements and that remove items that are not of interest; **criterion** (x, ω) The things that remain invariant, defined in terms of the semantic domain; **products** If the syntax and semantic domain also define products for elements nd projections then there is scope for slicing composition. As a slicer, your job is to find a smallest syntax element that can be generated from the original element so that the semantics are the same and where the slicing criterion holds.

4 GSF Applied to Programs

Figure 2 is a model for the semantic domain of programs. Trace instances will be represented as sequences of sets of maplets in the usual way.

Example 1: the semantics is $\sigma = \{[\{x \mapsto 41; d \mapsto 23\}]\}$ The slice criterion requires that any trace in σ when projected onto the second step has a value for d, so $\omega = \{\{d \mapsto v\} \mid v \in V\}$ Therefore the smallest set of traces for which there exists a structural projection is $\sigma' = \{[\{d \mapsto v\} \mid v \in V\}]\}$.
Example 2: the semantics is $\sigma = \{[\{a \mapsto 42\}, \{a \mapsto 42; x \mapsto 2\}, \{a \mapsto 42; x \mapsto 2; b \mapsto 65\}, \{a \mapsto 42; x \mapsto 2; b \mapsto 65; y \mapsto 3\}, \{a \mapsto 42; x \mapsto 2; b \mapsto 65; y \mapsto 3; c \mapsto 67\}]$ The slicing criterion requires that c has the same value in the original and the slice $\omega = \{c \mapsto v \mid v \in V\}$. The projection x maps σ states onto c in the 5th step. Therefore the smallest structural projection on the original program has a semantics $\sigma' = \{[\{a \mapsto 43\}, \{a \mapsto 42, b \mapsto 65\}, \{a \mapsto 42; b \mapsto 65; c \mapsto 67\}]\}$ where y maps states onto c in the 3rd step.
Example 3: the semantics is $\sigma = \{[\{\texttt{sum} \mapsto 0\}, \{\texttt{sum} \mapsto 0; \texttt{prod} \mapsto 1\}, \{\texttt{sum} \mapsto 0; \texttt{prod} \mapsto 1; i \mapsto v\}] + \text{trace}(v) \mid v \in V\}$] where trace(0) = [] and trace(v) is

$$\text{trace}(v-1) + [\{\texttt{sum} \mapsto \sum_{i \in 0 \ldots v} i; \texttt{prod} \mapsto !v; i \mapsto v\}]$$

The criterion requires that `prod` is a specific value $\omega = \{\texttt{prod} \mapsto v \mid v \in V\}$ and the projection x maps each trace onto the *last* maplet for `prod`. Therefore, the smallest structural projection omits the variable `sum` and the projection mapping y maps each trace onto the last maplet for `prod`.
Example 4: dynamic slicing defines the input to be `11` so the semantics is limited to $\sigma = \{[\{\texttt{sum} \mapsto 0\}, \{\texttt{sum} \mapsto 0; \texttt{prod} \mapsto 1\}, \{\texttt{sum} \mapsto 0; \texttt{prod} \mapsto 1; i \mapsto 11\}]\}$ and the projected semantics is therefore $\sigma' = \{[\{\texttt{prod} \mapsto 1\}, \{\texttt{prod} \mapsto 1; i \mapsto 11\}]\}$.
Example 5: has a semantics $\sigma = \{[\{a \mapsto v\}, \{a \mapsto v; x \mapsto 1/v\}] \mid v \in V, \text{pos}(v)\}$. The criterion requires the value of x to be invariant and therefore the projected semantics is the same, leading to the appropriate structural projection on the original program.

5 GSF Applied to Models

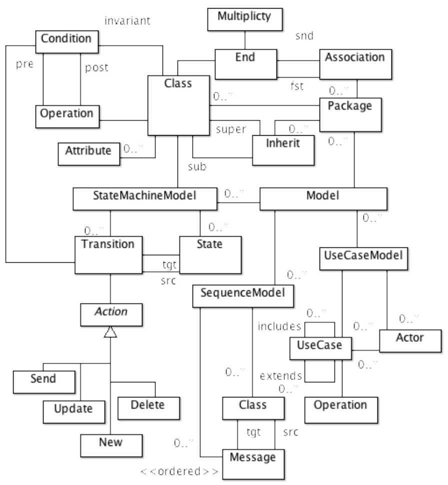


Figure 3: A Simple Modelling Language

The syntax for a modelling language (Γ) is defined as a meta-model and associated well-formedness constraints. Consider the meta-model defined in figure 3. It is intended to be representative of UML modelling where a model consists of both static and dynamic aspect. The static aspect of a system is expressed as a collection of packages containing classes and associations. The dynamic aspect of a system is expressed as use-case models, state-machines and sequence models.

The meta-model does not include all of the detail of the language, for example it omits structural information for each of the meta-classes. Therefore, it is not possible to give the details of the well-formedness constraints; however these are standard, for example requiring that all classes in a package to have unique names.

The GSF requires that syntax structures define a projection relation that should only hold between well-formed elements. The structure of programs is fairly simple, i.e. trees where certain nodes are labelled as statements, and the projections elide statement sub-trees.

The modelling language defined in figure 3 has graphs, rather than trees, as instances. Therefore projections hold between models, $p : \gamma \to \gamma'$ when the model γ' is a sub-graph of γ, when OCL constraints in γ imply constraints on projected elements in γ', and when both are well-formed. Therefore: packages may delete classes and associations; classes may delete attributes, operations; use-case models may delete use-cases and actors; sequence models may delete messages and classes; state-machine models may delete states and transitions.

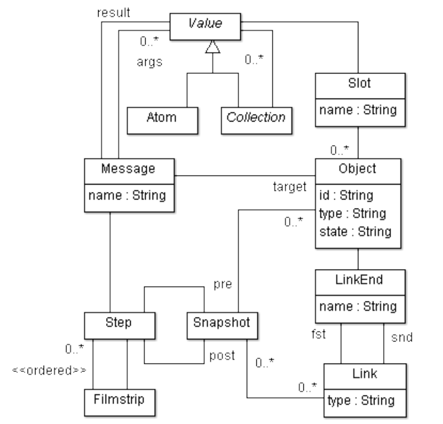


Figure 4: A Simple Semantic Domain for Modelling

The semantic domain for the modelling language (Σ) is defined as a model in figure 4. The meaning of a model is a filmstrip that contains an ordered sequence of steps. Each step is a relation between a *pre* and a *post* snapshot, a message, and a nested filmstrip. The message has a name, some arguments, a return value, and a target object. A snapshot contains objects and links.
In order to express the slicing criterion for a model, we need to be able to construct instances of the semantic domain. This can be expressed as a term as follows. Objects with id i, type t and state σ are represented as $(i, \sigma, t)[s = v; \ldots]$. Links of type t between objects with ids i and j are represented as (t, i, j). Snapshots are represented as a pair of sets(O, L). Messages are represented as $v = i \leftarrow m(v, \ldots, v)$. A step is represented as a 4-tuple (s, m, f, s). Finally, a filmstrip is a sequence of steps.

The well-formedness constraints on the semantic domain require that slots have unique names for each object, objects have unique identifiers for each snapshot, where a snapshot contains a link then the snapshot must also contain the objects attached ot the link ends. An atomic step has an empty nested filmstrip. The difference between pre and post snapshots in an atomic step arises due to: single slot changes; single object or link creation; single object or link deletion.

Since semantic domain terms are trees, we define semantic projections $q : \sigma \to \sigma'$ to hold when σ' is a structural projection of σ. Therefore: slots may be deleted from objects, elements deleted from sets; steps deleted from filmstrips, *etc.*

5.1 Case Study

Figure 5 shows a use-case model for a library. It shows the externally visible operations that are available to the users of the system. Figure 6 shows a package that defines classes for a simple library that supports registration of readers, addition of books, and book borrowing/return. The library timestamps borrowing records and imposes a limit on the length of time a book may be borrowed. If the limit is exceeded then a fine must be paid to reset the borrowing date. A book may only be returned if the limit has not been exceeded. The following is an example of an OCL constraint:

```
context Library::borrow(reader,book)
  pre not hasFine(reader) and
  post borrows->includes(b | b.reader = getReader(reader) and b.book = getBook(book) and
                             b.date = date())
```

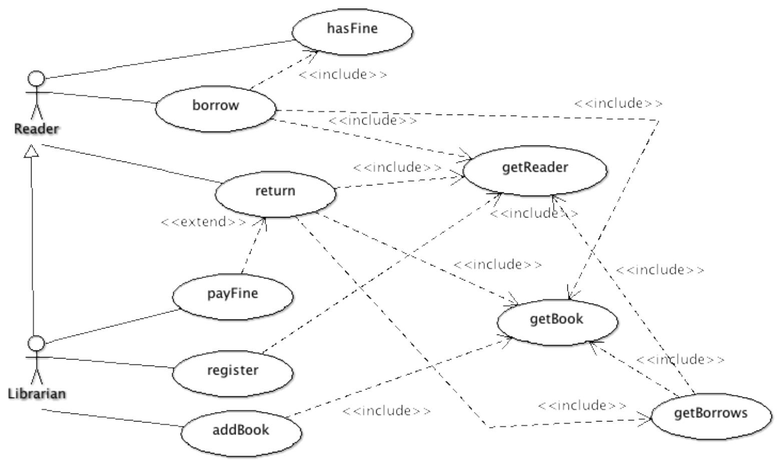


Figure 5: Library: Use-Cases

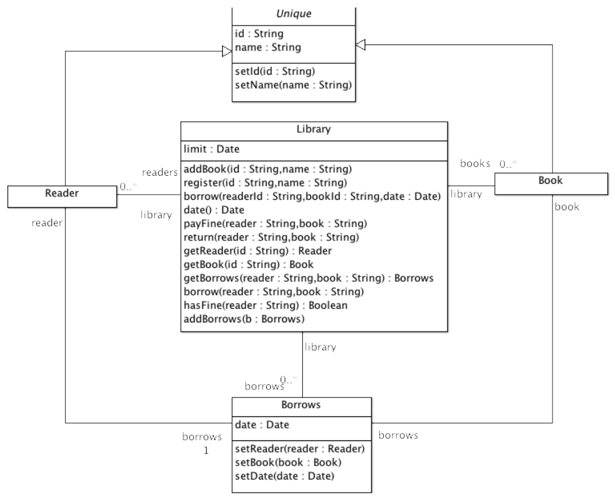

Unique
id : String
name : String
setId(id : String)
setName(name : String)
Library
limit : Date
addBook(id : String,name : String)
register(id : String,name : String)
borrow(readerId : String,bookId : String,date : Date)
date() : Date
payFine(reader : String,book : String)
return(reader : String,book : String)
getReader(id : String) : Reader
getBook(id : String) : Book
getBorrows(reader : String,book : String) : Borrows
borrow(reader : String,book : String)
hasFine(reader : String) : Boolean
addBorrows(b : Borrows)
readers
0..*
Reader
library
reader
books
0..*
Book
library
book
library
borrows
0..*
Borrows
date : Date
setReader(reader : Reader)
setBook(book : Book)
setDate(date : Date)
borrows
1
borrows

Figure 6: Library: Package

In general, each class in the library model may have a state machine that defines the life-cycle states of the class-instances. Figure 7 shows a typical state machine for the `Reader` class. When a reader is first created it is in the `new` state. After the name and id slots have been set the reader becomes `initialized`. Once the reader object is added to the library it becomes `registered`. From this point on, a reader may borrow and return books; it changes from `registered` to `borrowing` and back again. If the borrowing limit is ever exceeded then the reader becomes a `debtor`, reverting to `borrowing` when the fine is paid.

Each of the use-cases may give rise to a sequence of operation calls. Figure 8 shows an example of a sequence of calls that occur when a reader borrows a book. The calls are sequenced from left to right and from top to bottom.

The semantics of the model is given as an instance of the model in figure 4. For example, borrowing is described by the following step:

```
(s1,ok = 0 <- borrow(fred,b),f,s6)
  where
    s1 = ({(0,*,Library)[limit=1],(1,*,Book)[name=n,id=1],(2,initialized,Reader)[name=fred,id=2]},{})
    f = [(s1,false = 0 <- hasFine(fred),...,s1),(s1,2 = 0 <- getReader(fred),...,s1),
          (s1,1 = 0 <- getBook(n),...,s1),(s1,d = 0 <- getDate(),...,s1),
          (s1,3 = 0 <- new(Borrows),[],s2),(s2,ok = 3 <- setReader(2),...,s4),...]
```

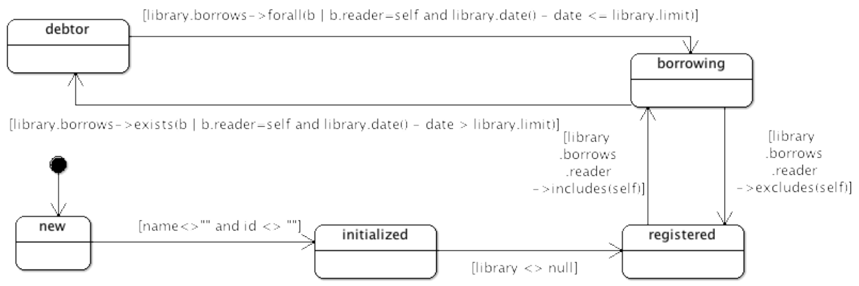

debtor
[library.borrows->forall(b | b.reader=self and library.date() - date <= library.limit)]
borrowing
[library.borrows->exists(b | b.reader=self and library.date() - date > library.limit)]
[library .borrows .reader ->includes(self)]
[library .borrows .reader ->excludes(self)]
new
[name<>"" and id <> ""]
initialized
registered
[library <> null]

Figure 7: Reader: State-Machine

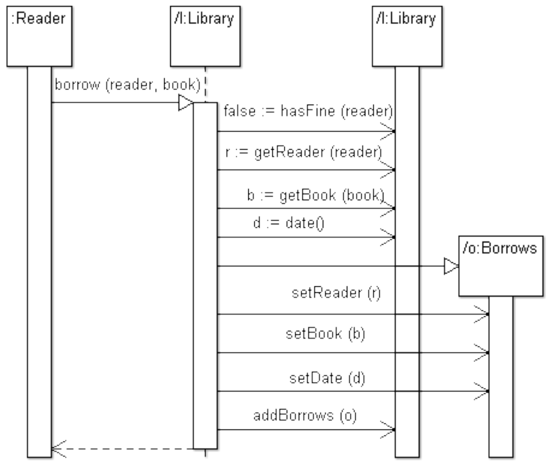


Figure 8: Borrow: Sequence Model

```
s2 = ({(0,*,Library)[limit=1],(1,*,Book)[name=n,id=1],
        (2,initialized,Reader)[name=fred,id=2],(3,*,Borrows)[]},{})
```

5.2 Slices

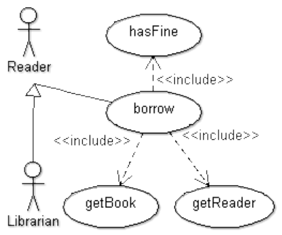
Section 5.1 has defined a simple model and the semantic definition for the modelling language provides a way of constructing a set of filmstrips as the meaning of any model. This section defines two slices of the library; the first isolates that part of the model that describes what happens when a book is borrowed and the second refines the first slice by removing the steps that occur during a borrowing operation.

The slicing criterion that is used to isolate borrowing behaviour is $(B_1)*$ where B_1 is defined as follows:

```
{[(s1,v = l <- borrow(r,b),F,s2)] |
  F <- Filmstrip, s1 <- Snapshot, s2 <- Snapshot,v <- Value, l <- Id, r <- Str, b <- Str}
```

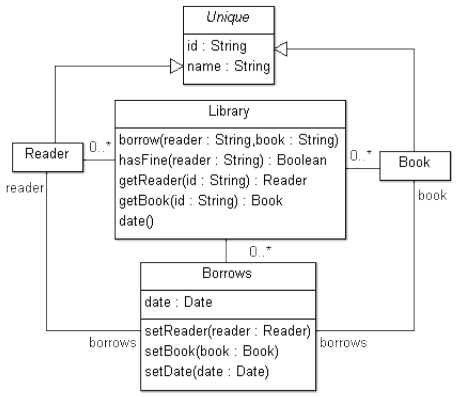
The mapping part, x_1, of the slicing criterion projects any filmstrip to the sequence of `borrow` steps that it contains. Therefore, any filmstrip containing no borrowing messages will be mapped to the empty filmstrip. Our task is to find the smallest projection of the original model whose filmstrips can be mapped onto $(B_1)*$ such that the diagram in figure 1 commutes.

Syntactic projections on use-case models can elide use-cases and their relationships. Recall that use-cases are linked to operations that name messages in steps, therefore, the resulting slice of figure 5 is:

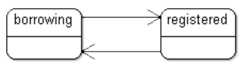


where use-cases that do not initiate or participate in borrowing are elided.

Packages may remove structural elements and may weaken constraints, therefore the class diagram:


Unique
id : String
name : String
Library
borrow(reader : String,book : String)
hasFine(reader : String) : Boolean
getReader(id : String) : Reader
getBook(id : String) : Book
date()
Reader
reader
0..*
0..*
Book
book
0..*
Borrows
date : Date
setReader(reader : Reader)
setBook(book : Book)
setDate(date : Date)
borrows
borrows

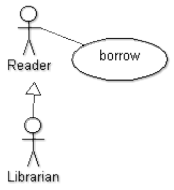
is the result of slicing the package and shows the structure and objects necessary to construct filmstrips whose steps are defined by B_1. All constraints that are not related to borrowing are removed.

The state machine model is reduced to the two states that can occur in borrowing filmstrips: 

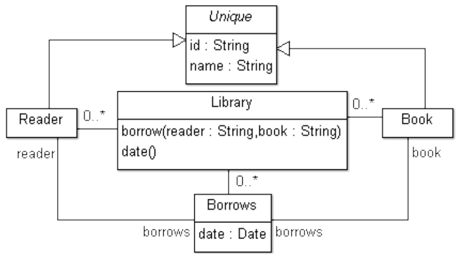
The slice produces no change to the sequence model shown in figure 8.

Now consider a refinement to the slice criterion that ignores any behaviour that occurs *within* a `borrow` step. The set B_2 is a projection of B_1 and is defined as follows:

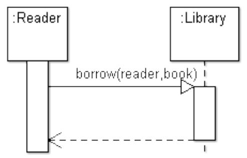
```
{[(s1,v = l <- borrow(r,b),[],s2)] | s1 <- Snapshot,s2 <- Snapshot,v <- Value,l <- Id,r <- Str,b <- Str}
```

The mapping part x_2 is a refinement of x_1 that maps all nested filmstrips to []. The sliced use-case is: 

The sliced package is:



There is no change in the state machine between the two slices because readers are affected the same way whether we include the details of the borrowing operation or not. The sequence diagram is sliced in order to slide the nested operation calls as follows:



Another slicing criteria might be a refinement of B_2 which removes any of the elements in the snapshots. The resulting slice would be empty models except for a single use-case: `borrow`. Dynamic slicing of models implies that the criterion is a specific filmstrip and that the slice produces the smallest model that generates the filmstrip. Conditioned slicing uses predicates to construct the criterion, for example, by requiring that all filmstrips include a fine payment.

6 Types of Slicing

The previous section has described how the GSF can be applied to models. Slicing of each type of model has been described in terms of a simple case study. This section reviews the different types of program slicing and discusses them in the context of model slicing.

Static Slicing A static slice of a model cannot limit the values of object slots, or the multiplicty of links in the slicing criterion. The criterion may limit the structural complexity of a model by ignoring classes, operations, attributes and associations, but cannot express predicates over their instances. In the case of the library, it would be possible to slice on `Library::date`, but not slice on a particular date. Most work on model slicing is limited to static slicing and expresses the slicing criterion in terms of a syntax transformation. However, the GSF approach requires the slicing criterion to be expressed in terms of a semantic domain and therefore ensures that all possible syntaxtic elements that are related to a selected syntax element are present in the resulting slice.

Dynamic Slicing A dynamic slice of a model is a particular instance of a semantic model. In the case of the library it is a single filmstrip, each step contains a particular operation call with respect to a fixed collection of objects and links. Dynamic model slicing only makes sense when a model has a dynamic semantics. For executable models, dynamic slicing can be used for debugging where the slicing criterion is a trace of model execution that terminates in an error. Unlike standard program slicing, there is no single starting point for an executable model, and therefore the criterion can limit the model based on the steps from the starting snapshot to the end snapshot of a filmstrip.

Conditioned Slicing Conditioned slicing is the most general case where a set of possible semantic elements is defined using a collection of predicates. The GSF explains provides a simple explanation of how a conditioned slice is constructed, without having to resort to complex analysis of syntax structures. Conditioned slicing can be used for controlling the size and complexity of models, and for debugging. In addition, they can be used for analysis of models, for example: is it possible to slice the model given a criteria that includes two readers with the same name?

Slicing Unions Programs typically a single homogeneous structure. UML-style models are constructed from multiple homogeneous and heterogeneous sub-models and therefore unions of slices can occur in various different ways. A single model may contain multiple models of the same type (for example class models) in which case name-spaces are used to define repeated occurrences of the same element. A slice may apply to any one of the components and multiple slices can be merged.

$$\text{M-1}\ \dfrac{\begin{array}{c} S \vdash M_1 < M_1' \\ S \vdash M_2 < M_2' \end{array}}{S \vdash M_1 \oplus M_2 < M_1' \oplus M_2'} \qquad \text{M-2}\ \dfrac{\begin{array}{c} S \vdash M_1 < M_2 \\ S \vdash p \Rightarrow q \end{array}}{S \vdash M_1\, \mathbf{when}\, p < M_2\, \mathbf{when}\, q}$$

$$\text{M-3}\ \dfrac{\begin{array}{c} S \vdash A_1 < A_2 \\ S \vdash C_1 < C_2 \end{array}}{S \vdash (C_1, A_1) < (C_2, A_2)} \qquad \text{G-1}\ \dfrac{S \vdash X < X}{}$$

$$\text{G-2}\ \dfrac{\begin{array}{c} S \vdash X_1 < X_1' \\ S \vdash X_2 < X_2' \end{array}}{S \vdash X_1 \cup X_2 < X_1' \cup X_2'} \qquad \text{G-3}\ \dfrac{S \vdash \emptyset < X}{}$$

$$\text{C-7}\ \dfrac{S \vdash (C)[] < (C)[] \oplus c}{} \qquad \text{C-4}\ \dfrac{S \vdash C_1 < C_2}{S \vdash \{C_1\} < \{C_2\}}$$

$$\text{C-5}\ \dfrac{\begin{array}{c} S \vdash p \Rightarrow q \\ S \vdash c < c' \end{array}}{S \vdash c\, \mathbf{when}\, p < c'\, \mathbf{when}\, q} \qquad \text{C-6}\ \dfrac{\begin{array}{c} S \vdash c_1 < c_1' \\ S \vdash c_2 < c_2' \end{array}}{S \vdash c_1 \oplus c_2 < c_1' \oplus c_2'}$$

$$\text{A-1}\ \dfrac{m_i = m_i'\, \mathbf{or} (m_i = 1\, \mathbf{and}\, m_i' = *)}{S \vdash \{(A, c, m_1, c', m_2)\} < \{(A, c, m_1', c', m_2')\}}$$

Figure 9: A Slicing Theory

7 Constructing Slices

A model M may be (C, A) a set of classes C and associations A, a model with an invariant p that holds *between* classes and associations $M\, \mathbf{when}\, p$, or a composition of models $M \oplus M'$. An association is (a, c, m, c', m') where a is an association name, c, c' are class identifiers and m, m' are multiplicities (1 or $*$). Well formedness constraints on models require classes named in associations to be present, names to be unique and predicates to reference classes and associations defined in the model. A class $(C)[n \mapsto T]$ has a name C and a slot n with type T, or has an invariant $c\, \mathbf{when}\, p$, or is empty $(C)[]$, or is a combination of classes $c \oplus c'$ where the identifiers and types on common attribute names agree.

Figure 9 defines a slicing theory for models that is consistent with the GSF. Rules labelled M refer to models, rules labelled A refer to attributes and rules labbeled C refer to classes. The theory defines a relation $S \vdash M < M'$ that holds when model M is a slice of model M' with respect to the slicing criterion S expressed as a snapshot and where S is an instance of M. The key rules are: M-2 that requires model invariants to be strengthened; A-1 that requires association multiplicities to be tightened; C-5 that requires class invariants to be strengthened; C-7 that allows attributes to be deleted. The theory can be used to specify a model transformation for model slicing. The theory can be extended for heterogeneous languages (as in UML). In addition, multiple transformations can be defined on the same model and the theory extended to specify the required single slice that is consistent with merging the individual transformations.

8 Conclusion

This paper has reviewed *program slicing* and observed that many of the benefits may apply to models. A general framework for slicing any structured elements has been proposed and shown to be a basis for both program and model based slicing. The framework provides a check-list of features that must be defined before any structured elements can be sliced.

References

[1] K. Androutsopoulos, D. Clark, M. Harman, Z. Li, and L. Tratt. Control dependence for extended finite state machines. *Fundamental Approaches to Software Engineering*, pages 216–230, 2009.

[2] J.H. Bae, K.M. Lee, and H.S. Chae. Modularization of the UML metamodel using model slicing. In *Information Technology: New Generations, 2008. ITNG 2008. Fifth International Conference on*, pages 1253–1254. IEEE, 2008.

[3] David W. Binkley and Keith Brian Gallagher. Program slicing. In *ADVANCES IN COMPUTERS*, pages 1–50. Academic Press, 1996.

[4] J.C. Corbett, M.B. Dwyer, J. Hatcliff, S. Laubach, C.S. Pasareanu, and H. Zheng. Bandera: Extracting finite-state models from java source code. In *Software Engineering, 2000. Proceedings of the 2000 International Conference on*, pages 439–448. IEEE, 2000.

[5] Csaba Faragó. Union slices for program maintenance. In *Proceedings of the International Conference on Software Maintenance (ICSM'02)*, pages 12–, Washington, DC, USA, 2002. IEEE Computer Society.

[6] K. Gallagher and D. Binkley. Program slicing. In *Frontiers of Software Maintenance, 2008. FoSM 2008.*, pages 58 –67, 28 2008-oct. 4 2008.

[7] M. Harman and R. Hierons. An overview of program slicing. *Software Focus*, 2(3):85–92, 2001.

[8] H. Kagdi, J.I. Maletic, and A. Sutton. Context-free slicing of UML class models. 2005.

[9] J.T. Lallchandani and R. Mall. Slicing UML architectural models. *ACM SIGSOFT Software Engineering Notes*, 33(3):1–9, 2008.

[10] J.T. Lallchandani and R. Mall. Integrated state-based dynamic slicing technique for UML models. *Software, IET*, 4(1):55–78, 2010.

[11] K. Lano. Slicing of UML state machines. *AICÕ09*, 2009.

[12] K. Lano and S. Kolahdouz-Rahimi. Slicing of uml models.

[13] V. Ojala. *A slicer for UML state machines.* Helsinki University of Technology, 2007.

[14] J. Qian and B. Xu. Program slicing under UML scenario models. *ACM SIGPLAN NOTICES*, 43(2):21, 2008.

[15] P. Samuel and R. Mall. A Novel Test Case Design Technique Using Dynamic Slicing of UML Sequence Diagrams. *e-Informatica Software Engineering Journal Selected full texts*, 2(1):61–77, 2008.

[16] P. Samuel, R. Mall, and S. Sahoo. UML Sequence Diagram Based Testing Using Slicing. In *INDICON, 2005 Annual IEEE*, pages 176–178. IEEE, 2005.

[17] A. Shaikh, R. Clarisó, U.K. Wiil, and N. Memon. Verification-driven slicing of UML/OCL models. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, pages 185–194. ACM, 2010.

[18] F. Tip. A survey of program slicing techniques. *Journal of programming languages*, 3(3):121–189, 1995.

[19] S. Van Langenhove. Towards the Correctness of Software Behavior in UML: A Model Checking Approach Based on Slicing. 2006.

[20] M. Weiser. Program slicing. In *Proceedings of the 5th international conference on Software engineering*, pages 439–449. IEEE Press, 1981.