In Proceedings of APSEC'95'96, Dec 7-9, Brisbane, Australia, IEEE CS Press.

Software environment support for integrated formal program specification and development

John C. Grundy† and John G. Hosking††

†Department of Computer Science
University of Waikato
Private Bag 3105, Hamilton, New Zealand
jgrundy@cs.waikato.ac.nz

††Department of Computer Science
University of Auckland
Private Bag 92019, Auckland, New Zealand
john@cs.auckland.ac.nz

Abstract

Formal program development has gained widespread academic interest as a rigorous software engineering technique. One of the main hurdles for the wider IT industry in adopting these formal techniques is a lack of tools to support their use in combination with more traditional development techniques. This paper describes an integrated environment for object-oriented software development which incorporates formal Object-Z specifications for classes. These formal specification views are kept consistent with more traditional design and implementation views, allowing software developers to design, refine, implement and document their software utilising integrated formal techniques.

Key words: software engineering environments, formal specification, integrated software development, consistency management, Object-Z

1. Introduction

Since its introduction as a programming paradigm in the early 1960's, object orientation has enjoyed increasing popularity. Not surprisingly, its use has spread beyond that of an implementation technique with the development of methodologies and notations for object-oriented (OO) analysis, specification, and design.

A large number of OO analysis and design methodologies and notations have been developed including Booch [3], OOA [5], OMT [24], and [25]. These are all, essentially, informal in their approach and hence do not lead to rigorous system specification. All require textual supplements to more completely define, for example, service behaviour. The methodologies are supported by a variety of CASE tools, some having code generation capabilities such as Rational Rose, for Booch and ObjecTOOL, for OOA.

A variety of formal OO specification approaches have also been developed. These include: LOTOS, which is not strictly an OO specification language but which has been adapted as such; Object-Z [6]; OOZE [1]; Z++ [18]; OSDL [20]; and VDM++ [7]. The advantage of using such specification languages is that a more rigorous and concise specification of systems results. These specifications can also be more easily reasoned with than informal implementation code and designs [4]. Disadvantages of such languages are that they are not directly executable, and, being mathematically based, many people find them too "hard" or "unwieldy". A variety of tools have been developed to support OO specification languages, with some providing prototyping and application shell generation. These include the OOZE tools [1]; the UQ editors [26], supporting Object-Z; and tools supporting OSDL [20].

A problem that both OO analysis and specification methodologies suffer from is that they are disjoint from the implementation of the systems they are defining. The code generation facilities of their support tools are useful, but do not ensure the code and specifications are kept consistent with one another. Similarly, there is no support for keeping OO analyses consistent with OO specifications, contributing to the lack of enthusiasm for the use of the latter. An exception is the Eiffel approach of design by contract, using method pre- and post-conditions, and class invariants to specify classes and their behaviour. This language-based approach, while laudable, has problems with the lack of expressive capability of the assertion language used in comparison with languages such as Object-Z.

While there have been some approaches to combining analysis and specification methodologies, such as the ROOA approach of [19] and the work of [15], there has been little tool support for this. What is needed are tools which support the use of informal requirements specification and OO analysis, integrated with more formal OO specification, and combined with the ability to proceed to design and implementation, all within the one environment. In this paper we describe such an environment, constructed by extending SPE, an existing environment supporting integrated OO analysis, design, and implementation, with views supporting OO specification using a variant of Object-Z.

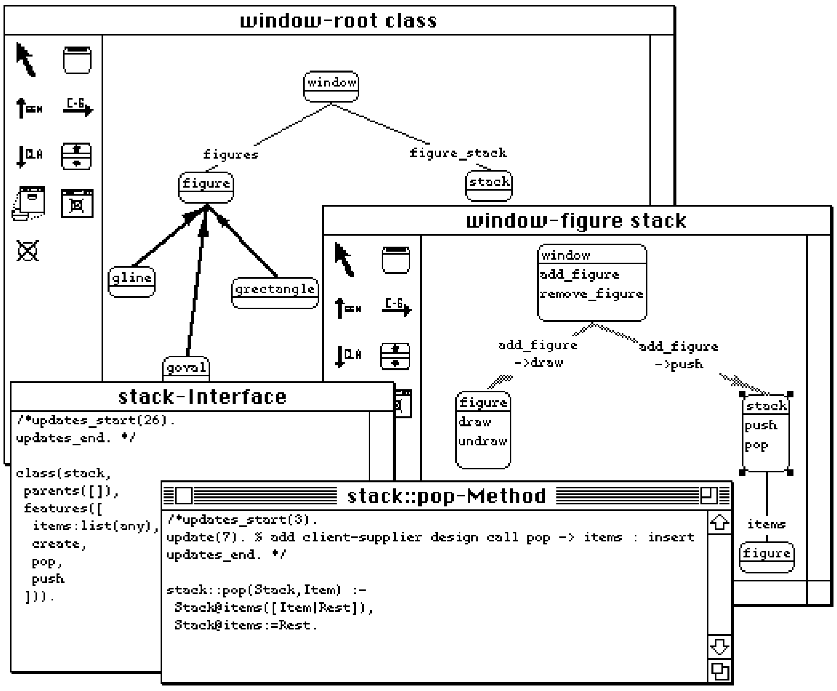
We commence with a brief overview of SPE. This is followed by a description of SPE's extension to include Object-Z views. Section 4 examines issues of consistency within the integrated environment. Section 5 discusses implementation details, while Section 6 provides discussion and conclusions.

2. Overview of SPE

SPE (Snart Programming Environment) provides an integrated environment for OO analysis, design, and implementation using Snart, an OO Prolog [10]. SPE supports multiple views of a system across multiple phases of development. It has a novel approach to consistency management based on the propagation of discrete *change descriptions* between views. These descriptions document a change made in one view and are propagated to all other views that could be affected by the change. The receiving views interpret the change and modify their contents appropriately.

Graphical views support the definition of classes (and their attributes and services), and inter-class relationships important for analysis and design such as (using OOA terminology, but different notation), gen-spec, whole-part, and instance and message connections. For example, Figure 1 shows two graphical views specifying part of the analysis (window 'window-root class') and design (window 'window-figure stack') for a drawing program application. Graphical views such as these allow programmers to define and browse the important analysis and design structures associated with an OO program. Analysis and design views are supplemented by textual documentation views, which can be associated with classes or individual methods. These are used to record informal requirements and specification details.

SPE also provides textual implementation views and two views of the implementation of the stack class are shown in figure 1. The class interface view (window 'stack-Interface') defines the interface to this class. The method implementation view (window 'stack::pop - Method') shows the Snart implementation of the stack's pop method. In this example, a stack stores its items by using a Prolog list attribute of the stack object. SPE's graphical views are interactively edited while textual views are free-edited and parsed.



```
/*updates_start(26).
updates_end. */

class(stack,
 parents([]),
 features([
  items:list(any),
  create,
  pop,
  push
 ])).
```

```
/*updates_start(3).
update(7). % add client-supplier design call pop -> items : insert
updates_end. */

stack::pop(Stack,Item) :-
 Stack@items([Item|Rest]),
 Stack@items:=Rest.
```

Figure 1. Drawing program analysis, design and implementation views in SPE.

SPE's change description mechanism is used to maintain consistency between these views. In many cases, such as the addition, deletion, or change to the name of a class, attribute, or service, or the introduction of an inheritance relationship between classes, other affected views can be automatically modified to reflect the change. This is because the affected property of the system has common semantics in both the graphical and implementation views.

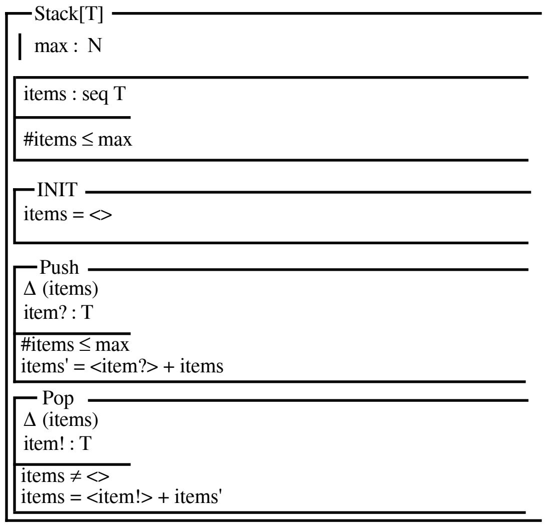
In other cases, such as an implementation view receiving notification of the introduction of an abstract instance or message connection, automatic modification is not possible as there is insufficient implementation information (such as whether an instance connection is to be implemented as an attribute reference or via an intermediate dictionary object). Rather than attempting full consistency in these cases, a partial consistency approach is taken. The received change description is translated into a readable form and inserted as an annotation in the view, as shown in the method implementation view in figure 1. The user is then made responsible for implementing the modifications required to regain full consistency. The same approach is taken with change descriptions received by documentation views. The user is responsible for inspecting the annotations and checking that informal requirements are still met, or that informal specifications are updated to reflect the change.

SPE thus provides an effective approach to integrating tools supporting software development across analysis, design, and implementation. However, it lacks tools to support formal specification of systems, relying instead on informal textual requirements and specifications. To correct this deficiency the integration of Object-Z specification views into SPE has been undertaken, as described in the following section.

3. Adding Object-Z views to SPE

We chose Object-Z as the formal specification language for use with SPE because of its object-oriented features and our experience with it. We use a simple stack example for conciseness in the following discussion, to illustrate these Object-Z views in SPE. Figure 2 shows an Object-Z specification for a simple stack class (adapted from [6]). This specifies that a stack holds items of type T (a generic parameter). Stacks have a maximum number of elements, max, and a sequence of items, items, where the number of items is always constrained to be less than or equal to max. Methods include: INIT, used to initialise a stack object on creation; Push, used to push an item onto the top of the stack; and Pop, used to pop the top item off the stack. Push and Pop alter the items attribute, and have one argument, item, the value to push or the value which was popped. Push is constrained to ensure the number of items does not exceed max, and Pop is constrained to ensure an empty stack is not popped.

Formal specification of classes can be supported by adding Object-Z views to SPE. Figure 3 contains a class specification view showing an Object-Z specification of the stack class. The Object-Z syntax has been adapted somewhat to a purely textual form for simplicity of implementation within the SPE framework.

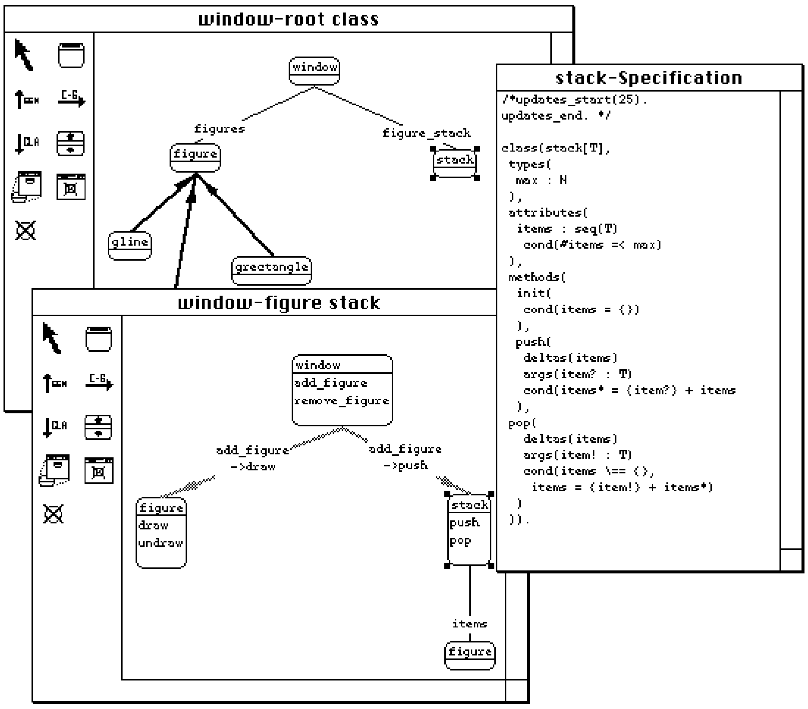


```
Stack[T]
max : N

items : seq T
#items ≤ max

INIT
items = <>

Push
Δ (items)
item? : T
#items ≤ max
items' = <item?> + items

Pop
Δ (items)
item! : T
items ≠ <>
items = <item!> + items'
```

Figure 2. Object-Z specification for simple stack class (from [6]).

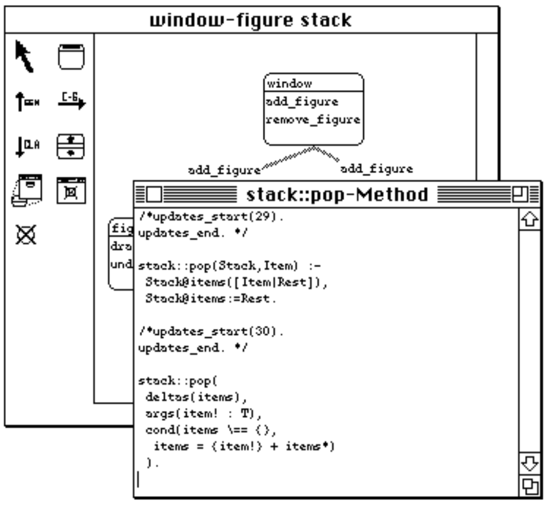
SPE's Object-Z specifications can be split into parts and multiple views of a specification may be provided. For example, the method specifications can be put into their own SPE textual views, or even displayed and edited in the same view as SPE method implementations. Figure 4 shows an example of this. The same can be done with the class interface specification, with constraints on attributes and the initial state of the class defined by an Object-Z schema.

Object-Z views are integrated with all other views of the software under development, so SPE design and analysis views can be used to visualise and navigate through complex Object-Z specifications. SPE provides hyper-links between different views, and thus a designer can move to the specification of a class by double-clicking on a class icon and requesting its Object-Z view(s) be shown. Similarly, a designer can move from an Object-Z view to related design or implementation views. The ability to use the high-level SPE design views to structure class specifications helps to solve to problem of complex, unwieldy specifications noted by [1], [26], and [20].



```
/*updates_start(25).
updates_end. */

class(stack[T],
 types(
  max : N
 ),
 attributes(
  items : seq(T)
   cond(#items =< max)
 ),
 methods(
  init(
   cond(items = {})
  ),
  push(
   deltas(items)
   args(item? : T)
   cond(items* = {item?} + items
  ),
 pop(
   deltas(items)
   args(item! : T)
   cond(items \== {},
    items = {item!} + items*)
  )
 )).
```

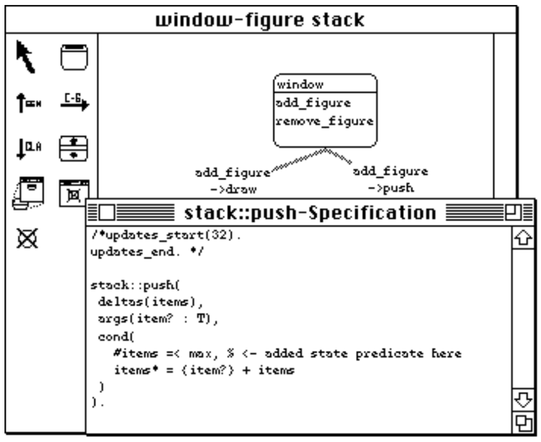
Figure 3. Object-Z formal specification view in SPE.



```
/*updates_start(29).
updates_end. */

stack::pop(Stack,Item) :-
 Stack@items([Item|Rest]),
 Stack@items:=Rest.

/*updates_start(30).
updates_end. */

stack::pop(
 deltas(items),
 args(item! : T),
 cond(items \== {},
  items = {item!} + items*)
 ).
```

Figure 4. Combining Snart code and Object-Z specifications in one view.

Specification refinement can be carried out using Object-Z views. For example, a designer may wish to refine the stack class specification to incorporate the items attribute state predicate into the push and pop operation predicates. The operations will then autonomously respect the state predicate, making translation into code much simpler [17]. The designer can modify the method specifications, either in their separate views, as in figure 4, or in the full class specification view, as in figure 3. If the refinement is carried out in separate views, SPE can automatically update other affected Object-Z views to reflect the change. Figure 5 shows an example of refining the push method specification to incorporate the items attribute's state predicate. SPE's Object-Z views also help to support more complex refinement, such as refining two or more class specifications, by informing the designer of inconsistencies between different views of the specifications.

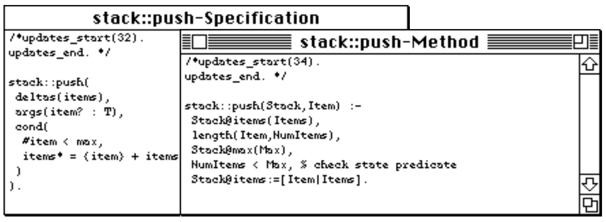


```
/*updates_start(32).
updates_end. */

stack::push(
 deltas(items),
 args(item? : T),
 cond(
   #items =< max, % <- added state predicate here
   items* = {item?} + items
 )
).
```

Figure 5. State predicate refinement into operation predicates.

4. Integrated Software Development

Programmers translate formal specifications into Snart code to produce an implementation. Snart code is translated from the Object-Z views in a manner similar to the proposals of [17] for C++ code generation. It is somewhat easier to generate Snart code, however, as Snart is a logic programming language. When generating C++ code, extra code must be used to test and set the success of invariants, predicates and operations. This is generally unnecessary with Snart implementations, as Snart methods are Prolog predicates and hence may fail, indicating failure to meet the specified constraints. In addition, as Snart allows Prolog list handling predicates to be used in the implementation, code requiring lists can be implemented in a similar style to formal specifications using list operators.



```
/*updates_start(32).
updates_end. */

stack::push(
 deltas(items),
 args(item? : T),
 cond(
  #item < max,
  items* = {item} + items
 )
).
```

```
/*updates_start(34).
updates_end. */

stack::push(Stack,Item) :-
 Stack@items(Items),
 length(Item,NumItems),
 Stack@max(Max),
 NumItems < Max, % check state predicate
 Stack@items:=[Item|Items].
```

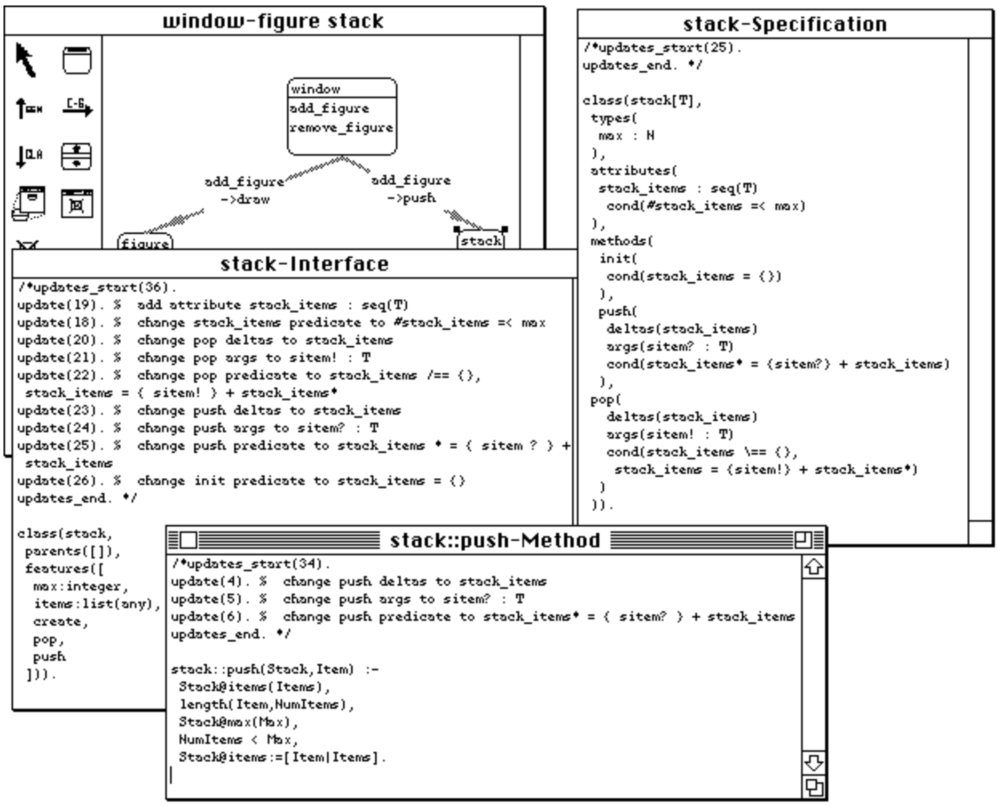
Figure 6. Snart implementation of stack::push.

Figure 6 shows a simple implementation of the push operation using Snart. First, the items state predicate is tested by retrieving the current contents of the stack (a Prolog list), computing its length, and checking it is less than the maximum allowable length. If this number is exceeded (ie the state predicate is not satisfied), the push method fails, indicating to the calling method that the stack is unable to accept any further items. If the stack can accept more items, the items list attribute is updated by prepending the new item to it. This implementation could be done in a number of different ways, and still be consistent with the formal specification. For example, a separate list object could be used to store the stack items, with push calling the list prepend and num_items methods.Typically, code generation from formal specification tools is done after the formal specification is complete. However, as with other approaches to partial system generation, this greatly lessens the ability of designers to evolve the design and specification of a software system along with its implementation. This is because changing a specification means regenerating the system implementation. This new system will not include any code refinements made to the previous system. Object-oriented software development is particularly amenable to evolutionary development [16]. Thus we feel it is essential that once Object-Z views and Snart code views have been defined they must be kept consistent during further evolution of the specification or the code.

SPE maintains consistency between its analysis, design and implementation views. Keeping Object-Z specifications consistent with the design and implementation views presents new problems. Some updates to design and implementation views can quite easily be automatically applied to Object-Z views by SPE, and vice versa. For example, the addition, deletion, and renaming of classes, attributes and services is straightforward to keep consistent, no matter which kind of view is updated. Adding, modifying, or deleting method arguments or their types is also straightforward. Other straightforward translations are described in [17]. This describes techniques for translating from Object-Z to C++, although many of these are straightforward to adapt to Snart or SPE's graphical notation.

Figure 7 shows an example of this type of consistency management for Object-Z views. The items attribute has been renamed to stack_items, and the argument to push and pop has been renamed to sitem. The method deltas, arguments and predicates have all been updated to reflect these changes. In the same manner as for other textual views, SPE expands change descriptions into the header section of the object views to inform programmers of the changes made in other views. Programmers can select the changes they want SPE to apply, and SPE can update the Object-Z specifications. For example, SPE can automatically rename the stack_items attribute in the ‘stack-Interface’ view. The other changes serve as documentation for programmers informing them of specification changes which may impact on method implementation code. For example, the programmer can see in the ‘stack::push-Method’ view that the items attribute should be renamed to stack_items in the method implementation code.

These changes are relatively easy for SPE to implement, as the Object-Z specifications or Snart code can be parsed and the changes incrementally unparsed into the view text (see [11] for details of this process). Our Object-Z syntax allows SPE to incrementally parse a view and locate information needing to be updated. SPE can then replace, for example, old argument names with their new names. A similar approach works for adding or deleting attributes, methods and method arguments. For example, if the programmer renamed the items attribute and push and pop methods in the class interface view, SPE can then automatically rename the attributes and methods in the specification.



```
/*updates_start(36).
update(19). %  add attribute stack_items : seq(T)
update(18). %  change stack_items predicate to #stack_items =< max
update(20). %  change pop deltas to stack_items
update(21). %  change pop args to sitem! : T
update(22). %  change pop predicate to stack_items /== {},
 stack_items = { sitem! } + stack_items*
update(23). %  change push deltas to stack_items
update(24). %  change push args to sitem? : T
update(25). %  change push predicate to stack_items * = { sitem ? } +
 stack_items
update(26). %  change init predicate to stack_items = {}
updates_end. */

class(stack,
 parents([]),
 features([
  max:integer,
  items:list(any),
  create,
  pop,
  push
 ])).
```

```
/*updates_start(25).
updates_end. */

class(stack[T],
 types(
  max : N
 ),
 attributes(
  stack_items : seq(T)
   cond(#stack_items =< max)
 ),
 methods(
  init(
   cond(stack_items = {})
  ),
  push(
   deltas(stack_items)
   args(sitem? : T)
   cond(stack_items* = {sitem?} + stack_items)
  ),
 pop(
   deltas(stack_items)
   args(sitem! : T)
   cond(stack_items \== {},
    stack_items = {sitem!} + stack_items*)
  )
 )).
```

```
/*updates_start(34).
update(4). %  change push deltas to stack_items
update(5). %  change push args to sitem? : T
update(6). %  change push predicate to stack_items* = { sitem? } + stack_items
updates_end. */

stack::push(Stack,Item) :-
 Stack@items(Items),
 length(Item,NumItems),
 Stack@max(Max),
 NumItems < Max,
 Stack@items:=[Item|Items].
```

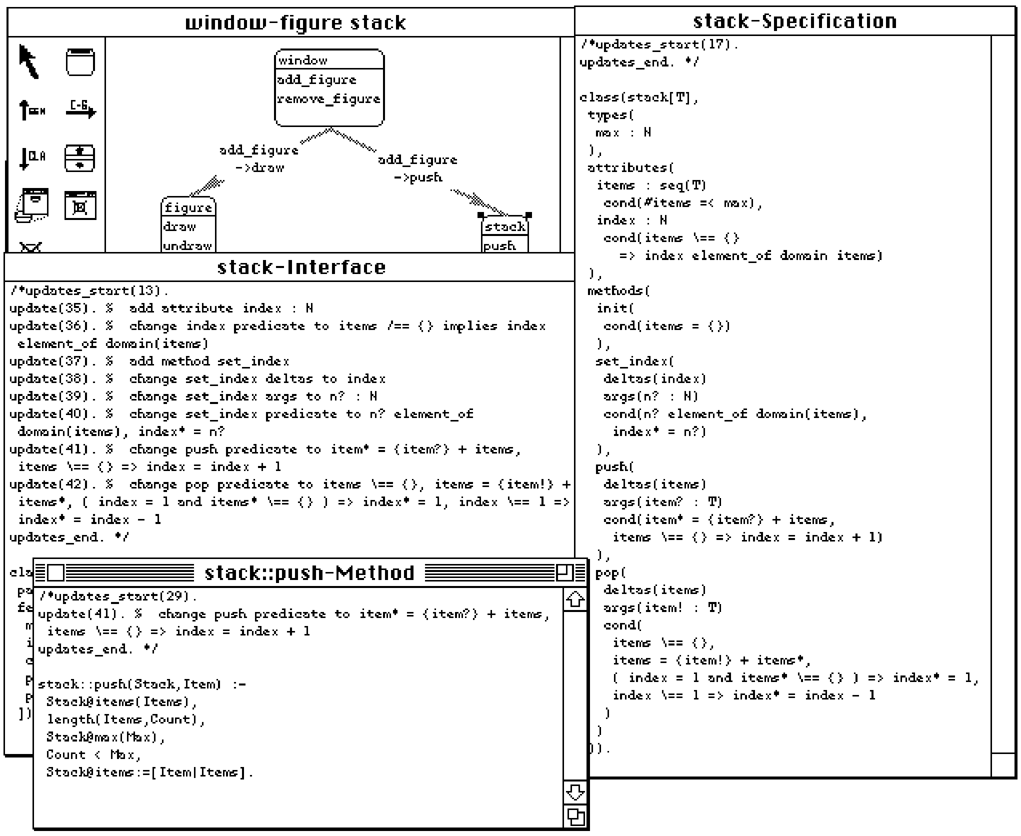
Figure 7. Simple consistency management between SPE and Object-Z views.

If Snart method code or Object-Z state or operation predicates are updated, however, consistency management becomes considerably more difficult. SPE can no longer automatically update code and specification views to keep them consistent and user intervention is required. Similarly, if a new code view statement is added, the affect on the formal specification view cannot be automatically deduced. In these cases the SPE partial consistency approach is taken and change description annotations are used to inform programmers of changes to other views. Many items of interest can be detected when parsing the Object-Z views and code views. These include changes to the delta list for a method specification, changes to predicates, and changes to the set of inter-object operations or method calls made by a specification or implementation view.

This last kind of change is potentially the most useful: if SPE detects that an implementation method calls different methods after being parsed, its likely that the formal specification view will need to be checked against the new method code. Similarly, if the formal specification view is updated and the operations invoked or arguments to these operations have changed, SPE can expand change descriptions specifying this new data into the method implementation view of the operation.

As an example, consider extending the definition of a stack to include an index attribute, acting as a marker to a distinguished element of the stack (once again adapted from [6]). We have updated the formal view in Figure 8 to specify the new state and behaviour of stack objects. In this example, the stack specification now includes the new index attribute and a state invariant for this attribute. The set_index method is used to modify this value, and the push and pop operations have been updated to ensure the index value is updated if items are pushed onto or removed from the stack.

Figure 8 also illustrates the change descriptions presented to the programmer in the stack implementation views. Note that SPE only displays change descriptions relevant to the implementation view item. Some can be automatically applied, such as adding the new index attribute. Others must be implemented by the programmer; the change descriptions serving to inform the programmer such updates are required. This simple example illustrates how SPE supports the evolution of a formal specification and its implementation within an integrated environment.

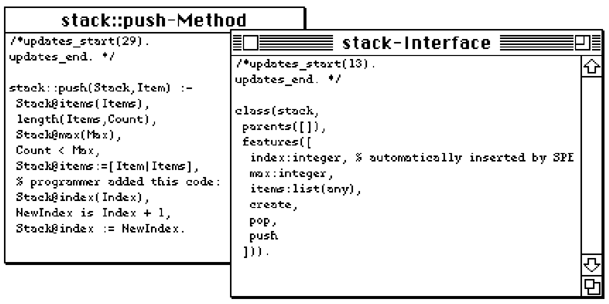


stack-Interface
```
/*updates_start(13).
update(35). %  add attribute index : N
update(36). %  change index predicate to items /== {} implies index
 element_of domain(items)
update(37). %  add method set_index
update(38). %  change set_index deltas to index
update(39). %  change set_index args to n? : N
update(40). %  change set_index predicate to n? element_of
 domain(items), index* = n?
update(41). %  change push predicate to item* = {item?} + items,
 items \== {} => index = index + 1
update(42). %  change pop predicate to items \== {}, items = {item!} +
 items*, ( index = 1 and items* \== {} ) => index* = 1, index \== 1 =>
 index* = index - 1
updates_end. */
```

stack::push-Method
```
/*updates_start(29).
update(41). %  change push predicate to item* = {item?} + items,
 items \== {} => index = index + 1
updates_end. */

stack::push(Stack,Item) :-
 Stack@items(Items),
 length(Items,Count),
 Stack@max(Max),
 Count < Max,
 Stack@items:=[Item|Items].
```

stack-Specification
```
/*updates_start(17).
updates_end. */

class(stack[T],
 types(
  max : N
 ),
 attributes(
  items : seq(T)
   cond(#items =< max),
  index : N
   cond(items \== {}
     => index element_of domain items)
 ),
 methods(
  init(
   cond(items = {})
  ),
  set_index(
   deltas(index)
   args(n? : N)
   cond(n? element_of domain(items),
    index* = n?)
  ),
  push(
   deltas(items)
   args(item? : T)
   cond(item* = {item?} + items,
    items \== {} => index = index + 1)
  ),
  pop(
   deltas(items)
   args(item! : T)
   cond(
    items \== {},
    items = {item!} + items*,
    ( index = 1 and items* \== {} ) => index* = 1,
    index \== 1 => index* = index - 1
   )
  )
)).
```

Figure 8. Harder view consistency when operation predicates updated.



stack::push-Method
```
/*updates_start(29).
updates_end. */

stack::push(Stack,Item) :-
 Stack@items(Items),
 length(Items,Count),
 Stack@max(Max),
 Count < Max,
 Stack@items:=[Item|Items],
 % programmer added this code:
 Stack@index(Index),
 NewIndex is Index + 1,
 Stack@index := NewIndex.
```

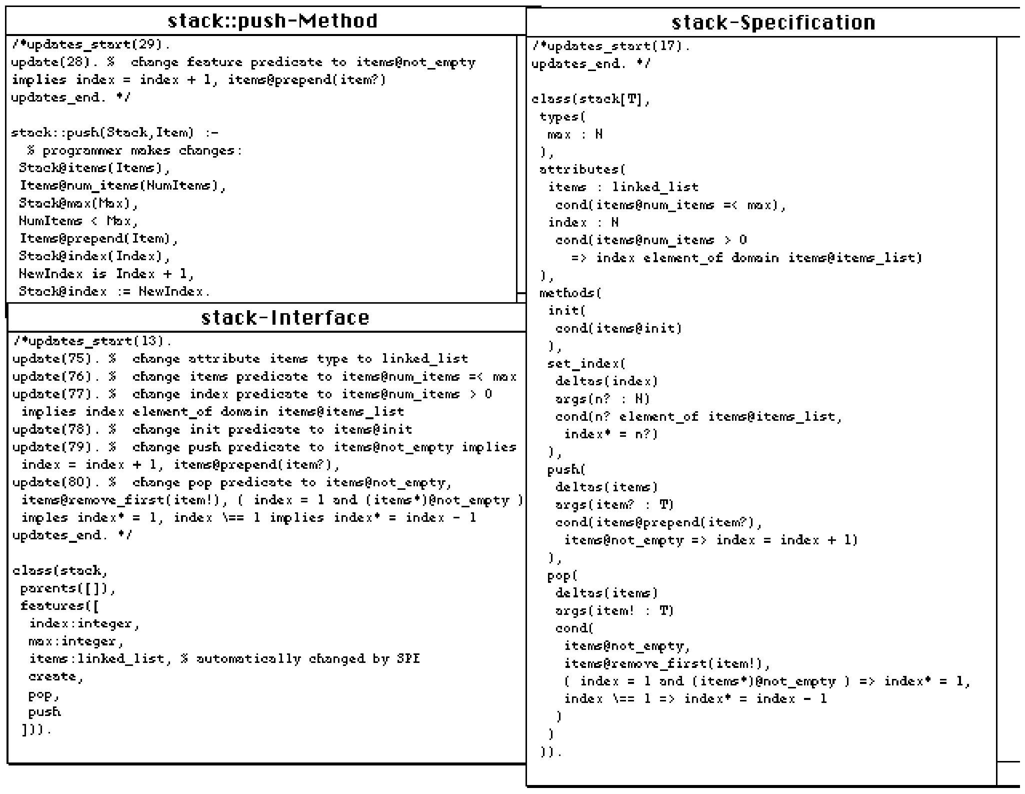
stack-Interface
```
/*updates_start(13).
updates_end. */

class(stack,
 parents([]),
 features([
  index:integer, % automatically inserted by SPE
  max:integer,
  items:list(any),
  create,
  pop,
  push
 ])).
```

Figure 9. Updated implementation views.

Figure 9 shows the modifications SPE and the programmer have made to the class definition and push method implementation views to accommodate the change in the formal specification views. The change descriptions have been removed as they are no longer required. They are, however, still retained by SPE, and can be browsed at any time by examining the modification history of components in a dialog [10].

This approach also works well when refining two or more class specifications together, or when updating the specification or implementation of two or more cooperating classes. For example, consider the example in figure 10, where the stack items list is now implemented using a linked list object. Inter-object operation application is denoted here by the '@' operator, which corresponds to the usual Object-Z '.' operator for operation application. If the formal specification is updated to reflect push using list operations, this can also be reflected as change descriptions in the push method implementation view. The programmer must modify the implementation view manually to achieve consistency between the formal specification and implementation. Note that if the implementation view was updated first, appropriate change descriptions are inserted into the formal specification view, thus view consistency is bi-directional between specification and code.

Other changes made to specification views which are only reflected as change descriptions include: class history predicates (which use temporal logic); the '•' (subsequent operation application) and '||' (parallel operation application) operators; and visibility information (not currently supported by the Snart language).



```
/*updates_start(29).
update(28). %  change feature predicate to items@not_empty
implies index = index + 1, items@prepend(item?)
updates_end. */

stack::push(Stack,Item) :-
  % programmer makes changes:
 Stack@items(Items),
 Items@num_items(NumItems),
 Stack@max(Max),
 NumItems < Max,
 Items@prepend(Item),
 Stack@index(Index),
 NewIndex is Index + 1,
 Stack@index := NewIndex.
```

```
/*updates_start(13).
update(75). %  change attribute items type to linked_list
update(76). %  change items predicate to items@num_items =< max
update(77). %  change index predicate to items@num_items > 0
 implies index element_of domain items@items_list
update(78). %  change init predicate to items@init
update(79). %  change push predicate to items@not_empty implies
 index = index + 1, items@prepend(item?),
update(80). %  change pop predicate to items@not_empty,
 items@remove_first(item!), ( index = 1 and (items*)@not_empty )
 imples index* = 1, index \== 1 implies index* = index - 1
updates_end. */

class(stack,
 parents([]),
 features([
  index:integer,
  max:integer,
  items:linked_list, % automatically changed by SPE
  create,
  pop,
  push
 ])).
```

```
/*updates_start(17).
updates_end. */

class(stack[T],
 types(
  max : N
 ),
 attributes(
  items : linked_list
   cond(items@num_items =< max),
  index : N
   cond(items@num_items > 0
     => index element_of domain items@items_list)
 ),
 methods(
  init(
   cond(items@init)
  ),
  set_index(
   deltas(index)
   args(n? : N)
   cond(n? element_of items@items_list,
    index* = n?)
  ),
  push(
   deltas(items)
   args(item? : T)
   cond(items@prepend(item?),
    items@not_empty => index = index + 1)
  ),
  pop(
   deltas(items)
   args(item! : T)
   cond(
    items@not_empty,
    items@remove_first(item!),
    ( index = 1 and (items*)@not_empty ) => index* = 1,
    index \== 1 => index* = index - 1
   )
  )
 )).
```

Figure 10. Inter-object operation application and consistency with implementation code.

5. Architecture and Implementation

SPE is implemented as a collection of classes, specialised from the MViews framework [9, 11]. MViews supports the construction of Integrated Software Development Environments (ISDEs) by providing a general model for defining software system data structures and tool views, with a flexible mechanism for propagating changes between software components, views and distinct software development tools.

In addition to SPE, the authors have developed several other ISDEs using MViews. MViewsER provides integrated Entity-Relationship diagrams and textual relational schema. MViewsER has been integrated with SPE to produce OOEER, an integrated environment for OOA/D and EER modelling [12]. MViewsDP provides a graphical drag-and-drop interface builder for dialog boxes, with the dialog interface and validation rules being defined in textual views [13]. Cerno-II [8] is a graphical debugger complementing SPE for visualising a running Snart program. EPE is an environment for constructing EXPRESS specifications and corresponding EXPRESS-G diagrams [2]. C-SPE and C-MViews provide support for collaborative, integrated software development via synchronous, semi-synchronous and asynchronous editing [14].

MViews supports the construction of Integrated Software Development Environments (ISDEs) by providing a general model for defining software system data structures and tool views, with a flexible mechanism for propagating changes between software components, views and distinct software development tools. MViews provides much more flexible view consistency mechanisms than comparable ISDEs, such as PECAN [22], Dora [21], and FIELD [23].

MViews describes ISDE data as *components* with *attributes*, linked by a variety of *relationships*. Multiple views are supported by representing each view as a graph linked to the base software system graph structure. Each view is rendered and edited in either a graphical or textual form. Distinct environment tools can be interfaced at the view level (as editors), via external view translators, or multiple base layers may be connected via inter-view relationships.

When a software or view component is updated, a change description is generated. This is of the form `UpdateKind(UpdatedComponent, ...UpdateKind-specific Values...)`. For example, an attribute update on Comp1 of attribute Name is represented as: update(`Comp1,Name,OldValue,NewValue`). All basic graph editing operations generate change descriptions and pass them to the propagation system. Change descriptions are propagated to all related components that are dependent upon the updated component's state. Dependents interpret these change descriptions and possibly modify their own state, producing further change descriptions. This change description mechanism supports a diverse range of software development environment facilities, including semantic attribute recalculation, multiple views of a component, flexible, bi-directional textual and graphical view consistency management, a generic undo/redo mechanism, and component "modification history" information.

New software components and editing tools are constructed by reusing abstractions provided by an object-oriented framework. ISDE developers specialise MViews classes to define software components, views and editing tools to produce the new environment. A persistent object store is used to store component and view data.

The Object-Z views have been implemented by defining new view and view component classes, and extending existing base class and feature (method) classes. All of the consistency management facilities (generation, propagation and rendering of change descriptions) are provided by the MViews framework, requiring no effort by the environment implementor. The main work involved in providing these views is in writing a parser for the Object-Z notation used and in specifying character substitutions for automatic update of the Object-Z view code for certain change descriptions (class/feature rename, attribute retyping, etc.). The Object-Z view parser is written using Definite Clause Grammars (DCGs), while the character substitutions use a regular expression grammar interpreted by MViews. The extensions to SPE to incorporate integrated Object-Z views took less than a person week to implement.

6. Conclusions and Future Research

It is well recognised that most software development lacks rigour, resulting in unreliable products. While there has been much research into formal specification methods to solve this problem, a lack of tools make these formal techniques inaccessible to software practitioners. Our extensions to SPE to provide integrated Object-Z views go some way to achieving an integration of formal specification with less formal design and implementation.

Our integrated environment supports both analysis, design and implementation of object-oriented Snart programs and formal Object-Z specification views for these programs. All of these views can be refined together within an integrated environment, with the environment helping to keep them consistent, either by automatically updating views or displaying change descriptions to assist programmers in identifying where updates are required. The formal specifications are made more accessible, as they can be browsed and navigated between using graphical design views, and can even be displayed in the same view with implementation-level class interfaces and method implementations.

There are, however, still problems with translation to and from the abstract formal specifications and the detailed implementation code. There is a need for more abstract, user-defined links between specifications and code. This would allow, for example, one Object-Z class/operation to be refined to an implementation using several Snart methods, possibly spread over several classes. We are extending MViews to support user-definable links between views, and also adding informal requirements (textual) views with user defined links to specifications and code. This provides a finer grained relationship than does the existing propagation of changes to textual documentation views.

We are extending SPE's support for Object-Z views to provide improved support for automatic update of Snart implementation views. Providing facilities to filter out changes which relate to particular methods or attributes would eliminate spurious change descriptions which do not affect the formal specification. We plan to add a facility to SPE to generate $\LaTeX$ files from the Object-Z views, which will use the standard oz.sty $\LaTeX$ macros to format the Object-Z specifications. This will allow high-quality documentation to be generated.

We have recently designed an extension to SPE which supports the refinement calculus, rather than Object-Z specifications. Proof obligation views are used to ensure a refinement is proveably correct. Programmers choose direct translation between refinements, where this is possible, or choose different translations, where there are ambiguities. It is possible to present the programmer with a list of possible provably correct alternatives and allow the programmer to carry out the refinement they require. SPE could be extended in power by turning more of the latter into either of the former. SPE's current support for specification/implementation consistency is a simple start with a small set of straightforward translations and a larger set of vague mappings. The partial consistency change description approach of MViews makes it possible to refine this environment incrementally.

References

[1] Alencar, A.J. and Goguen, J.A., "OOZE: an object-oriented Z environment," in *ECOOP 91, LNCS 512,* Springer-Verlag, 1991, pp. 180-191.

[2] Amor, R., Augenbroe, G., Hosking, J.G., Rombouts, W., and Grundy, J.C., "Directions in modelling environments," to appear in *Automation in Construction.*

[3] Booch, G., *Object-Oriented Design with Applications.* Benjamin/Cummings, Menlo Park, CA, 1991.

[4] Casais, E., Lewerentz, C., Lindner, T., and Weber, F., "Formal Methods and Object-Orientation," in *Tutorial at TOOLS Europe 93,* 1993.

[5] Coad, P. and Yourdon, E., *Object-Oriented Analysis*, Second Edition. Yourdon Press, 1991.

[6] Duke, R., King, P., Rose, G., and Smith, G., "The Object-Z Specification Language Version 1," Technical Report TR 91-1, SVRC, University of Queensland, 1991.

[7] Durr, E.H.H. and van Katwijk, J., "VDM++: a formal specification language for object-oriented designs," in *Proc TOOLS 7,* Prentice-Hall, 1992, pp. 63-78.

[8] Fenwick, S., Hosking, J.G., and Mugridge, W.B., "Visual debugging of object-oriented systems," in *Proceedings of TOOLS Pacific 94,* 1994.

[9] Grundy, J.C. and Hosking, J.G., "A framework for building visusal programming environments," in *Proceedings of the 1993 IEEE Symposium on Visual Languages,* IEEE Computer Society Press, 1993, pp. 220-224.

[10] Grundy, J.C., Hosking, J.G., Fenwick, S., and Mugridge, W.B., Connecting the pieces, Chapter 11 in *Visual Object-Oriented Programming*. Burnett, M., Goldberg, A., Lewis, T. Eds, Manning/Prentice-Hall, 1995.

[11] Grundy, J.C. and Hosking, J.G., "Constructing Integrated Software Development Environments with Dependency Graphs," Working Paper, Department of Computer Science, University of Waikato, 1994.

[12] Grundy, J.C. and Venable, J.R., "Providing Integrated Support for Multiple Development Notations," in *Proceedings of CAiSE'95,* Finland, June 1995, LNCS 932, Springer-Verlag, pp. 255-268.

[13] Grundy, J.C., Hosking, J.G., and Mugridge, W.B., "Supporting flexible consistency management via discrete change description propagation," Working Paper, Department of Computer Science, University of Waikato, 1995.

[14] Grundy, J.C., Mugridge, W.B., Hosking, J.G., and Amor, R., "Support for Collaborative, Integrated Software Development," in *Proceeding of the 7th Conference on Software Engineering Environments,* IEEE CS Press, April 5-7 1995, pp. 84-94.

[15] Hedlund, M., "The integration of LOTOS with an object-oriented development method," in *Proc FME'93, LNCS 670,* Springer-Verlag, 1993, pp. 73-82.

[16] Henderson-Sellers, B. and Edwards, J.M., "The Object-Oriented Systems Life Cycle," *Communications of the ACM*, vol. 33, no. 9, 142-159, 1990.

[17] Johnston, W. and Rose, G., "Guidelines for the manual conversion of Object-Z to C++," Technical Report, SVRC, University of Queensland, 1993.

[18] Lano, K., "An object-oriented extension to Z," in Z User Workshop: Proceedings of the Fourth Annual Z User Meeting, Oxford, Springer-Verlag, 1991, pp. 151-172.

[19] Morerira, A.M.D. and Clark, R.G., "Object-oriented analysis and formal description techniques," in *Proc ECOOP'94, LNCS 821,* Springer-Verlag, 1994, pp. 344-364.

[20] Nilsson, G. and Blysa, P., "Tool for object-oriented formal specification technique," in *Proc TOOLS 7,* Prentice Hall, 1992.

[21] Ratcliffe, M., Wang, C., Gautier, R.J., and Whittle, B.R., "Dora - a structure oriented environment generator," *IEE Software Engineering Journal*, vol. 7, no. 3, 184-190, 1992.

[22] Reiss, S.P., "PECAN: Program Development Systems that Support Multiple Views," *IEEE Transactions on Software Engineering*, vol. 11, no. 3, 276-285, 1985.

[23] Reiss, S.P., "Connecting Tools Using Message Passing in the Field Environment," *IEEE Software*, vol. 7, no. 7, 57-66, July 1990.

[24] Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., and Lorenson, W., *Object-Oriented Modeling and Design*. New Jersey, Prentice Hall, 1991.

[25] Shlaer, S. and Mellor, S., *Object-Oriented Systems Analysis: Modelling the World in Data.* Englewood Cliffs, New Jersey, Yourdon Press, 1988.

[26] Welsh, J., Broom, B., and Kiong, D., "A Design Rationale for a Language-based Editor," *Software - Practice and Experience*, vol. 21, no. 9, 923-948, 1991.