SMG - State Machine Generator

Kevin Quick
quick@null.net

Version 1.7
16 Apr 2002

Abstract

The SMG utility can be used to scan an input file for specific directives that describe a State Machine (States, Events, Transitions, and associated Code segments) and generate several different outputs: C code to implement that State Machine, Promela code to implement a formal verification of the State Machine using Spin, and a graphical representation of that State Machine for analytical purposes. In mechanical terms, the SMG may be thought of as a specific-purpose C preprocessor.

The SMG utility may be obtained from: `http://smg.sf.net/`

Contents

1 Background

1.1 Tunnel Vision

Modern programming techniques are often oriented around a procedural language such as C or a similarly structured object-oriented implementation. When developing software using these languages, the prevalent coding methodology is to design a set of instructions and functions that will perform the intended sequence of tasks, using a top-down approach. This sequence of tasks is identified as the primary objective of the program being developed. This programming methodology is referred to as **task-oriented** programming.

Frequently, however, the attention to developing support for portions of the code not related to the primary objective is diminished or in some cases nonexistent. This results in unhandled or badly handled error paths and an inflexibility of the code regarding changes or deviations in input or operating environment. The top-down approach can exacerbate this issue as the focus narrows as details are introduced, thereby causing those details to be developed with primarily local concerns.

Additionally complicating this programming methodology is the attempt to introduce multi-threaded scheduling to improve the overall responsiveness of the program to multiple simultaneous inputs. When introducing threading significant attention must be provided to properly protecting common data structures and maintaining each thread relative to all other threads and the overall program state.

Finally, validation of the resulting code is left to a rote process of verifying functionality and output of the code when presented with sample sets of input; coverage tools attempt to verify that most of the written code has been executed, but this does not necessarily mean that the full range of input or sequencing has been tested in the process. A technique known as modelling and formal verification can be used to abstract the functionality of the code, but this is usually only used in the architectural and early design phases; the resemblance of the final code to the formal verification model is tenuous at best and often minimal. Developers tend to handle unexpected input or events as "bugs" by providing local fixes to ensure that their code does not fail when these occur, but without consideration of how the system as a whole should properly handle those situations and rarely by returning to and updating the formal verification model.

1.2 Asynchronous Event Handling

Another one of the frequent challenges in developing computer software is implementing and maintaining code that can fully manage an operation that is subject to asynchronous events. Asynchronous events can take many forms, including: user-input, operation completion notifications, interrupts, and operation requests. In general, an asynchronous event can occur at any time and the software must be capable of determining what the appropriate response to that event is at that point in time.

These event-oriented environments are found in most non-computation oriented code and are especially prevalent in: Internet Servers, Device Drivers, and GUI implementations.

The task-oriented coding style discussed previously makes it difficult to anticipate and handle exceptions to the sequence of tasks which are introduced by Asynchronous Events and also leads to assumptions regarding when events will occur and what types of events will occur at various delivery points. Often ignored or lightly-addressed is the importance of determining the appropriate response to *all* types of events that may occur at *each* point where they may occur.

One technique commonly used in applications written for an Asynchronous Event environment is to declare an event queue onto which all events are placed in order of occurrence. The code then removes an event from the head of the queue, processes that event in a task-oriented manner until the event has been completely handled, and then returns to the queue to obtain the next event. This algorithm may be useful in some circumstances but its appropriateness is often invalid when events cannot be queued, when events must be prioritized, and when some events may interrupt other events.

1.2.1 EDSM Methodology

An alternative coding style used in these situations is the **Event-Driven State Machine** or **EDSM** methodology[1]. In this methodology, events are typically delivered to a common entry point and then a specific function or switch statement is invoked based on the current state or the event. Once in that code has been invoked, the other parameter (entry or state) is examined to select the code to execute for that state/event combination. While this style is more flexible in terms of handling unexpected events it is more arduous to develop code in this style due to the mechanics of reproducing state and event selection code throughout the code and the loss of the "flow" perspective for the primary sequence of tasks.

Some software designs do implement an EDSM methodology, but unfortunately, most of the software in these implementations focuses on the expected sequence of operations only; it's frequently the case that not all events are handled at each event-delivery juncture in the code.

Furthermore, in situations where state machines are implemented in the software, the completeness of the state machine tends to logarithmically increase the complexity and obscurity of that state machine, making them hard to understand and maintain. Each possible state has to consider the potential occurrence of an ever-increasing number of events, and the addition of each state causes an event-number of new paths through the state machine.

This type of code is difficult to maintain, especially for developers introduced to the code after it is written. State machine code is often spread throughout the body of the main code, making it hard to understand the entirety of the state machine. Furthermore, the state machine code has significant side effects; any change to a state machine's structure (i.e. adding a new state or changing the response to an event) will significantly impact code executing to handle future events. Understanding, predicting, and assessing the validity of any changes to the state machine quickly becomes a monumental task as the size of the state machine grows.

A formal verification model is a highly useful tool to properly evaluate and manage additions (states or events) to this type of code, but again, since the formal verification model is often retired or divergent at this point in development the opportunity is not often available. Some institutions even separate the coding and formal verification modelling into separate groups, the former being handled by architects and the latter by developers. Sometimes the developers are not even aware of the existence of formal verification models.

1.3 Non-blocking Operations

Another more recent advancement facing many implementations is the introduction of non-blocking function calls for performing various asynchronous tasks such as: I/O operations and remote procedure calls (RPC) found in distributed environments. In these situations, the program's initial request is not completed when the request call completes; instead the request is processed in parallel or at some later point in time and when completed, the initial program is notified of that completion (usually by invocation of a callback function).

This complicates the normally task-oriented methodology in the following ways:

1. The task-oriented flow is broken up into several different "chunks" of code, split by the need to await a completion indication after performing a non-blocking request.

2. The potential for other events to occur while waiting for or instead of the requested operation, with those events being delivered before or instead of the primary operation's completion indication.

3. The potential for request re-entrancy, allowing the program to handle a new request while waiting for an interim non-blocking operation to complete for a previous request. While this tends to increase the overall efficiency of the program, it significantly affects the management of

[1]The State Machine described in the document is a Finite Deterministic Mealy Machine, also referred to as a Deterministic Finite Automata or DFA.

common data and the management of this now-pipelined implementation.

4. The potential for the non-blocking request to signal completion at any point after the request call is made, possibly even before the request call returns to the calling process.

These complications require significant additional care in properly and efficiently implementing applications for this type of environment.

2 The SMG Solution

The proposed approach to handling these issues is to attempt to reconcile the EDSM methodology with the more customary task-oriented programming styles. In order to do this, we seek to automate the mechanics of EDSM and re-introduce the conceptual perspective of task-oriented coding styles. This is done by beginning with the EDSM methodology and making the following set of observations and changes.

To facilitate this approach, the SMG tool has been developed. Using the SMG tool, the developer describes the state machine by a set of SMG directives interspersed with the program's normal C code.

The SMG directives are intended to be more succinct in describing the state machine than the corresponding C code, thereby allowing the state machine to be more easily recognized and understood even at the input specification level. The SMG directives also allow default functionality to be specified and allows the state machine to be more "naturally" described in parallel to the task-oriented code segments.

Once the code (including SMG directives) has been developed, the code is passed through the State Machine Generator (SMG) as a preprocessing stage. The SMG utility converts the SMG directives into C code which implements the described state machine, allowing the result to be passed to the C compiler as a combination of the state machine and the task-oriented code, providing a complete functional solution.

2.0.1 Visual Analysis

The SMG utility also produces a description of the state machine that may be passed to the GraphViz utility to obtain a graphical representation of the state machine. The graphical representation allows easy interpretation and maintenance of the state machine.

The graphical representation shows the various states as nodes on the graphs and labels the arcs that connect those nodes with the names of the events that cause that transition. The arc labels also indicate which code objects are executed before and after the event causes the actual state to change.

Expected transitions are represented by thick arcs, making it easy to follow the normal code flow and differentiated from unusual or error handling transitions.

SMG will also *automatically* generate an error state if there are any states wherein the result of an event is not defined explicitly by SMG directives. The corresponding C code generated will cause a runtime error call to a programmer-supplied error routine if this undefined event transition occurs). All transitions to this error state will be labelled as errors for easy analysis; final versions of the code should not contain any undefined transitions of this type.

SMG will also group states in order to simplify the diagram. For example, if a group of 5 states have normal transitions to other states, but they all transition to an error state when a particular event occurs, SMG will assign those 5 states to a group and separately indicate that the group transitions to the error state on that event. This grouping is performed automatically.

2.0.2 Functional Verification

The third SMG output is a SPIN/Promela model of the state machine. This model is automatically extracted from the states, events, and transitions defined by the SMG directives. It is therefore capable of describing the overall functionality of the system for SPIN functional verification without any additional input from the developer.

To supplement and refine the functional verification process, standard Promela code can be identified in the input file by bracketing SMG directives. The SMG utility will then direct this Promela code into the appropriate locations of the generated model as

a parallel to the C code for the same transitions. In this way, both the overall functionality and detailed modelling code can be maintained in the SMG input file along with the corresponding C code. Thus, when additional states or events must be considered, or when one or more transitions must be redefined, the Promela model is correspondingly adjusted for continued formal verification.

As with the graphical and C outputs, the emitted Promela code generates assertions for undefined transitions, causing them to be flagged during formal verification (or they can be disabled to validate the currently defined subset of the application).

2.1 SMG Development Cycle

SMG development is done by designing C code modules as in a normal development operation and implementing SMG directives in one or more of the modules. Modules containing SMG directives typically have a "`.sm`" suffix to distinguish them from normal "`.c`" files (and so that SMG can generate a `.c` from the `.sm`). The SMG directives can appear exlusively in a file themselves or then can be intermixed throughout the C code to the degree desired by the coding style being used.

The SMG directives also establish the **States** and **Events** that can occur, and optionally any **entry points** into the code that represent Events. The definitions of these States, Events, and entry points are output by SMG into a header include file (`xxx_smdefs.h`). All modules needing to use a State or Event value or make other references to the state machine can include this file to obtain the appropriate external definitions.

Once the C code and SMG directives have been written, the `smg` utility should be run on the `.sm` input files. The graphical state machine output should be examined to visually verify the implementation, followed by formal verification using SPIN and the output Promela model. Once the state machine has been formally verified, the output C code should be compiled by the C compiler and the linker for all C source files (some of which may have been output by the smg utility from `.sm` files). All errors during the development cycle are resolved in the input `.sm` file rather than making modifications to interim files; the resulting SMG model is therefore fully complete and functional.

2.2 C Data Structures and Functions

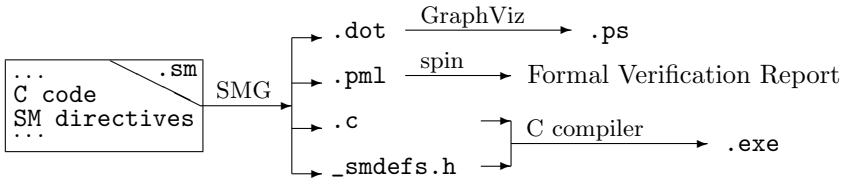
In order to implement the EDSM methodology, SMG attempts to make minimal assumptions about the programming environment in which it will generate the state machine.

SMG will automatically provide C definitions for the states and events described in its directives. There are only two required C data types and based on the external interface for the program, and one required C function (the error handling function). The programmer can optionally define the syntax of event delivery points within the code with additional directives and C declarations, but this is not required. SMG does assume that if the event delivery points are not specified that the C program will provide the event-specific interface and invoke the state machine with an indication of the underlying event type.

The first required C data type must be a structure and is the `SM_OBJ` structure. This is assumed to be an instance of a structure that is global to the current task. This structure may contain anything that the programmer desires to implement but it must also contain the following field: `<SM_NAME>_state_t sm_state;` where `<SM_NAME>` is the declared name of the state machine.[2] It is recommended that the SMG-related code not refer to any global writable data outside of the `SM_OBJ` structure although it is possible to deviate from this under certain circumstances.

The second required C data type is the `SM_EVT` type. This data type is assumed to represent any and all information related to the current event-initiated task. A common implementation is for the programmer to maintain a pool of `SM_EVT` structures which are used one-by-one as primary events arrive. The `SM_EVT` type may be any valid C data type; it is used to maintain the task-oriented context for the series

[2]Multiple state machines may be defined in the same `.sm` file, and they may even share a common `SM_OBJ` data structure; the use of `<SM_NAME>` in all relavent state machine interfaces keeps the state machines distinct and separate.



of secondary events that are initiated by the primary event. The SMG code will not use the `SM_EVT` variable directly but will pass the value of this variable from the event-initiated state machine entry to the selected state/event handling code.[3]

3 SMG Directives Overview

SMG directives are designed for simplicity in both parsing and specification. Although the user can choose to design a preprocessing stage to preceed the SMG operation (*e.g.* m4), SMG directives are simply presented and designed to be unambiguously distinguishable from the C code into which they are placed.

3.1 Common SMG Directives

The list of directives allows for significant flexibility in how the state machine is described and where the various operations occur. Some of these capabilities are only needed for special situations, and some are dependent on the way in which the state machine is implemented. Only the most basic directives are needed to identify a state machine and all other directives may be introduced only as needed when the basic directives are insufficient. The basic directives are listed below:

```
SM_NAME   TRANS
SM_OBJ    SM_EVT
CODE      CODE_{   CODE_}
```

The reader is advised to attend to this subset of directives initially and to review the additional keywords presented in the SMG Guide once basic familiarity with SMG is attained.

[3]Because the `SM_EVT` variable is not actually used by SMG-generated code, the associated variable may be unused throughout but it still must be a valid data type.

3.2 SMG Directive Keywords

This section describes *all* supported SMG keywords. As noted previously, the initial review should focus on the subset of common keywords, leaving the remaining keywords for more esoteric evaluation.

The syntax of SMG Directives is described in Figure 1. In the figure, `<xxx>` represents a user-supplied word, and `<xxx>...` represents one or more user-supplied words. Square braces ('[' and ']') enclose optional portions of the directive and are not actually present in the directive. There are no restrictions on `<xxx>` words except those noted here and excepting whitespace. Some `<xxx>` words must be valid C identifiers: `<state-machine-name>`, `<state-name>`, and `<event-name>`.

All SMG directives are line oriented, must be contained on a single line, and must start with the SMG directive keyword at the first character on the line. The remainder of the line contains the arguments and values for the SMG directive as whitespace separated words, up to the end of the line; SMG directives do not support line-continuation.

`SM_NAME` — Specifies the start of a State Machine of the specified name. An input `.sm` file(set) may contain multiple state machines. The `SM_NAME` identifies the state machine to which all of the following state machine directives apply until a new `SM_NAME` state machine declaration is read.

`SM_DESC` — Descriptive text describing the purpose of the State Machine

`SM_OBJ` — Type of the C object representing the instantiation of the State Machine. This ob-

```
SM_NAME <state-machine-name>

SM_DESC <description>...

SM_OBJ  <C-type>...

SM_EVT  <C-type>...

SM_INCL <filename>

SM_DEF  <context-name>
SM_IF   <context-name>
SM_ELSE <context-name>
SM_END  <context-name>

STATE   <state-name> [ <description>... ]

INIT_STATE <state-name> [ <description>... ]

ST_DESC <description>...

EVENT <event-name> [ <entry-code-tag> <setup-code-tag> [ <description...> ] ]

EV_DESC <description>...

TRANS  <current-state> <event> <new-state> [ [<pre-code>] <post-code> ]
TRANS+ <current-state> <event> <new-state> [ [<pre-code>] <post-code> ]
TRANS= <current-state> <event> <new-state> [ [<pre-code>] <post-code> ]

## <comment>...

CODE    <code-tag> <code>...

CODE_{  <code-tag>
CODE_}

PROMELA_{      <code-tag>|HEADER|INIT
PROMELA_}
```

Figure 1: SMG Directive Syntax

ject persists across the lifetime of the state machine and typically contains user information needed by the code containing the state machine and must include an `sm_state` field used by the SMG-output code to maintain the current state of the code.

`SM_EVT` — Type of the C object associated with an event. All information relative to the current event (except the event code itself) must be contained in this object. The body of the SMG-generated state machine will only have access to the `SM_OBJ` and `SM_EVT` variables.

`SM_INCL` — Includes another file. This operates in a similar manner to the `#include` preprocessor directive in C except that it works during the SMG preprocessing stage rather than the C preprocessing stage. Additionally, the optional second parameter specifies the version of the file to be included. The `SM_INCL` directive will look for the specified file based on a search path. The search path looks in the following locations in the order specified:

1. The current directory

`/usr/local/lib/SMG/*-vN`

2. This location is searched only if the optional second argument is specified on the `SM_INCL` line specifying the version of the interface. The `N` in the above path is replaced with the version number specified in the second argument and the `*` specifies a wildcard of all entries. Thus, if the optional version specified is "23", the `SM_INCL` will search in all of the /usr/local/lib/*-v23 directories.

`/usr/local/lib/SMG/*`

3. All subdirectories of the above are checked.

`/usr/local/lib/SMG`

4.

`SM_DEF` — Defines a directive for use in the `SM_IF` /`SM_ELSE` /`SM_END` statements. The `SM_DEF` is roughly equivalent to a C statement like the directive `#define <context-name>`, except that the SM versions are parsed by SMG and `#defines` are parsed by the C preprocessing stage of the compiler, which follows SMG parsing. One way to negate this is to make it an SMG comment (see below). This is a global directive and is not related to any specific state machine or other context.

`SM_IF` — Specifies the beginning of a block of code that is only included if the `SM_DEF` statement for the corresponding `<context-name>` has been seen. This is a global directive and is not related to any specific state machine or other context. Blocks of code demarcated by `SM_IF` /`SM_END` statements may be nested, even within the same `<context-name>`. Each `SM_END` statement exits the corresponding `SM_IF` context but no others. Context nesting is allowed but scope must be maintained.

`SM_ELSE` — Specifies the beginning of an alternate block of code that is only included if the `SM_DEF` statement for the corresponding `<context-name>` has *not* been seen.

`SM_END` — Specifies the end of a block of statements (C and SMG) that was previously identified with the `SM_IF` statement.

`STATE` — Defines a state and optionally a description of that state. This directive is optional; states are deduced by the SMG from `TRANS` directives as needed.

`INIT_STATE` — Specifies the initialization for a state machine that is initialized at instantiation. There may only be one of these for a state machine although it may appear in place of or with a corresponding `STATE` directive for the same state. State machines instantiated in this way *MUST* have zeroed contents when instantiated; since instantiation is external to the SMG generated code, the developer must insure that the new state machine, however created, is zero'ed (i.e. `memset(sm_obj, 0, sizeof(sm_obj))`) before any events are delivered to that state machine.

State machines which are not initialized at instantiation must be explicitly initialized with the `<name>_State_Machine_Init` operation before any events are delivered to that state machine.

`ST_DESC` — Provides additional description for the most recently declared `STATE` .

`EVENT` — Defines an event, optional entry point declaration, optional input variable preprocessing, and an optional description of the event. As with the `STATE` directive, the `EVENT` directive is optional if no entry point or preprocessing must be defined and events will be deduced as needed.

Most uses of the `EVENT` keyword will specify only the `<event_name>`.

In situations where an event entry point is automatically defined (e.g. an Event Library), the optional `<entry-code-tag>` and `<setup-code-tag>` specify the code to be generated for the event handling entry point. The `<setup-code-tag>` should not be defined (*i.e.* not `--`) unless the `<entry-code-tag>` is also defined (in other words, if `<entry-code-tag>` is "`--`" then `<setup-code-tag>` must also be "`--`").

`EV_DESC` — Provides additional description for the most recently declared `EVENT` .

`TRANS` — Defines a transaction. Specifies the handling of an Event for a specific current state in terms of the new state to go to when the event is received, along with any code to execute either before or after moving to the new state. The `TRANS` statement is the principle specification for SMG input.

The `<new-state>` may be "`--`" to indicate that there is no state change associated with this event in the current state.

The `<pre-code-tag>` and/or the `<post-code-tag>` elements may be "`--`" to indicate that there is no pre-state-change or post-state-change code to execute, respectively.

The `<current-state>` may be "`*`" to indicate that the transaction is a default transaction and applies to all states.

`TRANS+` — A special form of the `TRANS` directive. This directive indicates that the associated transaction information is defining "group" code that is performed in addition to the normal transaction. There must only be one `TRANS` for a current-state/event/destination-state transaction, but there may be zero or more `TRANS+` (in addition to the `TRANS`) for that same transaction that specify additional code. This directive is especially useful for specifying default code that is associated with an event that occurs in any state (using a wildcard specifier for the current-state as described below).

Another way of describing the difference between `TRANS` and `TRANS+` is that a multiply defined state transition error will only be detected for `TRANS` directives.

The `<new-state>`, `<pre-code-tag>`, `<post-code-tag>`, and `<current-state>` fields may have the special values described in the `TRANS` directive and have the same effect.

`TRANS=` — Another special form of the `TRANS` directive. This form is used to indicate the normal or expected transition in the current state (i.e. the primary code path). When the event described in this `TRANS=` directive has corresponding entry point and preprocessing code directives, the SMG preprocessor outputs specific code testing for and implementing this transaction before passing into the more general transaction processing routine, thereby increasing the efficiency and performance of the "common path" code.

The `<new-state>`, `<pre-code-tag>`, `<post-code-tag>`, and `<current-state>` fields may have the special values described in the `TRANS` directive and have the same effect.

`##` — When placed at the start of a line, this specifies that the line is an SMG comment; SMG comments will not be reproduced in the output C code.

`CODE` — Specifies a line of code and its associated tag. All code referenced by `EVENT` and `TRANS` directives is referred to by an associated code tag; the `CODE` directive specifies the actual C code that is associated with that tag.

Within the C code associated with the `CODE` tag, special keywords will be recognized and appropriate substitutions will be made in the output C code generated. All code-internal keywords are of the form "`_/xxx`" and may be one of the following:

`_/OBJ` — Substitute the name of the `SM_OBJ` -typed variable.

`_/EVT` — Substitute the name of the `SM_EVT` -typed variable

`_/NAME` — Substitute the name of the current state machine being defined.

`_/<STATE>` — Specifies that code should be inserted to set the state to the specified value. This is used for situations where the destination state cannot be determined solely from the current state and event. The code which is specified for this type of `TRANS` operation must programmatically make the determination of the appropriate target state and then set that state using the "`_/<STATE>`" keyword.

`_/<EVENT>` — Specifies that the associated event should be delivered to the state machine. This event is delivered *IMMEDIATELY* to the state machine, and is comparable to a recursive invocation of the state machine. It is possible to defer the delivery of these events to the end of the handling for the current event using a command-line flag, in which case they will be delivered in the order they were generated after all tagged code associated with the event has been executed.

If it is desireable to queue events to a state machine instead, for both external event deliveries and internally generated events, a separate mechanism must be provided by the user to implement this queueing. These directives should be used *CAREFULLY*.

`CODE_{` — Specifies the start of a multi-line section of code and its associated tag. This directive is special in that the code associated with this directive spans multiple lines, up to the closure directive. Other than the multi-line aspect, this directive is exactly like the `CODE` directive, including the keyword substitution activities.

`CODE_}` — Specifies the end of a multi-line section of code started by the `CODE_{` directive.

`PROMELA_{` — Specifies the start of a multi-line Promela code section. Promela is a modeling language that may be used with the Spin utility to model and validate a state machine. Promela code is output to the `.pml` file rather than the `.c` and `.h` files for corresponding `CODE` segments.

Special code tags of `HEADER` and `INIT` may be used to specify Promela code that should be output to the header or the end (init) of the Promela file. One common initialization function is to declare the Promela channel used to deliver events to the state machine and then run the state machine, passing that channel. The channel messages should be declared as follows:

Where the first mtype should be either `INITIALIZE_SM` or `SM_EVENT` as appropriate, and the second mtype should be the initial state or the event name, respective to the first mtype. The depth of the channel should always be zero to accurately model the immediate delivery mechanism of the state machine; queued event delivery should use a separate queueing process rather than simply expanding the channel depth to provide the proper semantics.

See the output of the examples for more information.

`PROMELA_}` — Specifies the end of a multi-line Promela code section.

3.3 SMG Wildcards

When specifying the arguments for the SMG directives described above, there are special wildcards that are recognized by the SMG preprocessor that may be used in place of a more customary value for that argument:

`*` — Wildcard argument. This is typically useable in place of a state name in a `TRANS` directive, indicating that any and all states apply for that transition. This can be useful in specifying the "default" transition for an event, and may be overridden for specific states by subsequent `TRANS` directives that do not use the wildcard.

`--` — No-action argument. This indicates that nothing should occur for the corresponding argument. For example, when used in place of the `new_state` name in the `TRANS` statement, this indicates that the event does not change the current state. When used in place of a code tag, it indicates that there is no code to be executed.

3.4 SMG Syntax Version

Currently the SMG directives syntax is unchanged from the original publicly distributed version. In the future, the SMG directives syntax might need to change to accomodate additions, changes, and removals from the current syntax. To allow `.sm` files to be written to conform to multiple syntax forms and/or validate the needed syntax to interpret the current file, SMG automatically generates one or more context definitions describing the syntax interpretation used by that `smg` utility.

This automatically generated context definition may be checked with the `SM_IF` directive, even though no `SM_DEF` directive explicitly defined that context.

Syntax extensions and additions are represented by additional automatically generated contexts defined simultaneously with the original syntax definition where the new syntax is backward-compatible with the original syntax. Syntactic changes incompatible with previous versions will be represented by a newly unique automatic context definition and the lack of previous definitions.

Also note that the syntax versioning supported in this manner is independent of the SMG utility version; multiple SMG utility versions might support identical syntax forms.

Context	Description
`SMG_SYNTAX_A`	Original SMG syntax

4 SMG Specification Requirements

In addition to the SMG directives described above, there are a few requirements for successfully integrating the generated state machine C code into the rest of the software module:

1. The `SM_OBJ` structure must contain a field (`sm_state`) that can be used by SMG to maintain the current state of the state machine. This field should NEVER be directly accessed by the C code.

2. When a run-time error occurs, the generated state machine C code will call an error function that must be supplied by the user-supplied C code; this is so that the error handling activity can be handled in a manner appropriate to the current implementation.

   The declaration for the error function to be provided by the user-supplied C code is defined as the `<SM_NAME>_State_Machine_Error` function in Figure 2.

   User-supplied code may also call the error function, but if the `err_id` used matches the errors defined in Table 1 then the `errtext` and additional parameters must additionally match.

   In the definition referenced, `<SM_NAME>` is replaced with the name of the state machine associated with this error (allowing separate error handlers for each state machine defined) and `<SM_OBJ>` and `<SM_EVT>` are replaced by the corresponding C type specifications from the similarly named SMG directives.

The `errtext` describes the error and may be followed by arguments to be used in `printf`-style format codes.

The `err_id` is an identifier value associated with this error. The `errtext` and the type and sequence of arguments for a specific `err_id` will never change, so error routines are free to key on the `err_id` value to perform specific actions as defined in Table 1.

3. When C code within the user-supplied software wishes to deliver an event to the state machine and therefore activate it to process that event to completion, it should call the `<SM_NAME>_State_Machine_Event` function as defined in Figure 2.

   In the definition referenced, `<SM_NAME>` is replaced with the name of the state machine to which the event is to be delivered. The `sm_obj` and `sm_evt` arguments must be the appropriate entities with types defined by the `SM_OBJ` and `SM_EVT` SMG directives, respectively. The `event_code` must be of type `<SM_NAME>_event_t`, where that type (and the actual event codes) are defined in the include header file output by the smg preprocessor.

4. For state machines which are not initialized at instantiation time (i.e. which do not contain an `INIT_STATE` directive) the state machine must be explicitly initialized before any events are delivered to the state machine. A state machine must be explicitly initialized by calling the `<SM_NAME>_State_Machine_Init` function as defined in Figure 2.

5. The `.sm` file should contain a "`#include "<FILE>_smdefs.h"` C statement, where `<FILE>` is the same as the `.sm` input filename (without the `.sm` extension). This header file is automatically generated by SMG and will define the states, events, and various SMG entry points in C-syntax code.

   This header file *must* be included into the `.sm` file at a minimum, and may be included into other `.sm` or `.c`/`.h` files in the module as needed. Its inclusion must be explicit by the developer to insure that the inclusion occurs at the right point in the code (e.g. following any other definitions needed by the SM definitions, but prior to actual usage of those definitions in the user-supplied code).

6. The `_smdefs.h` inclusion in requirement 5 must be preceeded by any type definitions required by code declarations in the `_smdefs.h` file, including the `SM_OBJ` and `SM_EVT` types.

7. The SMG code assumes that all external calls to `<SM_NAME>_State_Machine_Event` are externally synchronized and otherwise protected against multi-thread re-entrancy.

8. Code segments are output in the order in which they are specified for `TRANS+` code. All pre-state `TRANS+` code preceeds normal pre-state code, and all post-state `TRANS+` code follows normal post-state code.

5 SMG Usage

The SMG preprocessor is invoked from the command line prior to the C file compilation. It will produce a number of output files: a C code file (`.c`), a header include file (`.h`), a GraphViz dot input file (`.dot`), and a GraphViz output file in PostScript, GIF, MIF (Framemaker), or HTML imagemap format.

The SMG preprocessor may be invoked with no arguments or with the "`-h`" flag to obtain explicit usage information.

5.1 SMG Command Line Arguments

The following describes the command line arguments for SMG in more explicit detail than than provided by run-time help.

`-h` Displays the usage/help information. Usually displayed on a command-line parsing error as well.

`-i` Passes all generated `.c` and `.h` files through the indent program to improve readability. The current user environment provides the appropriate

```
void <SM_NAME>_State_Machine_Error(<SM_OBJ> _sm_obj,
                                   <SM_EVT> _sm_evt,
                                   int   err_id,
                                   char *errtext, ...);

void <SM_NAME>_State_Machine_Event(sm_obj, sm_evt, event_code);

void <SM_NAME>_State_Machine_Init(sm_obj, initial_state);
```

Figure 2: State Machine C-code Interface Declarations

Table 1: State Machine Error function `err_id` and `errtext` values

err_id	errtext
0	*unused*
1	Undefined State Transition (State <#>=<N>: <D>), (Event <#>=<N>: <D>)
2	Invalid STATE!! (<#>=<N>: <D>)
3	Invalid STATE/EVENT!! (State <#>=<N>: <D>) (Event <#>=<N>: <D>)
	<#> is the numeric value for the item in question <N> is the corresponding name string <D> is the corresponding description string

input to indent; no indent parameters are passed on the command line used to invoke indent. The indent application must be present in the current `PATH`. Note that some indent operations may cause the line numbers reported during compilation to be skewed; in particular, the `-cdb` (GNU indent) option is evil.

`-v` Verbose output. Enables various informational and progress messages. If verbose output is not enabled, the SMG preprocessor will not generate any messages to stdout.

`-D` Defer tagged code event generation. As described above, a `TRANS` directive may specify one or more tags identifying code that is to be generated for handling that event, and that code may contain keywords of the type "`_#<event>`" or "`_/<event>`" where `<event>` is a valid event for this state machine. Normally, the keyword will be directly replaced with appropriate code to generate the specified event, but if the `-D` flag is used, the code to generate the event will be placed at the end of the `TRANS` code segments.

`-N` Nested switch statements for state machine handling. When an event occurs, there are two elements which are used to determine the appropriate handling of that event: the event code itself and the current state. By default, these two values are combined and a singe C switch/case statement is generated to dispatch to the appropriate handling. When the `-N` flag is utilized, one C switch/case statement is used to scan for the current state, and then within the case handling for that state, another C switch/case statement is used to scan for the `event_code`.

`-T` Trace code output. When this flag is specified, state machine trace operations are embedded in the generated code. This is typically used for debugging and may or may not be specified for "production" versions of the code. More information on tracing is provided in Section 5.3.

`-e` Enum declarations. By default, the `XXX_smdefs.h` file generated uses `#define`

statements for states and events. When the `-e` flag is specified, `enum` statements are generated instead.

`-b` Bounds checking. Adds code to the state machine event handler to validate the event value for validity as an actual event code.

`-l` Suppress `#line` directives. By default, the generated code contains "`#line`" C precompiler directives that reference the input SMG file. This greatly aids in debugging as the compilation stage will usually reference the correct location in the input SMG file rather than the generated (and therefore somewhat unfamiliar) C file. By specifying the `-l` flag, these directives are suppressed and any C compiler messages will refer to the `.c` file instead; this may be useful for debugging errors that are not apparent in the SMG input file directly.

`-G` GIF diagram output format. Specifies that the state machine diagram that is generated is to be encoded in GIF format.

`-P` Postscript diagram output format. Specifies that the state machine diagram that is generated is to be encoded in Postscript format. This is the default.

`-M` MIF diagram output format. Specifies that the state machine diagram that is generated is to be encoded in MIF format (Adobe FrameMaker's Interchange Format).

`-W` Web imap diagram output format. Specifes that the state machine diagram that is generated is to be encoded as an html image map suitable for use on a Web page.

5.2 Support Functions

The SMG utility will automatically generate a set of support functions in addition to the primary `<name>_State_Machine_Init` and `<name>_State_Machine_Event` functions. These support functions may be used by the user's event-specific code or other code to obtain user-readable names and descriptions of the States and Events defined in the `.sm` file.

The following support functions (defined in Figure 3) are generated automatically by SMG:

`<name>_State_Name` This function converts its state value argument into a character string name for that state.

`<name>_State_Desc` This function converts its state value argument into a character string description for that state.

`<name>_Event_Name` This function converts its event value argument into a character string name for that event.

`<name>_Event_Desc` This function converts its event value argument into a character string description for that event.

5.3 Run-Time Tracing

When the SMG utility is invoked with the `-T` argument, the generated SMG code contains run-time tracing output. By default, this tracing output causes information to be printed to stderr regarding the various events and state changes that occur when the SMG-generated state machine operates.

Some environments do not provide fprintf access to a stderr output stream, and other situations might desire custom control over the tracing output. These scenarios can be handled by overriding the default tracing code generated. The overrides are specified in the form of a set of `#define` statements in the `.sm` file. These `#define` statements should appear fairly early in the `.sm` file, and must appear before the inclusion of the `xxx_smdefs.h` file.

The following macros should be overridden with specific `#define` statements to modify the tracing behavior:

`SM_TRACE` This macro should be defined to signal an override of the default tracing functionality generated by SMG. This macro does not need to be set to a specific value or translation: it simply needs to be defined.

```
char *<name>_State_Name(<name>_state_t state);
char *<name>_State_Desc(<name>_state_t state);
char *<name>_Event_Name(<name>_event_t event);
char *<name>_Event_Desc(<name)_event_t event);
```

Figure 3: SMG Auto-generated Support Functions

`SM_TRACE_INIT` This macro is called when the state machine is initialized. This macro can be used to trace the initialization event and the initial state of the state machine. The parameters for this macro are:

- `Obj` The `_/OBJ` variable for the state machine context. (Type: declared by the `SM_OBJ` directive).
- `Evt` The `_/EVT` variable for this event. (Type: declared by the `SM_EVT` directive).
- `SM_Name` The `SM_NAME` of the state machine. (Type: `char *`).
- `InitState` The initial state value. The `<name>_State_Name` and `<name>_State_Desc` support functions may be called to translate the state value into an identifying string and corresponding description if desired. (Type: `<name>_state_t`)

`SM_TRACE_EVENT` This macro is called when an event occurs. This macro should identify the event which occurred and the new state of the State Machine. The parameters for this macro are:

- `Obj` The `_/OBJ` variable for the state machine context. (Type: declared by the `SM_OBJ` directive).
- `Evt` The `_/EVT` variable for this event. (Type: declared by the `SM_EVT` directive).
- `SMNAME` The `SM_NAME` of the state machine. (Type: `char *`).
- `Event` The event value. The `<name>_Event_Name` and `<name>_Event_Desc` support functions may be called to translate the event value into an identifying string and corresponding description if desired. The new state value is available from `_/OBJ->sm_state` and it may likewise be converted to readble strings by the corresponding support functions. (Type: `<name>_event_t`)

`SM_TRACE_EXP_EV` This macro is called when an expected event occurs (*i.e.* an event transition declared with a `TRANS=` directive). This macro is otherwise identical to the `SM_TRACE_EVENT` macro, including the arguments, but this macro may wish to differentiate the occurrence of an expected event from the occurence of a normal event in the generated trace output.

6 SMG Libraries

An interesting and useful extension of SMG usage is as a new method for defining interface libraries, especially for asynchronous event–driven interfaces. SMG features that make it very useful in working with event–driven architectures. Event–driven architectures are often asynchronous by nature and lend themselves to state machine management techniques, but SMG has the ability to specify entry points and setup code for those asynchronous entry points as an SMG Library that can be included into the applications SMG specification using the `SM_INCL` directive.

6.1 Event Interface Background

For example, consider a hypothetical windowing environment called "Y". Applications written for Y-windows must specify functions that are called when certain events occur within their window. The fol-

lowing events must be handled by a Y-windows application:

- Mouse click, Button 1
- Mouse click, Button 2
- Keyboard key entered
- Expose event (the window has become visible)
- Iconify event (the window should be iconicized)

For a C development environment, there would be a Y-windows header file that defined these events and an event vector structure that would have to be initialized by the application:

```
typedef struct y_app_vtable {
      void *y_mouse_b1(int x, int y);
      // tricky: arguments switched:
      void *y_mouse_b2(int y, int x);
      void *y_key_entered(char keyval);
      void *y_expose(void);
      void *y_iconicize(void);
}  y_app_vtable_t;

int y_app_mainloop(void);
```

The Y-windows application would then need to define functions to be entered into the `y_app_vtable_t` to handle the various events. The application would also need to call `y_app_mainloop` in its main routine after initializing so that Y-windows could begin processing events and passing them to the application's handling functions.

The inconvenience of this methodology is that the process of creating the entry functions is a tedious, mechanical process that each Y-windows application must perform. This is exacerbated by the fact that the first thing that each entry point must do is check the global state to see how the event should be handled. For example, `myapp.c` would contain the following minimum code irrespective of the specifics of the application:

```
void myapp_mouse_b1(int x, int y) {
    switch (app_state) {
    case STATE_1:
          <some code>
          break;
    case STATE_2:
          <some code>
          break;
          :
    }
}

void myapp_mouse_b2(int y, int x) {
    switch (app_state) {
    case STATE_1:
          <some code>
          break;
    case STATE_2:
          <some code>
          break;
          :
    }
}

void myapp_key_entered(char keyval) {
      // same as above...
}

void myapp_exposed(void) {
      // same as above...
}

void myapp_iconicize(void) {
      // same as above...
}
```

Instead of exposing the developer to the tedium of manually writing this template an SMG Library can be provided for Y-windows. This also removes the risk of entering it incorrectly, (eg. as `void myapp_mouse_b1(int y, int x)` which is syntactically but not functionally correct).

6.2 SMG Library Implementation

An SMG Library is nothing more than an SMG file that should be `SM_INCL` included into the main application and which defines the events and entry points for those events. This library can be developed once and re-used by all applications that operate within that event-driven architecture. The application developer is freed to focus on the actual functionality of the application in response to the events rather than the mechanics of program infrastructure.

A subset of the example above can be used to show how the introduction of an SMG Library changes the development. The initial introduction of the SMG Library generated state machine would replace all the switch statements in the example above into calls to the `myapp_State_Machine_Event` function, thereby saving some of the superfluous repetition, but also making the entry points mere shells:

```
void myapp_iconicize(void) {
    myapp_State_Machine_Event(&myapp_global,
                              <event_obj>,
                              Iconicize_E);
}
```

Because the state machine function will vector the code to the proper handling code, the entry point function needs to do little more than marshall the entry points arguments into the `SM_EVT` object and call the SMG's state machine function.

In the revised example, it is assumed that `myapp_global` is a global structure whose type was reported to SMG by the `SM_OBJ` directive and that that structure contains the `sm_state` field, and it is also assumed that `Iconicize_E` is the name of an event declared with the `EVENT` directive (or implicitly as part of a `TRANS` directive).

The only yet-to-be determined portion of the revised code is the `<event_obj>` specification. This reference must be for a variable of a specific type that is useable to sufficiently represent the event in the state-machine-invoked code. For the `myapp_iconicize` entry point, there are no parameters and therefore the event is fully represented just by its presence. However, other events—such as the `myapp_mouse_b1`—have associated parameters that will need to be available to the event handling code. Therefore, the following event representation structure is defined in `myapp` and declared with the `SM_EVT` directive:

```
typedef union {
    struct {
        int x;
        int y;
    } mouse_coords;
    char keyval;
} myapp_y_event_t;

SM_EVT  myapp_y_event_t *
```

Now all that's needed is to pass a structure of that type to the SMG State Machine Event handler routine. There are a couple of choices as to where to obtain the structure:

- From the local stack. This has the implicit requirement that the event handling code completely handles the event before returning since the stack copy will be gone when that return happens and the event entry point is exited. This is not usually recommended, but it can be used in simple situations.

- From memory allocation. This is probably the most convenient method, but it does mean that the event handling might be delayed by memory allocation time and it also means that the event handling code must deallocate the event structure when it has been fully handled.

- From an event object pool. This uses a pre-allocated pool of structures which has the advantages (fast allocation) and disadvantages (resource limitations) of pool structures. The event handling code should return the event structure to the pool when finished with it.

The example shown in Figure 4 will use the simple memory allocation method. Also note that although *all* events must allocate the structure, different events must initialize the structure in a different manner and some events not at all, therefore a global macro can be used to perform the allocation.

```
#define INIT_EVT  myapp_y_event_t *evt = (void) malloc(sizeof(myapp_y_event_t)); \
                                  if (!evt) return;

void myapp_mouse_b1(int x, int y) {
    INIT_EVT;
    evt->mouse_coords.x = x;
    evt->mouse_coords.y = y;
    myapp_State_Machine_Event(&myapp_global, evt, Mouse_B1);
}

void myapp_iconicize(void) {
    INIT_EVT;
    myapp_State_Machine_Event(&myapp_global, evt, Iconicize_E);
}
```

Figure 4: Event Entry Actions

Now that the example is functionally correct again, all that remains is to recognize that the only variable part is the event encapsulation/initialization code; SMG can be tasked to create the invariant parts *independently* of the application. Furthermore the event object can be defined entirely by the SMG Library since it only needs to contain the explicit arguments defined by the asynchronous API. The SMG Library can define `EVENT` directives with `entry_code` portions that specify the correct functions and arguments for the event, and with `start_code` portions that perform the per-entry-point event structure initialization.

The code in Figures 5, and 6 show how this is done by implementing the SMG Y-windows Library.

Notice that this library contains SMG directives but that it does *not* specify a complete state machine...it does not even name the state machine itself. The directives defined in the library are meant to be embedded in the user's application state machine definition along with the actual event handling code. To continue the previous example, code for a trivial Y-windows application is shown in the following Figures:

- Figure 7 shows the application's declarations
- Figure 8 shows the event-specific routines
- Figure 9 shows the state machine transitions
- Figure 10 shows the main routines

The code shown in these figures can be concatenated in order into a `.sm` source file and handled directly by SMG and a C compiler to generate the sample application.

6.3 Library Synthesized Events

Some types of entry points are passed a parameter which further defines the actual event which occurred. For example, instead of `y_mouse_b1` and `y_mouse_b2`, the Y-windows system could have just defined:

```
typedef enum { Mouse_B1, Mouse_B2 } MButton_t;

void *y_mouse(MButton_t button, int x, int y);
```

In cases like this, the event-specific handling must usually be different based on the actual event which occurred as indicated by the "sub-event" parameter. Rather than require the developer to perform this additional selection based on the sub-event, the SMG Library often provides a synthesized event for each of the sub-event types. In this example this means that the SMG Y-Windows Library would probably

```
CODE_{  yw_mouse_b1_decl
    void yw_mouse_b1_e ( int x, int y )
CODE_}
CODE_{  yw_mouse_b2_decl
    void yw_mouse_bw_e ( int y, int x )
CODE_}
CODE    yw_key_entered_decl  void yw_key_entered_e(char keyval)
CODE    yw_expose_decl       void yw_expose_e(void)
CODE    yw_iconicize_decl    void yw_iconicize_e(void)

y_app_vtable_t yw_entrypoints = {
    yw_mouse_b1_e,
    yw_mouse_b2_e,
    yw_key_entered_e,
    yw_expose_e,
    yw_iconicize_e
};
```

Figure 5: Y-windows SMG Library declarations

still provide `y_mouse_b1` and `y_mouse_b2` events even though the actual event delivered to the application was simply a `y_mouse` event.

6.4 SMG Library Requirements

Using an SMG LIbrary for code development means that the normal SMG development requirements apply plus those specific to the SMG Library. More specifically, the SMG Library must define for the user:

1. the names of the events (including synthesized events),
2. the name and definition for the event structure, and
3. the names of the event and object variables declared by the entry points for the user's event handling code.

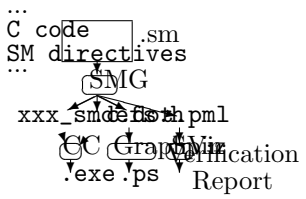
7 SMG Examples

Please see the Examples directory of the SMG distribution for examples and discussions of the use of most of the state machine directives and options in fairly simple programs.

7.1 File Example 1

This example shows a simple example of a menu-driven file processing application that has an under-specified state machine.

The example is located in the `Examples/file_ex1` subdirectory of the SMG distribution. The following sequence of events may be used to work with this example:



7.1.1 Building the Example

In the following example, the commands entered are shown in *italics* and the output of the utilities is shown in `this font`.

```
$ smg -viTP file_ex1
$ gcc -o file_ex1 file_ex1.c
```

Once built, it may be run by executing:

```
$ file_ex1
```

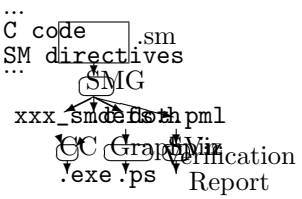
In addition to the executable, the source files (`file_ex1.c` and `file_ex1_smdefs.h`) may be examined. There is also a Promela model generated (`file_ex1.pml`) and a graphical representation of the state machine (`file_ex1S.ps`) based on GraphViz directives (`file_ex1S.dot`).

Examining the graphical representation, it can easily be observed that the state machine is underspecified by the presence of the `UNDEFINED_TRANSITION_RESULT` state. This can be confirmed by performing a sequence of menu operations with the executable that lead to this error state and observing the output dynamically.

7.2 Unmorse Example

This example shows the use of an SMG-generated state machine for translating from Morse Code back to plain-text.

The example is located in the `Examples/unmorse` subdirectory of the SMG distribution. The following sequence of events may be used to work with this example:



7.2.1 Building the Example

In the following example, the commands entered are shown in *italics* and the output of the utilities is shown in `this font`.

```
$ smg -viTP unmorse
$ gcc -o unmorse unmorse.c
```

Once built, it may be run by executing:

```
$ unmorse '- . ... - -..-. .---- .-.-.-'
Translation: TEST/1.
$
```

In addition to the executable, the source files (`unmorse.c` and `unmorse_smdefs.h`) may be examined. There is also a Promela model generated (`unmorse.pml`) and a graphical representation of the state machine (`unmorseS.ps`) based on GraphViz directives (`unmorseS.dot`).

8 SMG Maintenance

8.1 Dependencies

8.1.1 Python

SMG itself is written in Python code and therefore the system must have Python available. The initial version of Python used for SMG development was 1.5.2. Subsequent versions were maintained using Python 2.2; SMG should still run under Python 1.5.2 although this has not been specifically verified.

Python is available from: http://www.python.org/.

8.1.2 GraphViz

The GraphViz utility has been developed by AT&T Research and converts a text description of a graph into a representational Postscript figure. It can alternatively produce GIF, MIF or Web imap figures as well. The graphical output is very helpful in analyzing the resulting state machine.

SMG outputs a description of the state machine itself to a `.dot` file, which is then passed to GraphViz to generate the desired graphical output. The `.dot` file itself is not intended for analysis.

The GraphViz distribution version used for SMG development was 1.5. The GraphViz package is available at: http://www.research.att.com/sw/tools/graphviz

8.1.3 Spin

Promela and the Spin utility are independently developed by AT&T and are not related to the SMG

preprocessor. Spin is available at: http://netlib.bell-labs.com/netlib/spin/whatispin.html

8.1.4 Interactions

The SMG preprocessor is an independently developed utility and is not related to the GraphViz program developed by AT&T nor the Python language and interpreter developed by Guido van Rossum. Python is available from the above Web site. The GraphViz program is available independently from AT&T under their license.

8.2 SMG License

8.3 SMG Change Log

v1.7.1
- *[Internal]* Created graphviz module and modified `smg_figure` to use it.

v1.7
- Added documentation for `SM_ELSE` (not new to SMG, just not doc'd).
- Added path search for `SM_INCL`
- Added optional version parameter for `SM_INCL`
- [Documentation:]
  - Document updates: clarification on `<xxx>` keyword validity.
  - Cleanup of existing examples
- [Documentation:]
  - Documented SMG Libraries
  - Documented Support and Tracing Functions
  - Documentation rearranged somewhat

v1.6
- Minor updates. Preparation for SF file release.
- Addition of `SMG_SYNTAX_A` syntax version context definition.

v1.5
- Added `sm-mode.el` for Emacs syntax highlighting and indenting controls.
- Added handling for cascaded sub-events.
- Added validation to detect terminal states, source states, and erroneous combinations: non-`INIT_STATE` source states, and orphan states (no entries or exits)

- Added `-b` flag for bounds checking incoming events. This also changed the values (and ordering) for states and events in the `smdefs.h` file.

v1.4
- Added the `INIT_STATE` for instantiation-initialized state machines.
- Modified state machine diagrams for visual identification of key states (initial states, final states, and the undefined transition error state).
- Expanded examples to: `file_ex1`, `file_ex2`, `file_ex3`, and `file_ex4`.

v1.3
- Added `PROMELA_{` and `PROMELA_}` and output of Promela code for spin verification.
- Updated `xx_State_Machine_Error` function to include the `err_id` for identifying and processing specific error numbers.

v1.2
- Added clustering of nodes in figures

v1.1
- Added `SM_DEF` , `SM_IF` , and `SM_END` statements for conditional inclusion.
- Internal code cleanup primarily focused on:
- Significant Postscript output cleanup and labelling.

v1.0.2
- Better tracking of source `.sm` file, especially for `SM_INCL` directives
- Switch to "`_/`" instead of "`_#`" as the SMG identification prefix. Still recognize the old, but the new one prevents macro argument replacement errors.
- Updated `SM_TRACE` for explicit macros for each type of trace

v1.0.1
- Added `SM_TRACE` and "`-T`" command line option (tracing)
- Added "`-l`" command line option (#line output suppression)
- Added version number and output version on help line
- Added `_#SM_NAME` code keyword
- Added internal event generation deferral for `_#<EVENT>`

```
typedef union {
    struct {
        int x;
        int y;
    } mouse_coords;
    char keyval;
} yw_event_t;

SM_EVT  yw_event_t *

EVENT  Mouse_B1_E    yw_mouse_b1_decl    yw_mouse_b1_prep
EVENT  Mouse_B2_E    yw_mouse_b2_decl    yw_mouse_b2_prep
EVENT  Key_Entered_E yw_key_entered_decl yw_key_entered_prep
EVENT  Expose_E      yw_expose_decl      yw_noarg_prep
EVENT  Iconicize_E   yw_iconicize_decl   yw_noarg_prep

CODE_{   yw_mouse_b1_prep
// Standard event initialization; defined in CODE tag for _/xxx subst.
#define INIT_EVT  yw_event_t *evt; \
           evt = (void) malloc(sizeof(yw_event_t)); \
           if (!evt) return; \
           _/EVT = evt; \
           _/OBJ = &myapp_global;

    INIT_EVT;
    evt->mouse_coords.x = x;
    evt->mouse_coords.y = y;
CODE_}

CODE_{   yw_mouse_b2_prep
    INIT_EVT;
    evt->mouse_coords.x = x;
    evt->mouse_coords.y = y;
CODE_}

CODE_{   yw_key_entered_prep
    INIT_EVT;
    evt->keyval = keyval;
CODE_}

CODE     yw_noarg_prep    INIT_EVT;
```

Figure 6: Y-Windows SMG Library Event Management

```
#include <stdio.h>
#include <ywindows.h>
#include "myapp_smdefs.h"

SM_NAME  myapp

struct myapp_gstruct {
    sm_state_t sm_state;
} myapp_global;
SM_OBJ struct myapp_gstruct *

SM_INCL  Y_Windows.sm
```

Figure 7: Y-windows Application declarations

```
CODE_{ draw_window
        [user code here]
        free(_/EVT);
CODE_}

CODE_{  draw_icon
        [user code here]
        free(_/EVT);
CODE_}

CODE_{  add_char
        y_window.text.add(_/EVT->keyval);
        free(_/EVT);
CODE_}
```

Figure 8: Y-windows Application event handling routines

```
INIT_STATE NotDisplayed
TRANS NotDisplayed Expose_E       Displayed   draw_window
TRANS Displayed    Expose_E       -           draw_window
TRANS Iconicized   Expose_E       Displayed   draw_window

TRANS Displayed    Iconicize_E    Iconicized  draw_icon
TRANS Displayed    Key_Entered_E  -           add_char
TRANS Iconicized   Key_Entered_E  -

TRANS Displayed    Mouse_B1_E     -
TRANS Iconicized   Mouse_B1_E     Displayed   draw_window
TRANS Iconicized   Mouse_B2_E     -
TRANS Displayed    Mouse_B2_E     Iconicized  draw_icon
```

Figure 9: Y-windows Application state machine transitions

```
int main(int argc, char **argv) {
    myapp_State_Machine_Init(XXX);
    y_window_create(...);
    ...
    y_app_mainloop();
}
```

Figure 10: Y-windows Application main code