Exception Handling and Development Support

DSM has its roots in the late eighties—the early days of C++. Compilers didn't support exception handling then. Conventional error handling by passing error codes up the return stack is a prohibitively code-intensive approach in a large software project with many nested procedural levels such as HP PE/SolidDesigner. Therefore, we had to implement our own exception handling mechanism which is very similar to what has been implemented in today's C++ compilers.

HP PE/SolidDesigner's code is divided into code modules. Each module has its own *module information object* containing module-specific error codes and messages. In case of an error condition inside a module, the code triggers the exception mechanism by throwing a pointer to the module information object.

Code that wants to catch an exception inspects the module information object returned by the exception mechanism and acts accordingly. If it has already allocated resources, they are cleaned up and returned. The exception can then be ignored (and suppressed), or it can be escalated to the next code level.

The listing below shows a code example for this. You may notice the similarities to the exception handling mechanism introduced with C++ 3.0. Now that the throw/catch mechanism is finally available in many C++ compilers on various platforms, we will be able to adopt it with only a few changes in the code.

```
int process_file(const char *const fname)
{
  int words = 0;
  FILE *file = 0;

  TRY
    file = open_file(fname);
    words = count_words(file);
    close_file(file);
    file = 0;
  RECOVER
    if (file) {                                  // clean up resources
      close_file(file);
      file = 0;
    }

    // handle specific exceptions
    if (dsm_exception_code == F2_CORE::info_ptr) {
      switch(F2_CORE::errno) {
    case F2_CORE::BREAK_RECEIVED:     // User has cancelled processing
        // We won't escalate this "soft" exception.
        handle_break();
        break;
     case F2_CORE::MEM_OVL:                  // Out of memory
        // Free memory blocks allocated here, then escalate the problem.
        free_my_mem();
        ESCAPE(dsm_exception_code);          // "throw" in C++ 3.0
        break;
      default:
        break;
      }
    } else {
      // Pass up all other exceptions.
      ESCAPE(dsm_exception_code);
    }

  END_TRY

  return words;
}
```

Development Support

To find problems proactively, DSM stresses the importance of checking preconditions, invariants, and postconditions. It offers convenient assertion macros and a context dependent run-time debugging system which uses *debug module objects*.

These debug module objects hold their current debug level which can be checked using macros and set during run time. A debug module is associated with a certain code area. This allows fine-grained control for debug checks and messages. We think this control is important for the acceptance of a debug system; the programmer will ignore debug messages if there are too many, and won't find the system useful if it doesn't deliver enough detail where needed.

Macros are provided to reduce typing and #ifdef constructs:

```
bool compare(const char *s1, const char *s2)
{
  ME_MODULE_STOPWATCH("compare", foo);      // for run-time profiling

                                            // trace program flow
  if (DEBUG_LEVEL(foo) >= DEBUG_CALLS) {
    fprintf(DEBUG_STREAM(), "compare called");
  }

  DSM_ASSERT(s1 && s2);                      // check precondition

  // Now calculate the result
  ...

  DSM_ASSERT(some_condition);                // check post-condition
  return TRUE;
}
```

DSM also defines special debug modules to switch on sophisticated debugging tools. There are tools to find memory leaks, to calculate checksums for objects (allowing us to detect illegal changes), and to create run-time profiles for the code.

In a software package as large as HP PE/SolidDesigner, the common UNIX profiling tools were not applicable. Therefore, we had to build our own set of versatile, efficient and highly precise utilities. You can define a *stopwatch* for any function that might need profiling, and you start and stop the stopwatch using the debug module mechanism. The results can be analyzed, producing a hierarchical call graph that shows what portion of the run time was spent in the individual functions. We can also find out the amount of memory allocated for a function at run time using these tools.