Integration, testing and debugging of C code together with UML-generated sources

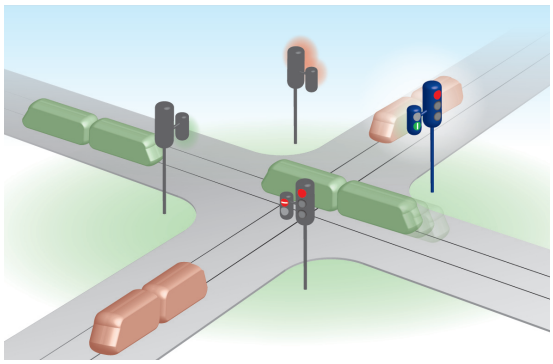

Fig. 1: Adding tram traffic lights to a traffic intersection

Introduction

It will soon no longer be possible to master the growing complexity of software without the use of CASE tools. UML, especially, has enabled the uniform design of modular software components, including automatic code generation, which prevents common coding errors right from the start.

However many companies shy away from the perceived effort required for the changeover to these new techniques. They have a code base that has been built up over decades and although the software has been constantly extended and improved. This has also led to the creation of more and more "spaghetti code". Millions of functioning lines of code would need to be analyzed, re-programmed, and finally re-tested.

In most cases these fears are unfounded. Existing C code can be transferred to a UML model element with almost no changes necessary. Future extensions can then be written as UML models, generated, and integrated with the "old" C code. As a result, an easy and progressive transition from C to UML is possible. The following example explains these procedures:

1. Transfer from "old" C code to UML.
2. Integration of a new functionality coded in UML.
3. Uniform testing and debugging in UML, C++ and C.

Old C code

At the outset, we have a finished project in C code. The following example uses a traffic light system (which is a favorite of software designers due to its simplicity). There is a standard intersection with four sets of traffic lights. The traffic lights located opposite each other are controlled in parallel. The traffic lights at the right angle to them are, of course, controlled complementary. The different traffic light phases are: red – red and yellow – green – yellow – red. During the red phase of one set of traffic lights, right of way switches to the opposite direction. As a result, there is a short period when the lights facing in all directions are on red (this is important, as we shall see later).

This circuit is coded in C as a simple control (switch-case) in a continuous loop (while (1)). This continuous loop is found directly in the "main()" routine of the application, and is called immediately after initialization (see Fig. 2).

```
...
int main (void)
{
   horizontal = vertical = red;
   state = closed_to_h;

   while (1)
   {
      switch (state)
      {
          case closed_to_h:
             wait (1);
             horizontal = yellowred;
             state = yellowred_h;
             break;

          case yellowred_h:
             wait (3);
             horizontal = green
             state = green_h;
             break;
          ...
```

Fig. 2: Existing C code for traffic-light control; the "main()" routine includes a continuous loop that contains the logic as a switch-case construct.

New requirements

The street planners have decided that a tramline should cross the intersection in both directions ⟩⟩

(vertically as well as horizontally, refer to Fig. 1 on page 1). Therefore, the trams must be stopped at the intersection by their own set of traffic lights. At the request of the tram driver, the intersection is blocked for cars at the next red light phase, and the tram is given right of way.

UML wrapper for C code

We could, of course, just "patch" these requirements into our C code. But doing so would only make the function larger and more complicated, which is exactly what we want to avoid. So instead of this, we will take a modular approach to the problem and solve it with the help of UML.

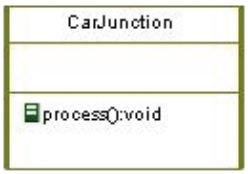

Fig. 3: The "CarJunction" class, containing the complete "old" application

Firstly, we need to "transfer" the existing application over to our UML model, which is easier than it sounds: We create one class, called "CarJunction", which contains the complete "old" application (see Fig. 3).

The only thing that we need to adapt are the corresponding interfaces. The usual continuous loop in main() is already part of the framework and therefore has to be omitted. Instead, "main()" is renamed "processJunction()", and this becomes our new starting point. The end point of the function will be the phase when all traffic lights are on red. Now, if required, we can execute additional actions during this red phase (see Fig. 4). – And that's it.

```
void processJunction (void)
{
   horizontal = vertical = red;
   state = closed_to_h;

   do
   {
      switch (state)
      {
          case closed_to_h:
             wait (1);
             horizontal = yellowred
             state = yellowred_h;
             break;
  /*........*/
          case yellow_v:
             wait (3);
             vertical = red;
             state = closed_to_h;
             break;

      } // switch (state)
   } while ((state != closed_to_h)
         && (state != closed_to_v));
} // processJunction()
```

Fig. 4: Modified C code that can be integrated in the UML tool in this form

UML design for new requirements

Now we can take care of the design for the new requirements. We build a class called "Junction", which contains the old traffic light controls, called "CarJunction", as well as two new tram traffic lights (Fig. 5). The design is now finished, and we can focus on the behavior of the extended intersection, using a state transition diagram.

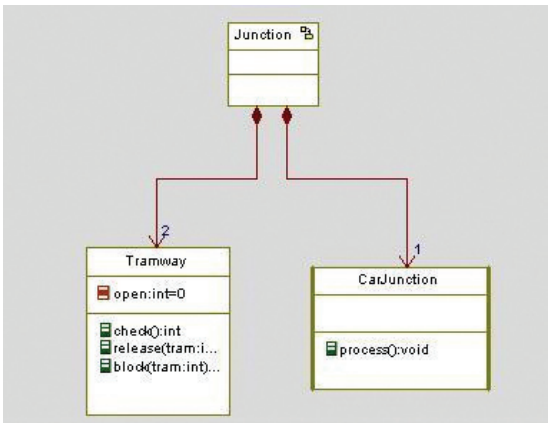

Fig. 5: The new class, "Junction", integrating the "old" traffic-light switching and the two new traffic lights for the tram

The old application is once again used as a starting point, and its whole behavior is incorporated into a single state of the new intersection (state: cars), see Fig. 6 on the opposite page. When the new model has this state, it works in exactly the same way as the old application. Only when the new feature – the tram driver requests right of way – is required, the old code is left and the new model starts. The phases and the transition of states for both tramlines are defined in the state diagram: request – wait – release – wait (possibly a switch between tramlines). This completes our model of the new requirements. ⟩⟩

Code generation

We can now generate the executable application. For good UML tools this is all that is required for generating the finished application for the above-mentioned model. However, now is the time to note a few points about integration. Our initial application is written, and remains, in C. UML tools, however, normally generate C++ code, meaning that our "CarJunction" is a wrapper class that references the C code. Here, the usual adaptations between C and C++ must be taken into account.

Typically, an RTOS is used to manage the model and must be integrated as required. Once all this has been done, a simple press of a button generates a ready-to-run application from the model.

Target run

In the field of embedded technology, the target hardware is often different from the computers used for development. This means that we need to load the generated application onto the so-called "target" and run it there. This can be done using any external tool that can install code on the target.

This is especially easy if the UML tool and the debugger communicate over a common (software) interface, such as the integration between the UML tool Rhapsody from Telelogic and the TRACE32 debugger from Lauterbach. This allows you, at the press of a button in the modeling tool, to load the application to the target over the debug interface, and – if required – start it straight away. The complete cycle of design – modeling – generation – run is possible in one GUI.

Testing in C

As was mentioned at the start, performing testing and debugging in a heterogeneous environment pose certain requirements. It must be ensured that the whole application can be tested in its respective function and implementation.

Although we are assuming that we already had a functioning application in C, debugging should still be possible. After all, the transfer of new features to the "old" code needs to be tested, and the old source code maintained. At least for this part of the application any commercial debugger can be used, as it is now standard in the field of embedded technology to debug C code.

Testing in C++

The process becomes more interesting when we begin testing the new features at source level. Most UML tools generate C++ code with all the ⟩⟩

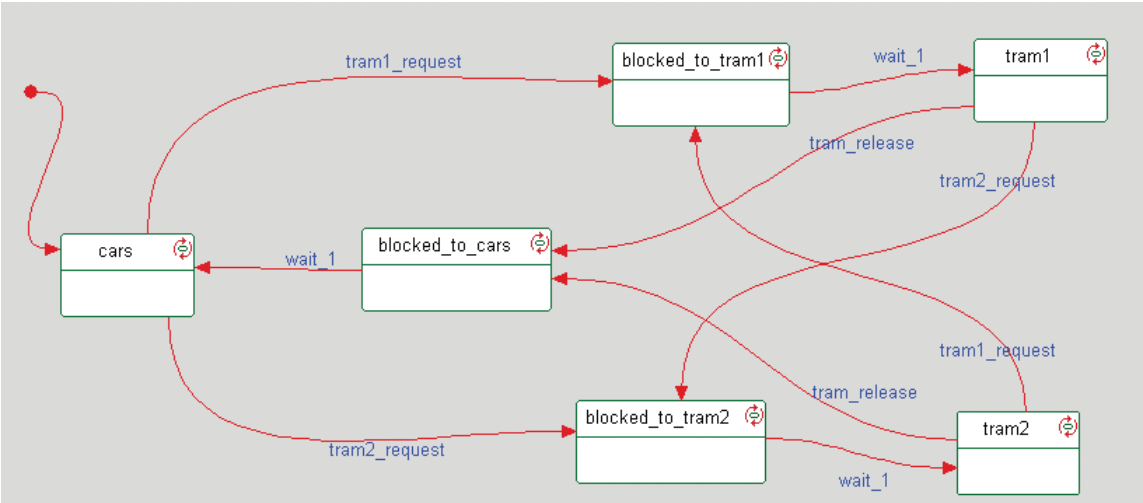

Fig. 6: State diagram of new traffic-light switching

characteristics that belong to it, such as templates, polymorphism, and exception handling. It must be remembered that the debugger used must provide full support for the C++ dialect of the compiler. The TRACE32 debugger from Lauterbach supports all current C++ compilers, and so guarantees comfortable debugging of object-oriented C++ code.

Testing in UML

But of course, we didn't write our code in C++, but rather in UML. And what would make more sense than to carry out debugging on the modeling level?

Rhapsody from Telelogic offers several options to serve this purpose.

If the behavior of the application is completely modeled in UML, the sequence can be simulated directly from the model. For simple behavior analysis, the actual target hardware is not required. As the target will have a different timing than the simulation, the application can also be run as an "animation" on the real target. The UML tool controls and visualizes the target process over a communication channel (serial or Ethernet). So it is possible to perform single steps through state charts or to induce events.

In combination with a TRACE32 debugger, this communication can also take place over the debug interface if there is no other interface available. Direct debugging in the UML model is made possible via "animation".

Integrated UML -> C++ -> C debugging

Now we have a situation spread over three levels: UML, C++, and C. When an error is found, especially in C++ code, it is better to repair it in the model, and not in the generated C++ code itself. On the other hand, the debugging of the implementation of a model element is best carried out in the source, without having to search for it.

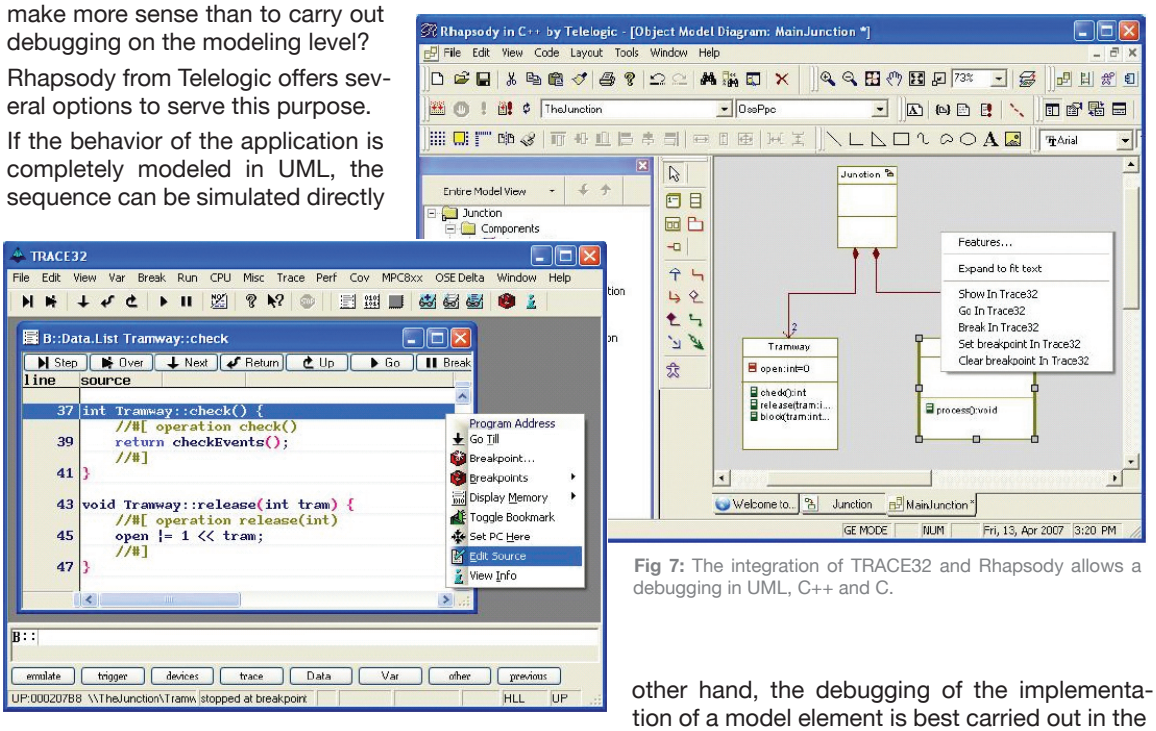

Fig 7: The integration of TRACE32 and Rhapsody allows a debugging in UML, C++ and C.

The integration of TRACE32 and Rhapsody offers extensive functionality in order to link these levels together (Fig. 7), such as the implementation of simple, interdependent navigation. A single mouse click on a model element in Rhapsody is enough to display the corresponding source code in TRACE32. This then allows the extensive analysis of variables, function calls, and so on. It also works the other way around; with a simple click on a line of source code in the debugger, the accompany-

ing model element is highlighted in the UML tool. This is especially helpful if a problem is found with the debugger, which needs to be repaired in the model. A lengthy search and navigation to the relevant element thus becomes unnecessary.

Debugger breakpoints can also be defined in the model and the target can be set into Go or Stop status. This enables you to set a breakpoint at a class and start the target, all from within the UML tool. The debugger stops the application as soon as the model element under examination is invoked.

Summary

A complete redesign is not always necessary. Particularly for projects where re-engineering is not possible due to time or organization constraints, a step-by-step approach is often appropriate. This limits the effort required and allows to import the existing software step by step into a modern and innovative CASE tool.

It is, of course, essential that the tools used in this process help and don't hinder; the selection of the correct tools is an important factor. They should not just be able to perform these tasks, but should also encourage them.

Lauterbach and Telelogic have achieved just this with the integration of their tools TRACE32 and Rhapsody. Both companies want to help their customers keep software quality at high levels, despite increasing complexity and ever more requirements. Now nothing stands in the way of modernizing your existing code bases with UML.