プリンター開発への適応をめざして

富士ゼロックス(株) DPC 商品開発統括部　IOT-PF 第一開発部
清水 哲

はじめに

弊社では、オフィス等におけるドキュメンテーションに関する機器・サービスを提供させていただいておりますが、私どものセクションにおいては、小型レーザープリンターの制御ソフトの開発を担当しております。レーザープリンターにおいては、ここ数年、ソフトウエア処理の比重がどんどん増加する傾向にあります。その背景のひとつには、カラープリンターの急速な立ち上がりがあげられます。これにともない、制御ソフトにおいても、従来の単色現像器のみの制御から、複数の現像器の制御へと、制御対象が拡大しただけではなく、画質を維持するためのさまざまな制御が求められるようになりました。もうひとつの背景としては、システムの複合機化があげられます。プリンターに、スキャナーやファクシミリなどのさまざまな機器を接続してひとつのシステムを構成するようにしたものですが、これにともない、サブシステム間での通信のサポートが、制御ソフトの中で、大きな割合を占めるようになってきました。このような「カラー化」あるいは「複合機化」という市場要求がある一方で、ここ数年で、プラットホームとなるCPUの事情も大きく変化しました。8 ビットから 16 ビットへ移行しても、コスト的には十分に見合うようになり、16 ビットの製品でも、処理速度がずいぶんと向上しました。このようにして、市場要求と、それを支える技術の向上とが、車の両輪のようにがっちりと結びつき、それが結果として、ソフトウエアの規模増大の要求へとつながってきたわけです。したがって私どもは、このような中にあって、高品質のソフトを、短期間で開発しなければならないという状況におかれています。

従来の開発における問題点

従来の設計プロセスは、まず、メカニカルエンジニアによって動作の概略が決定されると、それをもとにして、ソフトウエアエンジニアが一緒になって、ソフトでの実現性を勘案し再検討し、この過程を経てできあがった仕様をもとにして、ソフトウエアの概略・詳細設計に入っていくというステップをとることにしていました。まず初期的には、このようにしてスタートするのですが、ただ、当初の仕様がそのまま最後まで残ることはまれです。メカトラブルもありますが、新規技術を採用した場合は、そのカット・アンド・トライで、途中での仕様変更がかなり入ります。このような場合も、基本的には上記の設計プロセスで対応するのですが、ただ、常に概略設計からやり直すわけにはいかないので、その時々の仕様変更のメリットと、ソフト変更のリスクとを勘案して、ソフトとしてどこまで戻って設計変更するのか、という観点からの検討も実施し、最終的な落とし所をさぐります。これまでは、このようなプロセスで開発を進めてきたのですが、前任機の開発において、バグの多発が大きな問題になりました。その原因を分析してみると、仕様変更―再設計のプロセスの中で、設計段階における検討不足という問題が浮かび上がってきたのでした。

そこで、後継機の開発開始にあたり、バグ件数を低減するために、設計段階での検討不足に対する有効な手段はないかと、検討していた頃に出会ったのがZIPCでした。従来の設計プロセスにおいても、一部のサブシステムでは状態遷移表を設計資料として持っていましたが、全体としては、必須の設計資料とはされていませんでした。レーザープリンターを制御の面から考えると、基本的には、ステートマシンと考えることができるので、すべてのサブシステムで状態遷移表を作成することが可能なはず

です。そこで、後継機においては、一部のサブシステムでZIPCの導入を前提しながら、また今後の展開も考えて、すべてのサブシステムにおいて、状態遷移の考え方に基づいて、状態遷移表を作成して設計することにしました。

ZIPCの効果

状態遷移表を作成する設計プロセスにして良かったことは、まず状態定義をとおして、担当するサブシステムの状態モデルをじっくり検討することができたことです。将来の仕様変更も予想しながら、それにも対応できるような状態モデルを作成することは、ソフトウエアの基本設計のキーになるところです。ここは、レビューを重ねながら慎重に進めましたが、従来よりは見通しの良いモデルを作成できたと考えています。また状態遷移表にある入力条件をチェックすることによって、サブシステム間のインターフェイスの整合性を十分に検討することができました。システム間インターフェイスは、見落とすことが多い部分ですが、それを早期に発見することができました。また、イベントごとの処理の適否なども、じっくりと検討することができました。これらは、仕様変更への対応においても有効でした。このプロセスは、ZIPCのSTMエディタを中心に進められました。すなわち、STMエディタを使って、はじめは、処理をコメント文で記述しておいて、だんだんとC言語で書き加えていくという方法をとりました。これによって、最初から最後まで、同じフォーマットでレビューすることができました。そして、さらにジェネレータを使用すると、状態遷移表が完成すると同時に、コーディングまで終了することができました。

このように、設計プロセスに状態遷移の考え方を明示的に導入し、一部のサブシステムにおいてZIPCを導入したわけですが、この結果、後継機のソフト開発におけるバグ発生率は大きく改善されました。まだ、開発が終了していないので、正確な数字ではありませんが、おおよそ前任機に比べて、システム規模では約20%ほど大きくなったにもかかわらず、バグ発生件数では30%低減を達成するだろうと予測されています。この結果は、コストの面からも、十分に意味のあるものでした。プリンターのデバッグには、シミュレーションデバッグのほかに、実機で、実際に印字をしながらデバッグする方法とがあります。開発の最後のフェーズでは、完成度を上げるために、後者の方法による比重が大きくなります。このデバッグ方法は、コスト面から考えても、トラブルの再現工数だけでなく、その間のマシンのランニング費用も発生するために、あまりうまい方法とはいえません。それゆえ、バグの発生件数が抑えられたということは、品質だけでなく、コスト面でも大きな成果があったと考えています。

なお補足すると、ZIPCの導入にあたって懸念していたことは、ZIPCのメモリ消費量はどうなるのか、ということでした。メモリの消費量は製品のコストアップにつながるので、とくにRAMサイズについては制限が厳しく、ZIPCを採用したからといって、目標値をオーバーすることは許されませんでした。結果として、これは杞憂に終わりました。今回は、自分たちで同じ処理を作成した場合との比較はしていませんが、メモリ消費量が、状態遷移表の数や大きさに比例するのはやむを得ないとして、概ね納得のできるサイズに収まっていると判断しています。もしかしたら、今回は、ある程度、どんなコードに落とされるかを意識しながら、状態遷移表を作成したのが良かったのかもしれません。ただ、今後の展開の中では、そうでない場合はどうなのか、という検証もしておかなければならないと思っています。エンジニアのスキルへの依存度が小さいツールが望ましい、と考えるからです。

まとめ

今回は、ZIPCの豊富な機能のほんの一部分を利用したに過ぎませんが、現時点では、ZIPCは、制御用ソフトウエア開発にとって有効なツールであると認識しています。

今後、改善を希望するとすれば、以下の二点を申し上げたいと思います。

弊社では、現在、開発の最終段階を迎えています。納期がかなりタイトになってきたため、直接、プログラムに修正を加えています。ある程度目処が立ったところで、整理しようと考えていますが、状態遷移表まで戻るデバッグサイクルというのは、やはり時間がかかかるものです。また、弊社で採用したCPUは、ZIPCデバッガーのサポート対象外だったということもあり、デバッグ時に複数のツールを使用しなければならない煩わしさもありました。そういった意味で、今後は、ICEメーカーやツールメーカーなどと連携して、できるだけ多くのCPUをサポートした、統合的な開発環境を提供していただけると、さらに便利になるのではないかと期待しております。

また、商品開発に携わっていると、開発スケジュールに教育プログラムを入れることは、なかなか困難です。したがって、新しいツールであっても、すみやかに立ち上げなければなりません。その意味では、もう少し詳細なマニュアルがあると便利かとも思いました。また、新人教育を考えると、状態遷移表の作成手引きのようなものがあると良いとも思いました。また、ZIPCで作成されたソフトも、製品に組み込まれればメーカーの責任になりますので、ZIPCのコード生成仕様書を公開していただけると、なお安心して使用できるのではないかと思っています。

（しみず さとし）