Towards Dependable Development Tools for Embedded Systems A Case Study in Software Verification*

Uwe Petermann
Dept. of Computer Science
University of Applied Sciences Leipzig
P.O.B. 300066
D-04251 Leipzig (Germany)
uwe@imn.htwk-leipzig.de

Abstract

This case study describes the specification and formal verification of the key part of TeCOM, a development tool for the design of open loop programmable control developed at the University of Applied Sciences in Leipzig. TeCOM translates the high-level representation of an open loop programmable control into a machine executable instruction list. The produced instruction list has to exhibit the same behavior as suggested by the high-level representation.

We discuss the following steps of the case study: characterization of the correctness requirements, design of a verification strategy, and the correctness proof.

Key words: program verification, modular specification, interactive proving, programmable control

Introduction

Below we describe a case study devoted to the verification of essential parts of a development tool for programmable open loop control. This verification task is not an academic one. The translator TeCOM, which is subject to the verification is used by an engineering enterprise. The output of the program being verified serves as embedded software in safety relevant industrial installations. Therefore a formal verification is necessary. The verification is carried out in a tight co-operation with the authors of the translator.

TeCOM — the subject of verification

The translator TeCOM is part of a design suite for open loop programmable control projects in industrial plants. TeCOM bridges the gap between the process execution plan (PrEP), the graphical representation of an open loop programmable control program, and the instruction list which will be executed effectively by a control device. The usefulness of this tool has been proved by a number of successfully working control projects. Considerable savings of installation time have been observed due to the process oriented representation and the automated generation of the corresponding instruction lists. TeCOM hides the peculiarities of a particular target language and supports portable solutions. The well documented graphical representation may be part of the contract between solution provider and customer. Among the supported goal languages are widely used languages like STEP5, STEP7, DOLOG AKF, MEDOC-IL or FST 101. The focus of the verification project is the generation of the binary processing unit as the key fragment of an instruction list. Other feature like arithmetic operations, handling of timers or input and output of various signals are also supported by TeCOM but remained out of the scope of this case study.

The motivation for a formal verification

TeCOM should meet the following basic correctness requirement. An instruction list which is the output of TeCOM should exhibit the same behavior as the process execution plan given as input. The constructed instruction list is embedded software in a control project. The solution provider is responsible for the behavior of the realized control solution.

Proof reading the generated instruction list, though possible in principle, is error-prone and would cause the lost of efficiency gains enabled by the automated code generation.

Checking the correctness of the translation result automatically following (Pnueli, Shtrichman, & Siegel 1998) looks promising. However, one of the basic aims of the TeCOM-implementors was to obtain an improved translator. Therefore the verification of the translator itself was in the scope of interest. Nowadays tools for formal specification and verification of software make this possible. Correctness may be proved with mathematical strength based on a formal specification of the translation.

*Supported by the Federal Ministry of Education and Research (BMBF) and by the Research and Transfer Center at the University of Applied Sciences Leipzig.

The verification tool KIV

The chosen specification and verification tool KIV (Reif & Schellhorn 1997) supports the following software development steps:

- specification and implementation of data structures,
- proof of correctness and security properties of the implementation.

KIV supports structured specifications based on the operations enrichment, parameterization, actualization, union and renaming of specifications. The implementation of one specification in terms of another is defined by a module. The correctness of each module does not depend on any other module. Therefore the effort for proving the implementation correctness of the whole project is linear in the number of the modules of the project (Reif 1992). Advanced proof-engineering techniques of KIV support the re-use of proof fragments which remain correct after changes in specifications or implementations. This is extremely useful because many attempts are needed for designing a correct program. Strong heuristics assure a high degree of automation.

The compiler TeCOM

In this section we describe the main steps of the translation procedure which are performed by TeCOM.

The process execution plan

Figure 1 shows a sample process execution plan and Figure 3 the instruction list obtained by its translation. Mathematically, the process execution plan is a directed graph with nodes and edges labeled. Two kinds of nodes can be distinguished. Those which are represented by boxes are called *operations*. One of them is distinguished as the *initial operation*. Operations are labeled by sequences of *operation variables*, in the example: R, S or T. Operation variables correspond to activities to be carried out by the process which is subject to the control. The control triggers those activities if the corresponding *process conditions* hold. Process conditions are notated by rounded rectangles. Their labels are the *process variables* (A, B and C in the example).

The semantics of a process execution plan P is described by an automaton A_P (cf. Figure 2) with the operations of P as states and the initial operation of the process execution plan P as initial state. The input alphabet of A_P is the set of valuations of process variables by Boolean values. Similarly, the set of valuations of the process variables by truth values is the output alphabet of A_P. For example, the label $A\overline{B}$:R,S of the

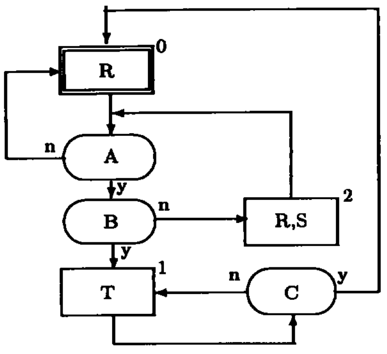

Figure 1: A process execution plan

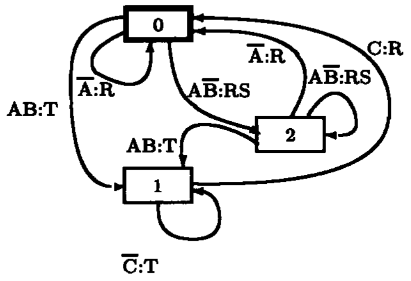

Figure 2: The automaton assigned to the process execution plan in Fig. 1 and its factorization

transition from state 0 to state 2 means that this transition takes place if and only if process condition A is satisfied and B not. In that case the process activities R and S will be triggered. The behavior of process execution plan P is the word function determined by the initial state of automaton A_P. Automaton A_P is factorized by a congruence relation $\equiv$ in order to achieve a compact instruction list. The single non-trivial equivalence class of the congruence of A_P in Figure 2 has been indicated by a rounded rectangle.

The instruction list

The result of the translation of process execution plan P in Figure 1 is the instruction list L_P in Figure 3. The instruction list is given in an abstract form from which it may be translated to the chosen target language. Basically, the instruction list is an assembler program operating on Boolean variables. There are three groups of variables: input variables (corresponding to process variables), output variables (corresponding to operation variables) and state variables corresponding to the states of the factorized automaton $A_P|_\equiv$. A translation table is provided at the end of the instruction list. The instruction list simulates state transition and out-

```
State_Fragment:
   IF NOT z{0,2} GOTO Endz{0,2}
   z{1} := xA AND xB
   IF z{1} RESET z{0,2}, h0, h2 GOTO Binary_Output
   GOTO Output_Preparation
Endz{0,2}:
   IF NOT z{1} GOTO Endz{1}
   z{0,2} := xC
   h0  := xC
   IF h0 RESET z{1} GOTO Binary_Output
   GOTO Binary_Output
Endz{1}:
Output_Fragment
Output_Preparation:
   IF NOT z{0,2} GOTO EndOutPrep{0,2}
   h0 := NOT xA
   h2 := xA AND NOT xB
EndOutPrep{0,2}:
Binary_Output:
   yR := h0 OR h2
   yS := h2
   yT := z{1}
================================================
                   Automaton AP|=     IL-Identifier
------------------------------------------------
   States:              {0,2}              z{0,2}
                         0                  h0
                           2                   h2
                        {1}                z{1}
------------------------------------------------
Operation variables:  R, S, T           yR, yS, yT,
Process variables :   A, B, C           xA, xB, xC,
------------------------------------------------
```

Figure 3: The translation of the process execution plan in Fig. 1 in an instruction list

put generation of the factorized automaton. The operating system of the open loop programmable control executes the instruction list repeatedly in an infinite loop. The three main parts of the instruction list are:

State fragment: The code fragment before the label `Output_Preparation` serves for determining the subsequent state depending on the current one.

Preparation of the output: The code fragment between the labels `Output_Preparation` and `State_Fragment` is executed if a non-trivial equivalence class is reached from the current state. This is necessary for determining the current state of the original automaton.

Computation of the output: The code fragment starting with the label `Binary_Output`. It is executed in order to compute the output.

Translating a process execution plan

The translation of a process execution plan P into an instruction list consists of two steps. The input of the first step is a representation of the process execution plan and an equivalence relation $\equiv$ on its set of operations. The output of the first and input to the second translation step is a representation of the factorized automaton $A_P|_\equiv$ which allows a rather straightforward generation of the instruction list.

Proving the correctness of TeCOM

In this section we identify the correctness conditions which should be verified and give an outline how to construct the correctness proof.

The verification task

The translator verification task will be discussed following a four step approach leading to a fully verified compiler (Langmaack 1997).

The **first step** consists of the definition of the mapping from the source language (the set of correctly formed process execution plans) to the target language (the set of correctly formed instruction lists), the definition of the semantics of both source and target language (in our case given for both languages by automata) and the definition of the correctness criterion for this translation. The translation is called correct if an instruction list L computed by TeCOM describes, up to renaming of the input and output alphabets, the same word function determined by the initial state as the process execution plan P given as input.

The **second step** is to prove the translation correctness. Here this means to prove the equality of the word functions determined by the initial states of the automata A_P and A_L (up to renaming of alphabets). This formal proof is straightforward.

In the **third step** a host language for the compiler implementation has to be chosen. The authors of TeCOM choose Delphi-Pascal. The verification of the Delphi compiler had to remain beside the scope of our project for obvious capacity reasons. Therefore, any of our correctness results can be formulated only relative to the correctnes of a Delphi-compiler.

The **fourth step** is to prove that the compiler implementation is correct with respect to the chosen host language. For this purpose we define *construction principles* relating a process execution plan P and an instruction list L which are sufficient for the equality of the behavior of the associated automata A_P and A_L. Let us mention some of the construction principles.

(1) There is a bijective relation between the sets of states of the automatons $A_P|_\equiv$ and A_L.

(2) Bijective relations of the input and output alphabets of the automatons $A_P|_\equiv$ and A_L are determined by renamings of the operation variables and process variables.

(3) Every state of the factorized automaton is represented by a code fragment of the state fragment of the resulting instruction list.

```
TIL_Output =
enrich TIL_Output-0 with
 functions
   til_output-build : toutput → til_output ;
 axioms
   ...
   til_output_0 = til_output-build(toutput_0)
  ∧ toutput-occurs(toutput_0, nat_0)
→    til_output-find(til_output_0, nat_0)
 = tdisjunction-intlist2dis
                  (toutput-find(toutput_0, nat_0))
end enrich
```

Figure 4: The specification `TAWL-Ausgabe`

This code fragment represents the set of direct successors of the mentioned state and the state transitions from that state.

(4) Every non-trivial equivalence class of automaton A_P is represented by a code fragment of the output preparation part of the instruction list A_L.

This code fragment allows us to determine which one of its elements has been reached.

(5) For every process variable there exists a code fragment in the output part of the instruction list L.

This code fragment represents the set of all states of the automaton A_P which are labelled by the considered process variable.

Those construction principles serve as correctness assertions which should be satisfied by the implementation. They are subject to a formal verification by the help of the specification and verification tool KIV.

The verification approach using KIV

In the present section we give an overview of the tool-supported formal verification of the implementation against the construction principles.

In order to keep the verification task manageable only a small number of basic data structures, e.g. lists and dictionaries are used. Those data structures are generic and can be actualized in various ways. All theorems proved about a generic data structures are available in every one of its actualization.

Specification TIL_Output (Fig. 4) illustrates this approach. This specification axiomatizes the function til_output-build, which transforms an intermediate representation of the binary output, a value of sort toutput from specification TOutput, into its final form, a value of sort til_output from specification TIL_Output (cf. Figure 6). Values of sorts toutput and til_output are dictionaries (cf. Fig. 5) which have been actualized according to Figure 7.

```
dictionary =
generic specification
 parameter TKey, TEntry target
 sorts TDictionary;
 constants emptydict : TDictionary;
 functions
   adjoin   :  TDictionary × TKey × TEntry
               → TDictionary ;
   find     :  TDictionary × TKey
               → TEntry ;
   ...
 predicates occurs : TDictionary × TKey;
 variables d2, d1, d: TDictionary;
 axioms
   dict generated by emptydict, adjoin;
   find(adjoin(d, ka, a), ka) = a,
   ka ≠ kb → find(adjoin(d, kb, b), ka) = find(d, ka),
   ...
   end generic specification
```

Figure 5: The specification `TDictionary`

```
TIL_Output-0 =
actualize TDictionary
  with TDisjunction, TOutput by morphism
    tdictionary → til_output,
    entry → tdisjunction,
    key → nat
end actualize
```

Figure 6: The specification `TAWL-Ausgabe-0`

The considered axiom of specification TIL_Output asserts that the result of function til_output-build provides for each key nat_0, which occurs in the argument $toutput_0$ of function til_output-build, an entry which has been constructed by function tdisjunction-intlist2dis from the entry toutput-find($toutput_0$, nat_0) which was associated with key nat_0 in the argument $toutput_0$ of til_output-build. Further axioms, not mentioned because of lack of space, assert that in the result of til_output-build occur the same keys as in the argument.

A module implements specification til_output using specification toutput as its import specification. The module provides a procedure for computing the value til_output-build($toutput_0$) by a simple while-loop. Because of lack of space we give some rather statistical information concerning one proof, the proof of the termination of the call of the implementation of function til_output-build. The proof consists of 30 steps and is rather straightforward. A slight generalization of the thesis (1st interaction) prepares the induction proof (2nd interaction). After unfolding the while-loop once (3rd interaction) the system completes the proof auto-

Corresponding parameter sorts in		
generic specification	actual specifications	
TDictionary	TDisjunction	TIL_output-0
key	nat	nat
entry	tdisjunction	til_output

Figure 7: Actualizations of specification TDictionary

matically using 12 simplifier rules which mirror properties of the underlying data structures.

Clearly, much more interesting proofs have been constructed by the KIV developers and other users. Nevertheless, in our opinion it is an important experience that the use of standard techniques at the coding level allows the systematic construction of correctness proofs. It seems to be not realistic to expect that usually correctness proofs are really difficult proofs which need a deep mathematical invention. Rather it should be possible to find them by good (proof) engineering practice supported by strong heuristics of the verifier.

Implemention vs. specification language

TeCOM is part of a larger project which has been implemented in Delphi-Pascal. The implementers prefered to use a homogeneous software platform for implementation. Therefore none of KIV's code generators producing LISP or C^{++} code could be used. Rules have been formulated for a subset of the target language which guarantee the conformity of the Delphi-code with the corresponding KIV-code. The conformity conditions assure that the correctness result obtained for the implementation expressed in KIV-code can be carried over to the Delphi-code.

Results and experiences

After carefully analyzing the code provided by the implementers the verification team decided to re-implement an essential fragment of the code — the generation of the binary processing unit of the instruction list — in order to improve the program structure. This way the verification task became realistic. If program development and verification would have been done in parallel a considerably larger part of the software could have been verified. Verification needs very clearly structured programs. Writing those program is nothing else than good software engineering practice.

One of the essential bugs found during the verification is the incorrect treatment of transitions to states which belong to non-trivial equivalence classes. The PrEP in Figure 1 and its two modifications in Figure 8 show three variants of paths leading from a process operation, i.e. from state 1 labelled by T via process condition C, to the equivalent states 0 or 2. In the

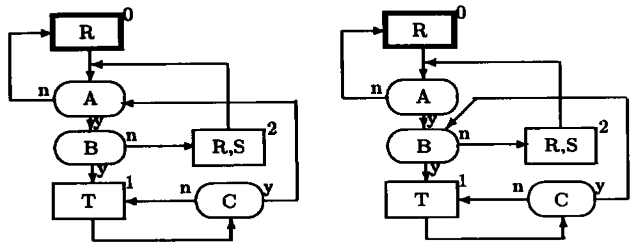

Figure 8: Two modifications of the PrEP

variants shown in Fig. 1 and 8 (left) the outgoing "y"-labelled edge of process condition node C leads either directly to a process operation, i.e. 0, or to the first process condition, i.e. A, within a path connecting the two equivalent states. The original implementation treated those cases correctly. It seems that the third case, illustrated by 8 (right), has been overseen. The solution of the problem which has been chosen in the revised version of TeCOM is to treat case 3 as a "direct" transition analogously to case 2.

Summary and outlook

We described the current state and experiences gained in a verification experiment which is devoted to a development tool for embedded software – the instruction list generator TeCOM. The experiences suggest that verification of real programs becomes feasible. At least critical fragments are in the range of present verifiers.

Acknowledgements

Thanks to E. Alder and A. Pretschner for their determination to verify TeCOM, to I. Klein for his cooperation, and to the referees for their helpful remarks.

References

Langmaack, H. 1997. Contribution to Goodenough's and Gerhart's Theory of Software Testing and Verification: Relation between Strong Compiler Test and Compiler Implementation Verification. *Foundations of Computer Science: Potential-Theory-Cognition. LNCS* 1337:321–335.

Pnueli, A.; Shtrichman, O.; and Siegel, M. 1998. Translation validation for synchronous languages. *Lecture Notes in Computer Science* 1443:235–246.

Reif, W., and Schellhorn, G. 1997. Theorem proving in large theories. In *Proceedings of the 14-th International Conference on Automated Deduction (CADE-14)*. Springer-Verlag. LNAI.

Reif, W. 1992. Verification of large software systems. *Lecture Notes in Computer Science* 652:241–256.