Reverse Engineering of Assembler Programs: A Model-Based Approach and its Logical Basis

Tom Lake and Tim Blanchard,
InterGlossa Ltd., Reading, UK
Tel: +44 1734 561919
email: {Tom.Lake,Tim.Blanchard}@glossa.co.uk

Abstract

The REAP project at InterGlossa is developing tools to support maintenance and reverse engineering of assembly language programs, concentrating on well-engineered hand-coded programs.

Abstraction of assembly programs takes place in the context of a selected 'engineering model' which includes the definition of the instruction set semantics but also constraints on the programs similar to those found in ABIs. The process of translation takes the form of a large-scale inductive demonstration that the program meets the constraints of the 'engineering model' as the translated abstraction is produced.

An engineer's interface makes this manifest to the engineer supervising the translation.

This approach can in principle handle programs whose models include a disciplined use of code self-modification or dynamic register bank switching. As intermediate language for the major analyses involved we use a representation based on the XANDF X/Open standard originating from the UK Defence Research Agency. XANDF is a standard for architecture neutral program representation which will permit support for analyses of portability. Concurrency is not yet covered but recent advances show how XANDF can be extended to encompass concurrency and distribution[10].

We illustrate the effectiveness of the tools with examples taken from live Intel 8051 and Zilog Z80 systems.

1. Introduction

The REAP project at InterGlossa[1] is constructing tools for the predominantly automatic reverse engineering of assembler programs. Our main targets are micro-controllers, where practice is still shifting from use of assemblers to high level languages for embedded systems. The particular nature of these processors and the variety of programming styles in use has prompted us to pay particular attention to the translation from the machine code or binary coding to intermediate language. Having expressed the meaning in an intermediate language we have found automatic transformations satisfactory for translation to C so far.

[1]This work is partly supported by the UK Department of Trade and Industry under the SMART programme.

The next section gives our objectives; section 3 briefly discusses related work; section 4 discusses our view of translation and defines terms for the rest of the paper; in section 5 we describe the reverse engineering process that ensues; section 6 gives a brief introduction to the XANDF intermediate language used; section 7 gives a practical example; section 8 discusses the current state and prospects of the REAP project; section 9 is a summary.

2. Objectives

The REAP tools are for translation and also for production of maintenance documentation (including interactive representations) of assembler code and binary systems, especially for micro-controllers, with engineer interaction supplementing automatic transformations and translations.

We have seen many examples of long-lived codes, e.g. in aerospace and defence applications which are currently maintained from assembler source and where translation can ease the maintenance task or provide a more familiar representation of the system for study. Reverse engineering for software maintenance is our primary objective in the REAP project.

Another objective in reverse engineering is that we hope to support safety arguments via reverse translation. We expect to provide sound, though not formal, arguments that the behaviour of the system has been captured. This is the reason for paying particular attention to the first steps of the translation. At present, safety arguments for micro-controller based systems have to be based on the compilation tool chain. Our REAP tools offer an alternative approach to certifiers wishing to satisfy themselves of certain simple properties of a code from its final installed form. While we would not propose such methods as the sole safety argument, we do believe that their use offers useful additional safeguards, especially with respect to the reliability of the tool chain.

Consider the safety argument for e.g. the confinement of a robot arm to a certain spatial region by, say, a system of sensors and power control. Such an argument could be made independent of large parts of the system. We believe that a similar approach is within reach by reverse engineering although a great deal of work is necessary before such arguments can carry real weight. At present they can be simply useful additional evidence of the safety of the implementation and the compilation tool chain in a given case.

We also expect to be able to provide portable translations of such code or to indicate where portability breaks down. The architecture neutral representation of store offered by the XANDF intermediate format is essential to achieving this objective.

3. Related Work

Other authors have discussed translation and understanding of assembler programs. Decompilation is a form of translation where a restricted and known set of idioms can appear in the source system. Decompilation for Intel x86 processors to C is discussed in [3] and a more general framework is given by Bowen [2]. For translation of possibly hand-written assembler systems the best-known work is that of Ward and Bennett [5] who describe their translation of assembler programs briefly. In some cases the authors are most concerned with the induction of design level information from the code rather than the deduction of its behaviour. Breuer and Lano [6] discuss the induction of objects from COBOL and Fortran in this sense. The maintenance system described in [8] is more relevant to our current concerns.

Ward and Bennett discuss assembler translation (in the context of the IBM 370 instruction set) in the following terms, with which we heartily concur:

"It is theoretically possible to have a perfect model of the language which correctly captures the behaviour of all assembler programs. Certain features of Assembler, such as branching to register addresses, self-modifying code and so on, would imply that such a model would have to record the entire state of the machine, including all registers, memory, disk space, and external devices and "interpret" this state as each instruction is executed. Unfortunately, such a model is useless for inverse engineering purposes since such trivial changes as deleting a NOP instruction, or changing the load address of a module, can in theory change the behaviour of a program."

Ward and Bennett then discuss three types of modelling of assembler programs which are apparently applied to different constructs during translation and which they judge suitable for all practical purposes:

1. Complete modelling: they appear to emphasise the explicit treatment of flag registers and the treatment of the memory as an array here.

2. Partial modelling: The treatment of dynamic branches is discussed under this head, with a set of labels (the set of possible targets) being associated, possibly by heuristic means, with dynamic branches. With this approach the dynamic aspect can be transferred from the jump target to a conventional case or choice structure.

3. Excluded or minimally modelled: e.g. self-modifying code, only special cases are recognised by the translator.

This very general framework coincides with our approach, although we will show that fully automatic methods can supercede heuristics in many cases, by iteration through the analysis.

Very little has been written about the accuracy of inference of data structures. In a private communication Ward indicates that the identification of distinct data items "falls down where the assembler program fiddles with base registers in such a way as to create aliasing between otherwise distinct symbolic names".

4. Translating Machine Code

As mentioned above, we expect each translation to be associated with an idealised model of the processor, simpler than the full behaviour. Aspects of behaviour which are not exercised in the particular system in question can be described in the notation of CSP[4] as `chaos` - in a given case we need not be concerned with the precise description of subsequent events once the bounds of our model are over-stepped.

In general, a range of models will be deployed for a given processor depending on the features of the processor which have been used in the system. For example, for the Intel 8051 family register banks can be switched dynamically, but may not be in a particular case, or again the registers can be addressed by indirection, but may not be in a particular case.

Suppose that system `S` does not exercise a particular feature `F`, for example code modification. The absence of `F` will be an invariant property, say `inv-F` of the system. A full description of the processor may be replaced by a description which leaves out the feature or describes the behaviour on its invocation as `chaos`. This may considerably simplify the description. For example in the absence of code modification the description need no longer include the binary decoding of instructions, which may be applied statically to the code at the outset. The true processor description is clearly a refinement of the particular description since `chaos` is replaced by a particular behaviour. However, the notion of the inverse of refinement does not precisely capture the nature of our description. Given the code for `S` and its starting and environmental conditions or a partial description of them we hope to prove the invariance of `inv-F`. Usually the processor semantics will allow the effect of a given instruction to be deduced from the state at the start and will specify the state at the end of the instruction. The truth of `inv-F` on the initial state will allow the simpler semantics to be used. If we can then prove that `inv-F` is maintained we have the induction step for proving the invariance of `inv-F` from its initial truth and properties of the rest of the system.

Suppose we are trying to prove that there are no register bank switches in a program. Initially we can assume that there were none up to the nth execution step and translate the instructions accordingly. (Otherwise we would have had to represent the processor register banks more explicitly in translation.)

Since the desired invariant is clearly true at the start we have to prove that it is preserved by the instructions of the nth step of the program from the premise that it is valid for the instructions up to the $n-1$th.

It is this hidden inductive proof that constitutes the logical basis of reverse engineering.

We describe as an `engineering model` the simplified processor description and the invariants that guarantee its correctness. The availability of a tractable engineering model is usually the result of the discipline applied during the systems design and implementation. When we have verified that the engineering model applies throughout the operation of the system we have already made a big step towards recovering the design of the system, albeit that part which related to its mapping to the processor.

We can thus see recovery of design information as the recovery of properties which guarantee the system's confinement to a state space in which its dynamics and properties are more simply described.

In most cases we cannot expect to demonstrate the induction step from a direct translation of the processor semantics, without a good deal of transformation and preparation. For this reason, we expect to carry the invariants which characterise an engineering model forwards through the analysis process as *proof obligations* to be discharged either automatically by analysis or by the explicit intervention of human agency.

Our methods are not formal. In order to produce a formal argument for the correctness of a description of behaviour derived by transformations it is necessary to demonstrate both that the transformations are in accordance with the semantics and that the transformation engine applies the transformations correctly. While authors have reported the former property [5] we do not know of systems for which both properties are formalised.

4.1. Analyses and Proof Obligations

The analyses which lead to discharge of proof obligations are carried out subject to these conditions. We need to be sure that we are not merely basing an argument on contradiction. In fact conventional data and control flow analyses used to support these demonstrations are satisfactory. Logical problems would arise if we tried to 'run the semantics backwards', that is, argue from operation results to operation inputs. As long as we avoid this both backwards and forwards analyses can contribute soundly to the overall inductive argument. We hope to formalise this argument in the future.

5. The REAP Reverse Engineering Process

We assume that the program to be reverse engineered has been produced in an engineering environment. By this we generally mean that certain invariants are preserved during execution with the result that the interpretation of instruction semantics in the context of the program is considerably simplified.

In this section we identify the engineering model for the program. As we process the program, we can check that the program conforms to the proposed engineering model (and iterate if not). We then present the REAP engineer's interface (see section 5.1).

Our choices of engineering model must be strong enough to allow the program to be translated into our chosen intermediate format. The essential characteristics involved are discussed in section 6.2.

When the program is known to conform to a given engineering model (either through assertions about the program by the engineer or by analysis of the source code) the program can be translated into one or more intermediate forms. Actually, the program need only partially be shown to conform to the model, while outstanding properties can be carried forward as proof obligations to be discharged later. Alternatively, a partial translation may be possible with partial knowledge, and further knowledge, derived from analysis of the translated form of the program, can be fed back to improve the partial translation, in an iterative scheme.

The XANDF-based intermediate format is referred to in the following as *IF*. In IF translation we provide an IF equivalent for each assembler instruction, subject to the validity of the engineering model for the given program. The registers must be modelled as globals but stack locations can be treated as locals provided that stack modelling has shown that their lifetimes are properly nested within those of routines.

Manipulation of IF allows us to further verify the conformance of the program to the engineer's model. We also reduce program size, and simplify the code, through transformations such as eliminating dead expressions and tests, idiom recognition and in-lining. Given conformance to the model we can produce a high level language translation or output the IF representation of the program for later code generation or further analysis. We describe this process in section 7.

In section 8 we present an example of the REAP tool used on a live Intel 8051 system, illustrated by fragments drawn from the original source text.

Some of the properties that make up an engineering model are almost always inter-procedural in nature, e.g. the correct nesting of call/return pairs. It cannot be expected that translation can be performed effectively on a procedure by procedure basis alone.

A useful example of translation is given by the return instruction in a typical assembly language. The return instruction necessarily specifies a data-dependent transfer of control. To substitute the most general semantics of the return instruction into a program is to invite chaos. Instead the return instruction should only be translated in contexts where its use is constrained. In the most common cases the program conforms to the constraint that the program returns to the instruction following the matching call, with correctly nested call/return pairs. This can be established by careful modelling of the call/return stack over the control flow of the program. An assembler program cannot be translated into the IF format unless its call structure has been analysed and can be modelled using the conventional calls and returns. Such modelling itself depends on the absence of overwriting or stack overflow of the return stack from the program's execution. Verification of this can be deferred until after translation to IF, when a target independent analysis can be used, since an IF representation infers that the original stack structure was fairly regular.

In summary, we carry out the translation in the context of a set of coupled global properties of the program which are verified at various stages of the translation.

5.1. Engineer's Interface

Figure 1 shows the entry interface of the REAP translator at the start of the translation process from Intel 8051 assembler to C. The model selected is expressed by a number of hypotheses embodying global properties of the system. There is a status associated with each hypothesis (PASSED, FAILED or UNRESOLVED). Hypotheses are grouped by phases where AS indicates hypotheses that must be substantiated for the assembler representation of the program; and IF those for substantiation in the intermediate format.

```
REAP HYPOTHESIS BROWSER (hypos)------------------------------
  1   SVOL AS     PASSED   Stack/stack pointer not volatile
  2    RTI AS     PASSED   Instruction after subroutine call
  3  SPOVW AS     PASSED   Stack pointer not overwritten
  4   REGB AS     PASSED   Register bank constant
  5   DTOK AS     PASSED   Data/return addresses trusty
  6   MVPC AS     PASSED   MOVC from PC unambiguous
  7   DPOW AS     PASSED   DPTR manipulates data labels
  8   SOVF IF UNRESOLVED   Stack doesn´t underflow or overflow
  9   SOVW IF UNRESOLVED   Stack not overwritten or read
 10   INTR IF UNRESOLVED   Scope and side effect of interrupts
 11   ROVW IF UNRESOLVED   No register overwriting by indirection
 12   DIND IF UNRESOLVED   Data is relocatable
```

Figure 1. REAP hypothesis browser (8051)

A number of analyses are available to the engineer who can also browse the code and any annotation to the code made by the translator.

As the engineer proceeds iteratively with the analyses and examinations of the points at which they show inconsistency with the hypotheses of the selected model, confidence in the orderly behaviour of the system can be established.

A certain number of analyses are performed directly on the assembly code. When a certain degree of structure has been established, particularly order in the call/return nesting, it is possible to translate the assembly code to the intermediate format, IF, which is based on XANDF, for an easier and more portable approach to program translation.

Some aspects of translation, such as translation of dynamic jumps or of pure binary, require, under Ward and Bennett's 'partial modelling' approach, that range information be input to the translations. In fact such information can be collected, albeit laboriously, by a grand iteration of the whole translation process, feeding new ranges discovered in the IF analyses back to the start of translation.

6. Introduction to XANDF

XANDF is an intermediate language developed by DRA at Malvern and adopted as its choice of ANDF (Architecture Neutral Distribution Format) by the Open Software Foundation[7]. XANDF is being standardised by The Open Group (X/Open)[1].

6.1. Purpose and Use of XANDF

XANDF is an intermediate language independent of source language and target architecture developed by the Open Systems Software Group at DRA Malvern. XANDF supports modular compiler development, permitting detailed checking of the target independence of programs and the correctness of their use of standard APIs. XANDF has been considered as a strategic technology for the OMI project cluster in the ESPRIT program. More generally, XANDF provides a standard semantic medium for programs at the implementation level.

XANDF is a tree-structured language with special features for portability. It preserves more program structure than low level IRs such as the RTL of *gcc*, but has no syntactic sugar and a weaker type system than typical high level languages (HLLs). Explicitly written XANDF appears verbose, but is implemented with powerful compression. In quantitative studies XANDF compares well in measures of performance and size with standard compilation.

A *producer* converts an HLL into a compact binary representation of XANDF. The fundamental unit of XANDF is the *capsule* which contains definitions and declarations of procedures, variables and tokens (see below). A capsule can export these declarations and definitions by binding them to external names. A *XANDF linker* allows capsules to be bound together using these names.

Even portable programs possess machine and operating system dependencies such as the implementations of `C` types `int` and `FILE`. XANDF possesses a mechanism to defer the representation of target dependent program components using parameterised place-holders called *tokens*. XANDF provides a linker that can be used to include parameterised definitions for tokens. These expand place-holders to satisfy target dependencies.

An *installer* converts a fully linked XANDF file to target specific object code which is then linked with required system object libraries using the target system linker to produce an executable. Producers have been or are being developed for C, Fortran 77, Ada95, Dylan and C++ along with installers for 80x86, MIPS, Alpha, PowerPC, HP/PA and ARM processors.

6.2. Expressiveness of XANDF

XANDF's description of store shapes, alignments and accesses allows it to give an architecture neutral description of store without losing the capability of representing C completely. XANDF views store as segmented into independently created chunks known as *original spaces*. It is not permitted to take the difference of pointers in different original spaces or, by XANDF pointer operations, to derive a pointer in one original space from a pointer in another[2].

XANDF describes all procedures as having single entry and returning to their caller. This means that additional conventions have to be adopted to represent the multi-entry and alternate return procedures used in some programs.

6.3. Translation to XANDF

When we translate from assembly language to XANDF, in the context of the engineering model which has been assumed, we do so for the particular data representation appropriate for the processor and system in question. To achieve an efficient architecture neutral translation we can proceed to demonstrate the representation independence of the program in XANDF. Alternatively, we can transform the program so that the required data representation is bound in, directly modelling data for the processor in question. This latter approach is likely to produce slower code because coercions to the processor's register formats has to be introduced in some places.

Demonstrating representation independence depends on establishing types for the program variables.

Our IF contains constructs corresponding to most of those in XANDF apart from constructs concerned with capsule level representation and install time computation.

Once all AS phase hypotheses have been validated we may translate the assembler program into IF.

The registers and various special memory locations are translated into variables wherever appropriate. We generally represent separately the smallest units in which registers can be addressed, e.g. bytes for the Z80 16-bit registers in order to allow liveness analysis the most freedom and re-unite the pieces in a later analysis. The status registers are generally translated to separate boolean global variables. The literals in the code stream become static constants.

The code translation is a semantic equivalent of the assembler instructions subject to the constraints of the engineering model selected. So, for example, the Intel 8051 instruction

```
MOV    A, #4
```

would be translated to:

[2]The environment pointers used to provide access to outer scopes are exceptional but there use is strongly constrained to avoid violating the independence of original spaces.

```
sequence
( assign(obtain_tag(A),
          make_int(~unsigned_byte(),make_signed_nat(false,4))),
  assign(obtain_tag(parity),make_parity(contents(A)))
)
```

A and parity are allocations of data spaces in the representation of the Intel 8051 memory model. obtain_tag returns the pointer to such a data space; contents retrieves the contents at that pointer. make_parity is a token standing for determination of parity.

Micro-controllers often provide optimised functions that cannot be recognised by their compilers from a source coding of their semantics. Our translation for functions like make_parity is an identifier which can be treated as a C macro but for which the analyses have respected the semantics in carrying out transformations.

7. Analysing IF

This form of translation leads to a far more compact intermediate representation than the use of a full processor description but one which is still verbose compared to the original. We therefore introduce a number of analyses to reduce the size of the code. These eliminate dead global variables, dead expressions and dead jumps using inter-procedural analyses. We also remove unnecessary movement of data. We identify multi-word arithmetic operations and shifts and group smaller variables which are used in common multi-word operations. This analysis is inevitably to some extent heuristic and is similar in some respects to the more general class abstraction processes used in identifying objects in code.

The code is also simplified by a novel intra-procedural control flow normalisation algorithm for IF, the principles of which are described in [9].

The obligation to verify the remaining parts of the engineering model should be discharged in this representation. We are not yet far enough advanced in the analysis of data representations to support this effectively by automatic tools. Analysis of data representations would also allow information on portability and the non-portable constructs to be reported.

At this point an effective translation to C can be produced. The method involves transformations of the IF to a normal form easily mapped to C control constructs and then direct translation.

8. Example

In this section, we present an extended example of the use of the REAP reverse engineering tool on an Intel 8051 Teletext Control System, provided by GEC Plessey Semi-Conductors. The system has 7000 lines of assembler source, which assembles to 14K of Intel 8051 code. We also show translations for Z80 as drawn from a live aerospace system.

8.1. 8051 Assembly Language Representation

The source code conforms to our model for Intel 8051 after engineer's assertions: the stack is not misused (the stack pointer is not reset, subroutines return to the point of call, etc); the register bank is identifiable; the data pointer (DPTR) is used only for referring to static data; and the static data accessed offset from the PC is unambiguously identifiable. There are a few places in the code at which the engineer must place an assertion to add information. We illustrate with the treatment of a cavalier use of returns in a routine, which we term *unconventional returns*.

This case affords a simplification of the structure in translation to IF. The subroutine chkint manipulates the call stack by popping off the return address and issuing a return, effectively performing a return from its caller. Our control flow analysis can support such unconventional exits from subroutines. A problem arises when the return from the routine, via the instruction labelled with nochk, is shared between both the normal exit path from the routine and the double return path. The analysis is flexible enough about the order of instructions to allow it to deal with patched code and hence has no real way of knowing whether the instruction after popping the return address from the stack belongs to the same routine or to its caller. Our default assumption is that it belongs to the caller. But the use of a common instruction with the normal return path causes the assembler routine to be considered as part of the IF procedure including its caller on translation to IF. A human engineer can easily determine what is intended in simple cases, but this judgement is based on heuristic knowledge and common sense - which our analysers as yet do not possess.

An assertion of the form AssertReturnOK by the engineer prevents the merging of the routine chkint with its callers into a single IF procedure. AssertReturnOK acts like an artificial return instruction, ensuring that there is no fall through between the chkint and nochk code blocks. Consequentially, the called routines and calling routines are not merged by the control flow analysis.

8.2. Intermediate Code Representation

Once all hypotheses for the AS phase requirements have been satisfied we are free to translate to IF. The

```
chkint: mov a,tflag
    anl a,#0FH
    jz  nochk       ;TOP acquisition?....
    jb  ie.0,nochk  ;EVENT?...
;
    setb     timflag.7   ;Abandon flag
    pop acc      ;Yes, balance stack
    pop acc      ;..and return from previous
                 ; interrupt
        [ AssertReturnOK ]
nochk:  ret
```

Figure 2. 8051 Assembler Example

```
TdfExp_Label(: chkint!8,TdfMake_Proc(
  TdfInteger(TdfVar_Width(TdfFalse,32),2849),[],N/A,
  TdfLabelled(TdfGoto(TdfMake_Label(: chkint!8),2851),
  [ TdfExp_Label(: chkint!8,TdfSequence(
    [ TdfSequence(
      [ TdfAssign(TdfObtain_Tag(_A,mov A tflag,2856),
           TdfContents(TdfInteger(_,_,_),
             TdfObtain_Tag(_,_,_,_),mov A tflag,2857),
           mov A tflag,2855),
        TdfAssign(TdfObtain_Tag(_PARITY,mov A tflag,2861),
           TdfParity(TdfContents(_,_,_,_,_),mov A tflag,2862),
           mov A tflag,2860)],mov A tflag,2854),
        TdfSequence([ TdfAssign(TdfObtain_Tag(_A,anl A #15,2868),
          TdfAnd(TdfContents(_,_,_,_,_),
              TdfMake_Int(_,_,_,_,_),anl A #15,2869),
          anl A #15,2867),
          TdfAssign(TdfObtain_Tag(_PARITY,anl A #15,2875),
             TdfParity(TdfContents(_,_,_,_,_),anl A #15,2876),
           anl A #15,2874)],anl A #15,2866),
        TdfInteger_Test(TdfNot_Equal,TdfMake_Label(: nochk!8),
          TdfContents(TdfInteger(TdfVar_Width(TdfFalse,8),2882),
tdf_browse>
```

Figure 3. IF tree browser

REAP tool suite includes an interactive IF tree walker, so that the engineer can determine the correspondence between the IF and the original assembler. Figure 3 given a typical output from the tool. Note that the tool maintains links to the original assembler by including the original assembler instructions in the IF primitives derived from them.

We can now validate the IF hypotheses; and reduce the complexity of the program through various analyses. The program fragment given in figure 3 could be reduced to the code shown in figure 4:

At this point, we can translate the program to C. The resulting C for the subroutine **chkint** is given in figure 5.

At the two return points from the subroutine, there is a differing ret0 field value: one signifies an unconventional return; the caller will test this value and perform a subsequent return as a result, thereby modelling the

```
TdfMake_Proc(TdfStructure(
      [ TdfInteger(TdfVar_Width(TdfFalse,32),4),
        TdfInteger(TdfVar_Width(TdfFalse,8),5)],3),[],N/A,
  TdfVariable(N/A,TdfMake_Tag(30),
    TdfMake_Value(TdfInteger(
          TdfVar_Width(TdfFalse,8),8),: chkint!0,7),
    TdfSequence([ TdfMake_Top(10),
      TdfAssign(TdfObtain_Tag(30,anl A #15,12),
        TdfAnd(TdfContents(
          TdfInteger(TdfVar_Width(TdfFalse,8),15),
            TdfObtain_Tag(tflag,mov A tflag,16),mov A tflag,14),
          TdfMake_Int(TdfVar_Width(TdfFalse,8),15,anl A #15,17),
          anl A #15,13),anl A #15,11),
      TdfConditional(TdfSequence(
      [
        TdfInteger_Test(TdfNot_Equal,: nochk!0(: nochk!0),
          TdfContents(TdfInteger(TdfVar_Width(TdfFalse,8),22),
            TdfObtain_Tag(30,jz nochk!0,23),jz nochk!0,21),
          TdfMake_Int(TdfVar_Width(TdfFalse,8),0,jz nochk!0,24),
          jz nochk!0,20),
        TdfConditional(TdfSequence(
        [
          TdfInteger_Test(TdfEqual,LABEL 56,
            TdfChange_Bitfield_To_Int(
```

Figure 4. IF program fragment, post analysis

```
struct_chkint_ chkint_()
{
  BYTE _A_30;
  /* TOP acquisition?....
  EVENT?...

  Abandon flag
  Yes, balance stack
   */
  _A_30 = (*tflag_ & 15);
  if (_A_30 != 0)
  {
    if (BIT_VALUE0(*ie_) == 0)
    {
      BIT_SET7(*timflag_) = 1;
      /* ..and return from previous
         interrupt
       */
      chkint__return (1,_A_30);
    }
  }
nochk_: ;
  chkint__return (0,_A_30);
}/* chkint_ */
```

Figure 5. C program fragment, post analysis

```
A_TO_HL: LD  H,A
         LD  L,0
         SRL H
         RR  L
         SRL H
         RR  L
         SET 7,H
         RET

BOUND:   BIT 7,D
         JR  Z,PNV
         BIT 6,D
         JR  Z,BNG
         BIT 5,D
         JR  Z,BNG
         RET
BNG:     LD  A, 80H
         RET
PNV:     BIT 6,D
         JR  NZ,BGS
         BIT 5,D
         JR  NZ,BGS
         RET
BGS:     LD  A, 7FH
         RET
```

Figure 6. Z80 Assembler Example

original double return control flow construct.

8.3. Z80 Assembly Language Representation

Two small utility routines from a Z80 aerospace application are shown - first in assembler and then in their C translation. We see how the inter-procedural analysis gives a clear indication of the dataflow at calls and returns.

9. Conclusions and prospects

The suitability and orthogonality of the constructs of XANDF and the availability of machine readable syntax for it has allowed us to proceed rapidly in constructing transformations and analyses based on XANDF. This is a key advantage in using XANDF as an intermediate form for reverse engineering. We make use of the publicly available definition for XANDF and have produced our own toolsets for its manipulation. A long term advantage in using XANDF is it gives us the ability to interoperate with other tool chains, potentially

```
SHORT A_TO_HL_(BYTE _A_)
{
    SHORT           _H_L_69;
    _H_L_69 = ((((SHORT) _A_) << 8) >> 2);
    BIT7((BYTE_I(_H_L_69, 1))) = 1;
    return (_H_L_69);
}               /* A_TO_HL_ */

BYTE BOUND_(BYTE _A_, BYTE _D_)
{
    if (BIT7(_D_) == 1)
    {
      if (BIT6(_D_) == 1)
      {
        if (BIT5(_D_) == 1)
        {
        return (_A_);
        }
      }
    BNG_: ;
      return (128);
    }
  PNV_: ;
    if (BIT6(_D_) == 0)
    {
      if (BIT5(_D_) == 0)
      {
        return (_A_);
      }
    }
    return (127);
}               /* BOUND_ */
```

Figure 7. C program fragment, post analysis

allowing reverse engineering tools to become a standard facility in a developer's toolset.

We now have automated translations to C from Z80 and 8051 code, as our examples demonstrate, and the means to introduce new processor architectures with limited effort. The C code produced is good quality and clearly exhibits the intent of the original programmers. We can deal with theoretically any size of well engineered code conforming to the currently fairly simple engineering models that we support (i.e. no dynamic jumps, a single stack etc). Practically, we have been automatically translating medium sized embedded applications of about 10000 lines.

Development of the REAP tools is on-going. We have a tool chain which supports translation from 8051 and Z80 to C via XANDF. We have a number of IF analyses, related to code restructuring and code elimination. We are adding analyses connected to data independence and original spaces. Of particular interest will be analysis which supports the creation of portable code by typing constructs derived from assembler code.

We also expect to be using our inductive framework to give evidence of the behaviour of critical codes.

References

[1] The XANDF guide. Technical Report G508, X/OPEN, 1996.

[2] J.P. Bowen. From Programs to Object Code and Back Again. *J. Software Maint.: Research and Practice*, 5(4), 1993.

[3] C.Cifuentes and K.Gough. Decompilation of Binary Programs. *Software - Practice and Experience*, 25(7):811–829, 1995.

[4] C. A. R. Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[5] M.P.Ward and K.H.Bennett. Formal Methods for Legacy Systems. *J. Software Maint.: Research and Practice*, 7(3):203–220, 1995.

[6] P.T.Breuer and K. Lano. Creating Specifications from Code. *J. Software Maint.: Research and Practice*, 3(3):145–162, 1991.

[7] OSF research institute. ANDF Technology Source Book. Technical report, OSF, Cambridge, MA, 1993.

[8] S.Chen, K.G.Heisler, W.T.Tsai, X.Chen, and E.Leung. A Model for Assembly Program Maintenance. *J. Software Maint.: Research and Practice*, 2:3–32, 1990.

[9] B. Sloman, T. Lake, and S. Williams. Identifying loops in flowgraphs. report SEG/GN/94/1, Dept. of Computer Science, Reading University, Dept. of Computer Science, University of Reading, PO Box 225, Whiteknights, Reading, RG6 2AY, UK, March 1994.

[10] Ben Sloman and Tom Lake. Featherweight threads and ANDF compilation of concurrency. In *Proceedings of EUROPAR'95*. Springer-Verlag, 1995.