Portable Code for Complex Critical Systems†

N.C. Audsley, I.J. Bate and A. Grigg
Real-Time Systems Group, Dept. of Computer Science, University of York, UK
{neil,ijb,alan}@cs.york.ac.uk

Abstract

A common requirement on future safety-critical systems is to support hardware interchangeability. In this paper, work sponsored by British Aerospace Military Aircraft and Aerostructures is reported which addresses this issue. Interchangeability is motivated by the need to perform technology upgrades within a system when components become obsolete – hardware computer components are often superseded within a few years, whereas the total system may have a lifetime of decades. Hardware interchangeability, implies that software needs to be moved to a new platform and execute with minimal rework or disturbance to the rest of the system. Movement of software to a new (different) hardware platform is a difficult proposition without rework, e.g. re-compilation of the software. For safety-critical systems, the rework could also include test, analysis, verification and validation efforts, adding to the overall cost of the change. In this paper, the problem of movement of software to a new platform is considered, within the critical systems domain. The solution to the problem proposed in this paper is Portable Code *(PC) whereby source code is compiled to an intermediate portable form that can then be instantiated to, or directly executed by, any platform. This solution can remove much or all of the rework costs involved in moving software to a new platform, thus substantially reducing system lifecycle costs. The contributions of this paper are twofold. Firstly, a PC suitable for critical systems is described. This is a subset of an existing PC, namely ANDF (Architecture Neutral Distribution Format). Secondly, a compilation approach suitable for PC is described. This has the benefit of being traceable, thus increasing the ability to perform static analysis at the PC level, in turn increasing the ability to move the code to a new platform without invalidating analysis and other evidence gathered for the original platform.*

† Supported by BAe via Centre of Excellence contracts SPO-S-0117383 and SPO-S-0196010 with the University of York.

1. Introduction

Within the aerospace domain, Integrated Modular Avionic (IMA) systems aim to reduce the problems of upgrade by ensuring the interchangeability of both hardware and software modules [13, 21, 20, 8]. In this paper, work sponsored by British Aerospace Military Aircraft and Aerostructures is reported which addresses the interchangeability issue. Interchangeability provides "like-for-like" (i.e. common specification and interface) interchange of hardware modules. The new hardware module may well have a different processor. The motivation for interchangeability is the reduction of lifecycle costs, through mitigating against hardware obsolescence. A further key advantage is the ability to perform continual evolutionary upgrades as new technologies become available, rather than delaying improvements to form batched mid-life updates.

From a software perspective, interchangeability requires that software should run on a new hardware module without change. It could be argued that this merely requires re-compilation for the new target. However, this is not sufficient for safety-critical systems, due to:

Limited technology transparency – the software requirements, design and/or implementation may contain hardware specific elements (e.g. a busy loop to effect a wait for a specific time) [4] to speed-up development, or obtain greater system performance. Such software requirements / design / implementation have to be reworked prior to moving to a new platform.

Additional validation and verification – if a software component changes or is re-compiled, significant costs are involved. These include development of the component to an appropriate safety-critical standard (e.g. DO178-B [14] and DefStan 00-55 [19] for civil and military aircraft respectively), cost of verification and validation of the component and/or system to the appropriate regulatory authority (e.g. FAA, CAA for civil aircraft). The costs of verification and validation of the system can be a significant proportion (>50%) of the total costs [10, 12].

Clearly, to reduce costs of software movement, it should be developed once for *all* platforms. Firstly, hardware dependencies in the source should be removed. This can be achieved by designing and implementing the software in a technology transparent manner, e.g. by utilising standard operating system interfaces rather than directly accessing the hardware from the application [4]. Secondly, target independent compilation should be used.

The focus of this paper is upon software portability from the compilation perspective, concentrating upon a method for achieving software interchangeability for IMA systems. The solution presented is *Portable Code* (PC). This is a technology independent (i.e. target-independent) code to which source code written in a high-level language can be compiled, effectively splitting the compilation into *high-level*, from source code to PC, and *low-level*, from PC to target-specific code. In the same manner that source code is written in a high-level language, PC is expressed in an *Intermediate Language* (IL). For example, ANDF (Architecture Neutral Distribution Format) [22, 23] is an IL that has defined syntax and semantics. Note that the low-level compilation could occur on the host, similar to conventional compilation, or on the target, as part of system initialisation or dynamically "on-the-fly", either using hardware translation or software interpretation. The PC approach is illustrated in Figure 1.

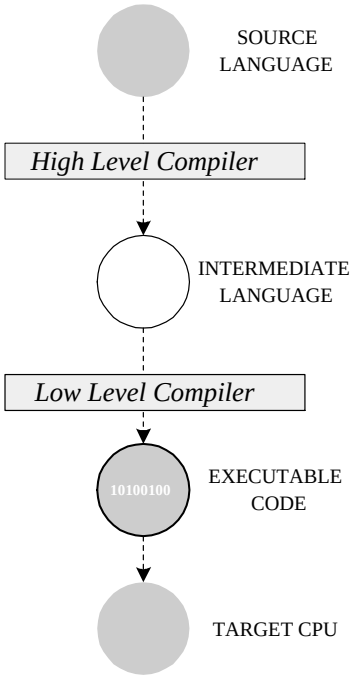

Figure 1. The Two-Stage Compilation of Portable Code.

Advantages of PC for critical systems include:

Increased analytical evidence – (most) analytical evidence gathered on software for critical system verification [2] (e.g. data-flow analysis) can also be gathered at the PC level, increasing confidence in the system.

Reduced cost of verification – PC increases the ability to re-use associated test and verification evidence, as some evidence can be gathered from the PC, which does not need to be re-gathered on platform change.

Reduced cost of software tools – PC uses a standard compiler, independent of the eventual target hardware.

Ease of hardware interchangeability – PC greatly increases the ability to move software to a different platform with minimal rework and minimal disturbance to the rest of the system (context: in-service/on-ground replacement of hardware modules).

Ease of reconfiguration – to meet a requirement for IMA systems [15], PC enables dynamic reconfiguration at the software level, i.e. to mitigate against failures by moving software modules (whilst the system is running) to non-failed nodes.

Each of the above capabilities could serve to reduce the lifecycle cost of a critical system.

Initially, in section 2 this paper reviews a generic PC solution for critical systems, as given in [3]. Then, a background to ANDF is given is section 3. In section 4, an IL suitable for critical systems is described. Section 5 gives a compilation approach is given. This has the additional benefit of providing traceable translation of source code to PC and then to target specific binary. This enables analysis to occur on the PC that can be related back to that performed upon source code and target code, so providing increased evidence for validation and verification purposes, as often required for critical systems [14, 19]. Finally, in section 5, conclusions are offered.

2. Portable Code Solution Overview

Critical systems impose specific constraints on many aspects of development and implementation, since often their failure can have catastrophic effects, in terms of human life or business cost. Usually, some form of *Verification and Validation* (V&V) of the system is required prior to the system being used. For example, an aircraft has to be certificated as suitable for flight prior to entering service.

V&V for critical systems includes a sound development process, dynamic testing and static analysis. The latter can be complex for critical systems. For safety-critical systems

this can incorporate formal proofs, data-flow analysis, timing analysis, object-code verification etc. Typical analysis techniques used for critical systems are outlined in [2].

For use in critical systems, it is important that any PC solution does not compromise the ability to validate the overall system to the required level (i.e. safety-critical, mission-critical etc.). In [3], a generic solution for PC in critical systems was given. The main principles of the proposed PC solution are:

1. A subset of an existing PC solution is used. be used.
2. Traceable translations between source code, PC and target binary.

The main reason for (1) is that current PC solutions, such as ANDF [22, 23] and Java (byte-code) [17, 18] are, in general, unsuitable for critical systems in their current forms. However, it was argued in [3] that subsets of the ILs used could be identified that were amenable to critical systems. This keeps as close to standards (i.e. ANDF) as possible.

The main reason for (2) is to support the requirement in critical systems for analysis and V&V. This also helps when attempting to move software, together with its analysis and V&V evidence to a new platform. Traceability allows the binary to be related back to the source – in conventional compilers this is extremely difficult due to information loss, changes in code structure and code movement (often during optimisation).

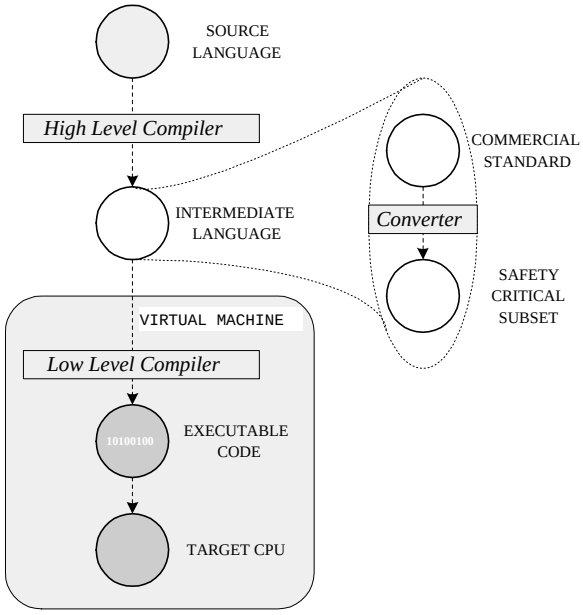

Figure 2. Overview of Portable Code Solution.

The PC solution for critical systems is illustrated in Figure 2. The high-level compiler produces either PC in an IL that conforms to a standard, which is then translated into a subset of the same language, where the subset is suitable for critical systems; or, PC in a suitable IL subset directly. This choice permits pragmatism in that suitable existing commercial compilers could be used, e.g. DDC-I Ada95-ANDF compiler. Also it permits the implementation of a specialist compiler that has the PC subset as its output (or the modification of an existing compiler). The virtual machine in the figure merely indicates that the PC could be translated to target code or interpreted – both conceptually using a low-level compiler, the former offline, the latter at run-time.

3. ANDF

ANDF was developed in a research effort led by the UK Defence Research Agency [23, 22]. ANDF offers a relatively high-level of abstraction (closer to source code than binary), maintaining much of the structure and information of the source program, in terms of loops, types etc. In terms of tools, ANDF is intended for use on the host (for both high and low level compilation). Common languages such as Java, C, Ada, Pascal etc. can all be represented in ANDF.

The general structure of ANDF is tree-like and procedural, built from constructors such as “plus” (integer addition) and “integer” (type). Each constructor belongs to a category with generic properties (e.g. operations on integers). Constructors may have explicit attributes, such as visibility to represent block-nesting in common languages. Types are represented by concrete properties, e.g. maximum and minimum values of integers. ANDF is “flat”, in that whilst procedures and functions can be represented, no nesting of these can occur within ANDF. The total number of constructors is relatively large at 272 – the tendency has been to introduce a single “new” constructor rather than structure the required semantics from the existing constructors (c.f. CISC and RISC instruction sets). As an example of ANDF, consider the following Ada expression:

```
i = a  (b + 10) ;                                                    1
```

This is represented in ANDF by:

```
assign(                                                              1
    obtain_tag(tag_1),                                               2
    mult(impossible,                                                 3
        contents(integer(var_ width(true,32)),                       4
            obtain_tag(tag_2)),                                      5
        plus(impossible,                                             6
            contents(integer(var_ width(true,32)),                   7
                obtain_tag(tag_3))                                   8
            make_int(var_width(true,32),  10))))                     9
```

In the ANDF, the variables required for the computation are “tag_1”, “tag_2” and “tag_3”, denoting Ada variables “i”, “a” and “b” respectively. The constant “10” is declared

in the final line of the ANDF. The Ada and ANDF given above is also shown, from a parse-tree perspective, in Figure 3. Clearly, there is structural equivalence between the Ada and ANDF. This is extremely useful when considering a traceable compilation approach between Ada and ANDF.

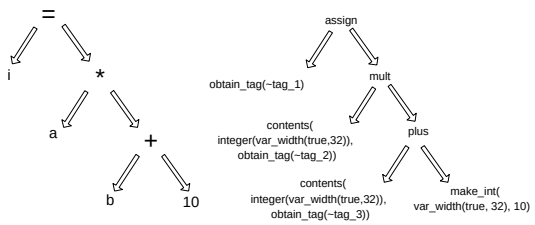

Figure 3. "`i = a * (b + 10)`": Ada Parse Tree (left) and ANDF (right)

Conventional language syntax description is normally performed in BNF (Backus-Naur Form). Instead of using BNF to describe ANDF, explicit constructors are used. These can be thought of as functions, taking a number of parameters and returning a single value. Hence, a constructor takes other pieces of ANDF as parameters and returns a piece of ANDF. For example, the ANDF construct used in the declaration of a variable is:

```
  variable:                                          1
opt_access: OPTION(ACCESS)                           2
name_intro: TAG  POINTER(alignment(x))               3
       init: EXPx                                    4
       body: EXPy                                    5
             => EXPy                                 6
```

The "variable" construct takes four parameters, namely "opt_access", "name_intro", "init" and "body". These define access to the variable, name, initialisation code and scope of the variable respectively.

4. SC-ANDF: Safety-Critical Subset of ANDF

This section describes SC-ANDF, an ANDF subset for safety-critical systems. ANDF was chosen as a basis for the subset, due to its advantages for critical systems over Java byte-code [3]. The principle of subsetting ANDF appeals to the approach for using the Ada programming language in safety-critical systems – only features amenable to safety-critical systems are used, e.g. the SPARK Ada subset [7].

Whilst SC-ANDF was developed with safety-critical systems in mind, it is suitable for most other critical systems. Thus, mission-critical or other critical systems could use the SC-ANDF subset. However, the converse is not necessarily true, e.g. safety-critical systems may not be able to use a subset developed for mission-critical systems. It is noted that other subsets may be more efficient for non-safety-critical systems.

4.1. Subset Criteria

Candidate criteria used for the definition of a PC for safety-critical systems include:

Domain Relevance – critical systems may require high integrity levels; experience indicates that only those features of programming languages (i.e. ANDF) that are required should be in the language.

Verification – certain analyses need to be performed on programs to ensure their applicability to the critical domain, so dictating the removal of difficult to analyse features.

Predictability – one aspect of potential analysis is temporal, with only those language features that are predictable in the temporal domain used. Other aspects include resource analysis, which also requires predictable behaviour.

Usability – in terms of impact on expressibility and simplicity (the desire for a small simple subset to aid verification of the subset and its use).

Performance – in terms of impact on run-time efficiency and size of executable.

4.2. SC-ANDF Definition

SC-ANDF was defined assuming that basic static control-flow (i.e. imperative) programming languages, suitable for safety-critical systems, would be supported, e.g. SPARK Ada [7] or the Ravenscar Ada95 [6, 9][1]. It is noted that the full ANDF definition caters for all programming languages (including specialised languages such as Lisp) and has features to specifically support such languages.

SC-ANDF differs from full ANDF in two main ways,. Firstly, some constructors are totally discounted due to being (a) unpredictable, or potentially leading to an unpredictable executable; (b) not required for supporting static control-flow languages such as SPARK Ada; (c) redundant functionality, i.e. some constructors replicate the functionality of others, hence are unnecessary. Secondly, some constructors are limited in their use to ensure predictability and suitability for analysis as part of verification and validation. Note that the decision to omit constructors on the grounds of replicated functionality bears in mind potential speed-ups if the operation is likely to be supported by hardware,

[1] It is noted that some elements of SPARK Ada are unsuitable for portable safety-critical systems, e.g. representation clauses through which the Ada source software can dictate the precise representation of types on underlying hardware.

i.e. by a processor instruction. For example, all logical instructions, such as "or" can be represented by a combination of "nand" – however, logical operations are typically supported directly by hardware, so are included in SC-ANDF.

Rather than modify constructors by limiting their use, the subset could have defined new constructors. However, the disadvantage with this approach is that the ANDF standard is then violated[2].

For example, the "*make_general_proc*" constructor is not included in SC-ANDF, as the same functionality can be achieved by the "*make_proc*" constructor. Also, one parameter of "*make_proc*" allows variable length parameter lists, which are not permitted in static languages, hence the appropriate "*make_proc*" parameter is disallowed.

As another example, the "*abs*" constructor, which gives the absolute value of an integer, is not included in SC-ANDF. The reason is that the operation is not often supported by hardware, and can be fabricated with a simple test and multiplication.

The resultant SC-ANDF definition fulfills the criteria given above. The ANDF specification has 272 constructors, the SC-ANDF definition permits 137 constructors, of which 8 are modified and 27 are only used for linking. Space limitations prevent detailing of the subset, although a definition can be found in [5].

5. SC-ANDF Compilation

One of the principles of the PC solution for critical systems outlined in section 2 was a traceable compilation approach. This is required due to the problems with traceability through conventional compilers. These tend to form a "black-box" within the software development process. Source code is input into the compiler, with an executable translation of that source code output from the compiler. The exact workings of the compiler, or how it has translated from input to output are not specified. Typical characteristics of the compiler include information loss during the movement from source to executable, e.g. type information is lost[3]; relative code movement occurs, i.e. interleaving of target instructions derived from different source statements; additional control flow is introduced or control flow is changed, e.g. case statements are often implemented as jump tables [1].

Given the behaviour of compilers above, it is extremely difficult to reconstruct the source from the output of a compiler. In critical software developments, this leads to the need for object code verification, where the output is traced back to the input and checked for validity – this manual process is extremely tedious, time-consuming and expensive.

The PC solution for critical systems must ensure that the translations between source code, PC and target binary are traceable. This is achieved by a table driven compilation approach where templates of PC are defined for each grammatical unit in the source language, e.g. procedure call. Likewise for PC to native translation, templates of native code are defined for each PC instruction. These can be combined to form a representation for each PC template. Whilst seemingly a good straight-forward method for compilation, it is not used in commercial compilers as the code is not at all optimised and can be seen as inefficient.

This template driven compilation approach leads to far simpler code generation (either from source to PC, or PC to native). This is an advantage when arguments regarding the "quality" of the compiler are being made – i.e. whether it is of sufficient standard for use on safety-critical projects. Also, it is apparent that if a compiler is created specifically for PC, it can utilise and preserve annotations, such as those used in SPARK Ada [7] for static analysis, and those used for worst-case execution time analysis [11]. This form of information preservation can only help in the later stages of software development.

It is noted that a traceable template driven approach to compilation reduces the opportunities for optimisation of the code. However, for critical systems, the benefits gained in terms of traceability and ease of verification outweigh the benefits of faster code.

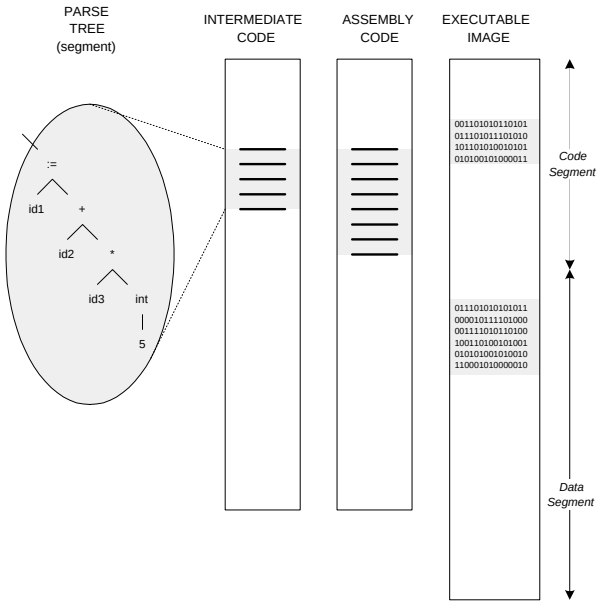

Figure 4. Traceable Compilation.

[2] Also, existing low-level compilers (from ANDF to native) would not work on such a subset.

[3] Note that whilst "debug" information can reduce the amount of information loss during compilation, this is insufficient for traceable compilation. Indeed, debug information merely refers to line numbers in the source file, rather than allowing the reconstruction of the source file from the executable.

Figure 4 illustrates the use of templates during compilation (and beyond). Essentially, for any segment of the parse tree, all subsequent representations (intermediate code, assembly code, executable image) associated with that segment are contiguous. As a result of traceable compilation much information is available, including mappings between the different representations of the code during the compilation process (i.e. source, intermediate, target); annotations from the code, particularly those which enable data-flow [7] and WCET analyses [11]; pertinent analysis information from the source, or derived during the compilation process; and behavioural consistency information, to enable the code to behave the same on all platforms. This information supports traceability between the compilation phases.

5.1. High-Level Compilation

For high-level compilation, templates need defining between source language and SC-ANDF. The process for defining the templates involves consideration of the syntax and semantics of each syntactic rule in the language, then selecting the most appropriate (and obvious) mapping to SC-ANDF constructs. The emphasis is on preserving the structure and meaning implied by the source language, whilst attempting to create a concise and efficient SC-ANDF template.

When considering SPARK Ada95 as the source language, it is clear that the mapping to SC-ANDF is relatively straightforward. In most cases, an Ada95 statement maps to a small set of SC-ANDF constructs.

It is anticipated that this approach can be applied across the complete source language – especially if the source language is static, like SPARK Ada95 [7] or Ravenscar Ada95 tasking subset [2]. It is beyond the scope of this report to state the complete Ada95 to SC-ANDF template mapping. This requires consideration of the complete syntactic definition of Ada95 (or an appropriate safety-critical subset). However, the following sections provide a number of examples illustrating how the templates can be developed.

Example: Ada95 "For" Statement

The "For" statement in Ada95 has to be mapped onto the only loop construct in SC-ANDF, namely the "`repeat`". Consider the following Ada95 code segment:

```
for Low in IndexType range 1 .. MaxTableSize-1 loop
   Key := FindSmallest(Table, Low, MaxTableSize);
   if Key /= Low then
      SwapElements(Table, Low, Key);
   end if;
end loop;
```

The Ada "`for`" construct must map onto the only loop construct in SC-ANDF, namely the "`repeat`" construct. At first glance, the SC-ANDF "`repeat`" statement is insufficient, as it does not contain any conditional expression for testing if the loop should be exited. However, by refining the Ada "for" syntax a closer correspondence emerges (in BNF):

```
loop_statement ::=                                                  1
  loop if def_id in [reverse] discrete_type_name [range exp..exp]   2
       then seq_of_statements                                       3
            def_id := def_id + 1 [- 1];                             4
       else exit loop; end if; end loop;                            5
```

This places the conditional statement within the loop. In SC-ANDF, the conditional must be placed as the first expression within the loop. Thus, the loop condition is checked immediately on entering the loop, executing the loop body if the condition still holds. Thus, the template is given by:

```
loop_statement(def_id, discrete_type_name,                          1
               seq_of_statements, def_id, exp1, exp2)               2
=> repeat (label_name_1, make_top,                                  3
     conditional(label_name_2,                                      4
       sequence(body_cond(def_id, discrete_type_name, exp1, exp2),  5
         body_loop(seq_of_statements, def_id, label_name_1)),       6
         return))                                                   7
```

Note that "`label_name_1`" represents the label used to reiterate around the loop. This would be called at the end of "`body_loop`". Also, "`body_cond`" is a further template representing the conditional statement body at the start of the loop. The label used if the loop conditional fails at the start of the loop is "`label_name_2`". Jumping to this label effectively ends the loop as the "`return`" construct is executed. The main loop body is represented by "`body_loop`", as defined by "`seq_of_statements`". This concludes with the automatic increment/decrement of the loop identifier "`def_id`" together with a "`goto(label_name_1)`" construct to give loop repetition.

5.2. Low-Level Compilation

Traceable low-level compilation conserves the traceability of high-level compilation. In this section, a low-level compilation approach is described which maintains the traceability of the high-level compiler.

Code Optimisation for PC

Code optimisation is performed in conventional compilers to ensure reasonable final code. In general, little care is taken when generating the pre-optimised code – much reliance is placed upon subsequent optimisation to improve the code to an acceptable level. It has often been argued [16], that better final code can be achieved by better

initial code generation. This places less emphasis on optimisation, mainly because some of the basic optimisations are redundant as they are effectively incorporated into the initial code generation. As an example, constant substitution [1] can be carried out during code generation.

Many code optimisations use the data-flow of the code to improve code. For example, the removal of dead code. However, to perform data-flow based optimisation, the compiler needs knowledge of the data-flow. Usually, this is performed by examining the native code and attempting to understand the data-flow through registers and memory.

A key advantage of the approach to compilation outlined in this section is the preservation of the data-flow. This is due to the data-flow described at the source level by annotations (e.g. SPARK Ada95 annotations [7] are maintained by the source language to SC-ANDF mappings. This allows some code optimisations dependent upon data-flow to be incorporated directly in the SC-ANDF to native code generation stage, without a separate optimisation phase. For example, dead code can be determined at the source level with SPARK annotations [11].

SC-ANDF to Native Code Mapping
Whilst SC-ANDF to native mapping is relatively straightforward, traceability must be maintained whilst achieving an acceptable level of performance efficiency. Therefore, in a similar manner to high-level compilation, a "table-driven" approach to translation is used. Thus, a SC-ANDF construct is replaced by a template of native code in the same way that an Ada statement was replaced by a SC-ANDF template during high-level compilation. The process for defining the templates involves consideration of each SC-ANDF template, then selecting the most appropriate mapping to native code. A number of issues need to be considered when defining the mapping:

Simplicity – It is probable that the templates need to be expressed in a subset of the available processor instruction set. This appeals to the example set by SPARK Ada and the subsequently less onerous task of certification.

Efficiency – Whilst only a small number of SC-ANDF templates are suitable for a given source language statement, far more flexibility is available for SC-ANDF to native code templates. As long as traceability is maintained, then the most efficient template should be used – within the bounds of the processor instruction subset.

Traceability – It is important that the traceability from source to SC-ANDF is maintained from SC-ANDF to native code, to assist in timing analysis, and object code verification, together with making the compilation system more trustworthy. Thus, *no* optimisation occurs on the SC-ANDF or native code before, during or after translation – for each SC-ANDF construct, a simple native code template is used.

Clearly, the simplicity and efficiency issues are part of a trade-off. Using a richer subset of the processor instruction set will generate more efficient code. However, using a minimal subset keeps to the SC-ANDF model better, and is easier to validate and verify as part of a "trusted" compilation strategy for safety-critical systems.

5.3. ANDF to SC-ANDF Translation

In section 2 it was noted that in the absence of a bespoke source to SC-ANDF compiler, a translator would be required to convert the ANDF into SC-ANDF. This raises the issues of how to perform translations within the ANDF model; definition of ANDF to SC-ANDF translations; and maintaining traceability. The first problem is solved by the ANDF definition [22], which permits ANDF to ANDF transformations, thus allowing ANDF to SC-ANDF translation. The other issues are addressed below.

Example: make_int $\Longrightarrow$ make_value
The "*make_int*" construct allows an integer value to be declared, initialised and its value returned in a single statement. This facility does not exist for any other data type. The more general "*make_value*" construct can also be used to define space for integers, giving a consistent approach for all data types:

```
make_int(v, value) =>                                              1
    variable(_, make_value(integer(v)),                            2
        sequence(assign(obtain_tag(tagx),value),                   3
            contents(integer(v),obtain_tag(tagx)))                 4
```

Note that the first parameter of "*variable*" is ignored and that "`tagx`" represents the next internal name used in the expression tree (usually generated by the compiler).

6. Conclusions

This paper has reported work supported by British Aerospace Military Aircraft and Aerostructures addressing the issue of interchangeability in safety-critical systems, using portable code. Expanding upon the approach given in [3], this paper has tailored the approach towards Integrated Modular Avionic (IMA) systems, in order to achieve software interchangeability in the aerospace domain. The contributions of the paper have been twofold. Firstly, a safety-critical subset of ANDF, called SC-ANDF, has been described. Secondly, an appropriate compilation approach has been defined for portable code, that ensures traceable

compilation. This aids the analyses that need to be performed on the source and object code to ensure that it is fit for use in a critical system. This is particularly useful for IMA (and other safety-critical systems) where software may need to be moved to a new hardware platform – ease of analysis is key to reducing the system development and lifecycle costs.

The definition of SC-ANDF is to be treated as preliminary. Only after extensive testing and further analysis will the definition of SC-ANDF be complete. It is not anticipated that constructors will be found during this work that should not be in the subset, rather the usage of the SC-ANDF constructors needs to be refined.

The compilation approach for portable code is based around template-based substitution of source language statements into SC-ANDF as part of high-level compilation; substitution of SC-ANDF constructs into native code as part of low-level compilation. This approach forms the initial step toward a traceable trusted compilation approach – which is lacking in current safety-critical developments.

Further work is required to complete the approach. This includes the definition of a complete source language (i.e. Ada 95) to SC-ANDF mapping; definition of a complete SC-ANDF to native code mapping; identification of optimisations that can be incorporated into low-level compilation without violating the traceability of compilation; and demonstration of key concepts.

The cost benefits of such a compilation approach to critical system developments should not be ignored. Usually, significant time and expense is absorbed in ensuring that the code produced by the compiler is adequate. This would be reduced by the proposed compilation approach.

References

[1] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1986.

[2] Annex H (Safety and Security) Rapporteur Group. *Guidance for the use of the Ada Programming Language in High Integrity Systems*, version 3.7 edition, 1998.

[3] N. Audsley, I. Bate, and A. Grigg. Portable Code: Reducing the Cost of Obsolesence in Embedded Systems. *IEE Computing and Control*, 10(3):98–104, June 1999.

[4] N. Audsley and M. Burke. Distributed Fault-Tolerant Avionic Systems - A Real-Time Perspective. In *Proceedings IEEE Aerospace Conference*, Aspen, USA, 1998.

[5] N. C. Audsley, I. J. Bate, and A. Grigg. Portable Code for Avionic Systems: Phase 3. Technical Report COE/SPO-S-0196010/Phase3, BAe. Centre of Excellence, Dept. of Computer Science, University of York, York, UK., 19th March 1999.

[6] T. Baker and T. Vardanega. Session summary: Tasking profiles. In A. Wellings, editor, *Proceedings of the 8th International Real-Time Ada Workshop*, pages 5–7. ACM Ada Letters, 1997.

[7] J. Barnes. *High Integrity Ada: The SPARK Approach*. Addison-Wesley, 1997.

[8] J. Borky, R. Lachenmaier, J. Messing, and A. Fink. Architectures for Next Generation Military Avionics Systems. In *Proceedings IEEE Aerospace Conference*, Aspen, USA, 1998.

[9] A. Burns, B. Dobbing, and G. Romanski. The Ravenscar tasking profile for high integrity real-time programs. In *Reliable Software Technologies, Proceedings of the Ada Europe Conference, Uppsala*, pages 263 – 275. Springer Verlag, 1998.

[10] A. Burns and J. A. McDermid. Real-time safety-critical systems: Analysis and synthesis. *Software Engineering Journal*, pages 267–281, November 1994.

[11] R. Chapman. Static Timing Analysis and Program Proof. Technical Report YCST-95-05, Dept. of Computer Science, University of York, York, UK., March 1995.

[12] G. Clark and A. Powell. Experiences with Sharing a Common Measurement Philosophy. In *Proceedings International Conference on Systems Engineering (INCOSE'99)*, Brighton, UK, 1999.

[13] R. A. Edwards. ASAAC Phase I Harmonized Concept Summary. In *Proceedings ERA Avionics Conference and Exhibition*, London, UK, 1994.

[14] European Organisation for Civil Aviation Electronics. *DO-178B: Software Considerations in Airborne Systems and Equipment Certification*, December 1992.

[15] D. Field and A. Grigg. The Impact of Interchangeability Requirements on Operating Systems for Modular Avionics. In *Proceedings ERA Avionics Conference and Exhibition*, 1994.

[16] C. H. Forsyth. More Adaptable Code Generation. DPhil. Thesis YCST/91/01, 1991.

[17] J. Gosling, B. Joy, and G. Steel. *The Java Language Specification*. Addison-Wesley, 1996.

[18] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1997.

[19] Ministry of Defence. *Defence Standard 00-55: Requirements for Safety-Related Software in Defence Equipment (Part 1: Requirements, Part 2: Guidance)*, August 1997.

[20] C. Multedo, D. Jibb, and G. Angel. ASAAC Phase II Programme Progress on the Definition of Standards for the Core Processing Architecture of Future Military Aircraft. In *Proceedings ERA Avionics Conference and Exhibition*, London, UK, 1998.

[21] R. Torbet. Future Offensive Air System - Avionic Requirements. In *Proceedings ERA Avionics Conference and Exhibition*, London, UK, 1997.

[22] X/Open Company Ltd. *X/Open Guide: Architecture Neutral Distribution Format*. X/Open Company Ltd., UK, 1996.

[23] X/Open Company Ltd. *X/Open Preliminary Specification: Architecture Neutral Distribution Format*. X/Open Company Ltd., UK, 1996.