A TOOL ARCHITECTURE FOR APPLICATION OF PORTABLE CODE TECHNOLOGIES TO FUTURE AVIONIC SYSTEMS

N.C. Audsley, I.J. Bate and A. Grigg
Dept. of Computer Science, University of York, UK

M.A. Fletcher and A.S. Wake
BAE SYSTEMS, Brough, UK

AUTHOR BIOGRAPHICAL NOTES

Alan Grigg received a BSc in Mathematics at Thames Polytechnic (now University of Greenwich) in 1985. He subsequently joined BAE SYSTEMS and worked for several years on the specification/design of a software architecture for military IMA systems. In 1996, he joined the BAE SYSTEMS Dependable Computing Systems Centre (DCSC) at the University of York to pursue research into real-time systems design and timing analysis. His DPhil Thesis concerns an incremental timing analysis approach for IMA systems.

Neil Audsley received a BSc in Computer Science and DPhil from the University of York in 1988 and 1993 respectively. His doctoral thesis considered the scheduling and timing analysis of safety-critical systems. He has published numerous technical papers. He is now a Senior Lecturer in the Department of Computer Science at the University of York, researching the design and implementation of real-time embedded systems.

Iain Bate received a MEng in Electronic Systems Engineering and DPhil in Computer Science from the University of York in 1990 and 1999 respectively. His doctoral thesis considered the scheduling and timing analysis of safety-critical systems. He has published numerous technical papers. He has worked with the Rolls-Royce University Technology Centre (1994-1998) and the BAE SYSTEMS Dependable Computing Systems Centre (1999-) at the University of York, specialising in timing analysis, kernels and related issues for safety-critical systems.

Martyn Fletcher received a BSc degree and diploma in Applied Physics from the University of Hull and a MSc degree in Integrated Circuit System Design from the University of Manchester Institute of Science and Technology. For the past nine years, he has worked in R&D at BAE SYSTEMS. Initially, his work involved investigating techniques for distribution and fault tolerance in computer systems. He has now worked for some time on the IMA software architecture, both on BAE SYSTEMS internal R&D programme and on national and international programmes.

Allan Wake received an honours degree in Engineering from the University of Humberside. He joined BAE SYSTEMS in 1986, working initially on aircraft training aids development. He currently leads the BAE SYSTEMS IMA software team which is heavily involved in collaborative, multi-national research programmes including ASAAC.

ABSTRACT

The application of portable code technologies is being considered as a means of reducing the impact of hardware obsolescence in future avionic systems. This paper describes a portable code development toolset architecture that has been defined (and demonstrated in concept) to meet the high dependability needs of avionics applications with real-time and safety-critical components. The tool architecture specifies a number of static analysis tools (for timing and other analyses) as well as portable code development and translation tools. The paper also describes a set of guidelines on tool procurement that have been produced based on a consideration of COTS and bespoke development options and experiences gained from practical demonstration of our portable code for avionic systems concept.

1. INTRODUCTION

Embedded computer systems often have a long in-service time, during which it is inevitable that certain parts of the original computing system will become obsolete. At some point, the hardware used may not even be manufactured anymore, with limited spares availability. This is known as the computer systems *obsolescence* problem. Many industries are affected by obsolescence but civil and military aerospace are prime examples due to the long lifetime of aircraft.

Conventional methods for mitigating against the problem of obsolescence include buying stocks of components and storing them until they are needed, or facing the possibility of having to modify the software for a new platform (often referred to as porting) at a later date. Obviously, both of these options incur a cost penalty. The former ensures that sufficient hardware components are available for the lifetime of the system but it does not enable new technologies to be used. The latter option often results in a costly re-design. For example, if the software is ported to a new platform after 20-30 years, then it is unlikely that any of the original designers remain available and if they do, then the platforms and tools on which they built the system are unlikely to be available.

Component obsolescence is being addressed for civil and military avionic systems as part of their respective IMA programmes, ARINC[7] and ASAAC[8]. IMA is aiming to define a standard avionics computing system architecture through the use of standard, modular hardware and software components. IMA systems aim to provide highly flexible, reliable and integrated solutions, exploiting new advances in technologies as they become available and not merely as part of a large "mid-life update" effort. IMA system requirements promote the need for software reuse, both within and between aircraft projects, and interchangeability between standard hardware modules (with a common specification but different implementation, e.g. from different suppliers).

The work described in this paper aims to complement ongoing IMA interface standardisation work to provide an overall solution to the software obsolescence problem. A portable code infrastructure is described that allows avionics software to execute on a wide range of platforms.

2. PORTABLE CODE TECHNOLOGY - AN OVERVIEW

Conventionally, source code is translated directly to target-specific executable code by a compiler[2] (Figure 1a). This ensures the obsolescence of the executable software as soon as the hardware changes. The compiler is executed on a host / development system with, the executable code loaded onto the target for execution. If the source code is compiled for a different target, a different compiler is required – each compiler is limited to a single host-target combination (Figure 2a).

Portable Code (PC) employs a target-independent *Intermediate Language* to support a two-stage compilation process as follows (see Figure 1b):

1. *High-Level Compilation* – the translation of source code, written in a high-level language, into the PC (intermediate language) form; this stage is performed off-line.
2. *Low-Level Compilation* – the translation of PC into target-specific executable code; this stage can be performed whenever the target hardware becomes known, either off-line or on-line.

Under the PC compilation approach, the movement of source code to a new platform merely involves low-level recompilation of the PC to the new target executable format. Hence the number of tools required for porting software is much reduced (see Figure 2b).

An intermediate language is a more abstract representation than an executable form but less abstract than a high level programming language. Whilst it is possible to program directly in the intermediate language, it is not recommended, since this is less intuitive, less structured and harder to comprehend than a high level source language. Also, software engineering experience suggests that programming at a lower level is more difficult, more error prone and hence more costly than programming at a higher level. Therefore, a high-level compiler is used to produce code in the intermediate language.

Two main existing portable code solutions have been proposed:

1. *ANDF (Architecture-Neutral Distribution Format)*[3] – ANDF offers a high-level of abstraction, providing an efficient way of representing programs written in any one of many high-level programming languages. ANDF maintains much of the structure and information of the source program, in terms of loops, types etc.
2. *Java Bytecode* – This approach is commonly used in internet applications. The problem of executing programs across the internet is the ultimate portability problem, since the program developer has no idea of the final machine configuration (processor, operating system, etc.). Java consists of the Java programming language[4]; an intermediate language called Java Bytecode[5]; and the Java Virtual Machine (JVM)[5]. Java programs are compiled into the portable Java Bytecode, which is interpreted by the JVM. The JVM is intended to execute over a conventional operating system. Java bytecode offers a low-level of abstraction – it is intended to be as close to a generic target assembly language as possible (for efficiency reasons).

3. PORTABLE CODE SOLUTION FOR AVIONIC SYSTEMS

The PC solution is built around SC-ANDF, a subset of full ANDF. The solution is depicted in Figure 3 and described below.

3.1 Safety-Critical ANDF (SC-ANDF)

Instructions in ANDF were omitted from the SC-ANDF subset for the following reasons:

1. *Redundancy* – Many ANDF instructions can be synthesised with other instructions.
2. *Inapplicability* – Instructions, such as those that appear to provide particular support for specific programming languages (such as Lisp), that are not likely to be used in the development of avionic systems, are classed as inapplicable (assuming that avionic systems will be programmed using imperative languages).
3. *Unpredictability* – ANDF instructions whose run-time operation may be difficult to predict without a high degree of pessimism.

On this basis, SC-ANDF has been defined to meet the likely needs of future avionic systems. SC-ANDF contains only about half of the original ANDF instruction set. This approach appeals to that for using the Ada programming language in safety-critical systems – only those features amenable to safety-critical systems are used, e.g. the SPARK Ada subset[6].

3.2 Compilation

This approach allows existing high-level compilers to be used, giving two possible routes between source code and SC-ANDF:

- Source to ANDF via a COTS ANDF compiler, then to SC-ANDF via a bespoke *Converter* tool;
- Source to SC-ANDF via a bespoke high-level compiler.

The “black box” nature of COTS compilers eliminates traceability between source code and intermediate code. If this is lost at the ANDF level then it will be difficult or impossible to regain at the SC-ANDF level. This leads to the need to perform manual validation of the SC-ANDF on a per-compilation basis, which would be time consuming and costly. The best approach is the development of a bespoke high level compiler that maintains traceability between source code and SC-ANDF.

4. PORTABLE CODE TOOL ARCHITECTURE

The overall tool architecture for the portable code solution proposed for IMA is shown in Figure 4. The diagram illustrates the aforementioned high and low level compilation tools but also those for analysis and platform-specific validation. The tools for each of the stages of translation and analysis are illustrated in further detail in Figures 5-8 and are discussed further below.

4.1 High-Level Compilation Phase

Three tools are required; a high level compiler, a SC-ANDF linker and a debugger (Figure 5). The high level compiler was described above. The SC-ANDF linker must resolve all external references in

a given SC-ANDF file (such as OS calls, *eg.* APOS[9] calls in the ASAAC IMA solution or APEX[10] calls in ARINC). All such references are resolved via target-*independent* libraries - all linking with target-dependent libraries are postponed to the low level compilation stage for improved portability. The debugger provides symbolic debugging facilities with debugging information obtained from the high level compiler via a separate file rather than embedded in the SC-ANDF.

4.2 Static Analysis Tools

The analysis toolset (Figure 6) supports a range of static analyses at the source and SC-ANDF level with comparison between the two sets of results to demonstrate traceability. Certification evidence is generated on the basis of this off-line, target-independent analysis.

4.3 Platform Compliance Tools

A range of IMA tools are required to assess the suitability of a given target hardware platform for execution of the software (Figure 7). Portable code requires two specific tools at this stage to validate the worst-case execution time and memory usage of the code in each SC-ANDF file. The final part of the certification 'jigsaw' is completed by demonstrating compliance for a particular platform.

4.4 Low-Level Compilation Tools

In general, a single tool is required for low level compilation (Figure 8). However, in order to compile the IMA OS itself, an additional object code linker may be required to allow MOS calls to be resolved in favour of hardware-dependent library references. The integrity of the low level compiler is a major concern, especially where this is to be applied on-line.

Currently, a portable code testbed is being developed to demonstrate key concepts of the proposed portable code solution, including the above tool architecture.

5. COTS TOOLSET

This section examines the impact of COTS tools upon the IMA PC toolset. This assumes both high-level and low-level compilers are COTS. The main differences are:

- *ANDF to SC-ANDF Conversion Required* – A COTS high-level compiler outputs ANDF rather than SC-ANDF. In accordance with previous work[15] an additional (bespoke) tool is required to convert the ANDF to SC-ANDF. This is illustrated in Figure 3.
- *Analytical Evidence Required at Object Level* – In the ideal toolset, once the high-level compilation phase is complete, all analytical certification evidence has been produced. With COTS tools this is not achievable, as traceability is lost from the source by use of a COTS compiler. With a COTS low-level compiler, the *Low-Level Compilation Phase* must perform object code analysis to check that the executable code is a good representation of the SC-ANDF.
- *Removal of Comparator Tools* – A COTS high-level compiler precludes the ability to compare analysis performed at the source and SC-ANDF levels, due to the lack of traceability between source and the output of the high-level compiler, i.e. ANDF.
- *Removal of Platform Compliance Phase* – For a COTS toolset, the *Platform Compliance Phase* as described in section 4 is no longer applicable. Essentially, a COTS *Low-Level Compiler* can only produce the target executable code offline, with no traceability or template control between SC-ANDF and object code. This means checking of time and memory usage by conventional means, eg. by analysing the object code or by test.

6. TOOL PROCUREMENT

Issues related to the procurement of tools for PC-technology-based systems are now considered. Many of these issues are common to the procurement of tools to support development of software for systems that could be described as large-scale, high integrity or IMA. This section assumes the use of a bespoke toolset (section 4) rather than a COTS toolset (section 5).

6.1 Tools for High Integrity Systems

The integrity of the airborne system is a by-product of the integrity of the process by which it was developed. That process is typically implemented by a set of manual and automated procedures, implemented by human engineers and computer-based tools respectively. The integrity of the tools involved is therefore a significant concern:

- *Defence Standard 00-55*[12]– states (in section 36.2) that software development tools should be validated to a level of integrity designated by a specific hazard analysis and safety risk assessment of the system to be developed by the tools in question.
- *DO-178B*[13] – states (in paragraph 12.2.1) that any tool used in the development of airborne software should be qualified to the same level of integrity as the software it is used to develop, unless it can be explicitly justified otherwise.

Common practice indicates that the tools used are indeed justified to be of lower integrity than the airborne software. Indeed, it is commonly argued that software development tools do not themselves require such rigorous development as the actual software that forms the final airborne system, even for safety critical components. PC suggests the need for more rigorous development of the toolset as the level of trust in some of these tools is raised significantly compared to their status in the traditional development process. The use of a PC within the software process has additional benefits:

- the PC offers a potentially seamless interface between compiler front-ends and back-ends from different suppliers, making the integration of such tools more straight forward;
- the division of the software translation process into two distinct stages with separate tools should help with tool qualification and maintenance;
- the use SC-ANDF assists tool validation by reducing the scope (and presumably the size) of such tools since they operate on a reduced subset of the ANDF.

As stated above, however, the use of PC implies an increased level of trust in certain parts of the software development toolset. Correspondingly, this may lead to a need for higher integrity tools.

6.2 High-Level Compiler

The issue of compiler integrity has been the subject of much research over the past thirty or so years and is still a problem today. In general, there are two approaches to the problem[15]:

1. *Verified Compilation* – general compilation scheme is formally proven to provide correct translation from a (formally) specified source language to a (formally) specified target language. Thus, a compiler is verified to *always* produce a valid translation of the input, given that the input is valid source language. Such a compiler is unlikely ever to be a reality for an industrially sized language[14].
2. *Trusted Compilation* – shown to a high degree of confidence to provide either a correct translation between source and target languages or no translation at all, but not an incorrect translation.

The trusted compilation approach is more practical. It generates, on a per-compilation basis, evidence that the translation is correct by utilising simplistic template based code-generation and translation. For many years, this evidence has been generated in industry by target code verification -- this mitigates some of the risk of using low integrity COTS compilers.

The IMA PC solution proposed in this project offers further advantages to high-level compiler integrity. In particular, the use of templates throughout the compilation process promotes simplicity of the compiler implementation and traceability of its results.

6.3 Low-Level Compiler

Verified low-level compiler development is feasible[1] as only simplistic translation occurs. A verified low-level compiler is essential for the IMA PC toolset, especially if pre-run-time installation or run-time interpretation is to occur – the tool is now an active part of the system. The development of the low level compiler should be rigorous, if not formal. Indeed, under DO-178B or 00-55, such a low-level compiler must be developed to the highest integrity level. Alternatively, a trusted low-level compiler could be used, potentially restricting the use of the PC solution to the off-line version only.

6.4 Other Tools

The other tools in the IMA PC toolset do not cause many problems from an integrity perspective. The other tools are only used offline, and are mainly simple, e.g. the linkers. The more complex offline tools, e.g. analysis, do not have specific integrity requirements due to PC (unlike the compilers). Therefore, the same approach used for current tools is likely to be sufficient, i.e. it is likely that the tools can be of lower integrity than the resultant airborne software.

7. CONCLUSIONS

Portable code provides a technology by which software can be developed and compiled once, but executed on different platforms without change (or source code recompilation). Hence, portable code should reduce re-development costs associated with system modification, including those due to hardware obsolescence.

In this paper, an approach has been outlined for the use of portable code in safety-critical embedded systems. The approach is based upon the ANDF portable code solution. Since safety-critical systems are required to be predictable and analysable, the proposed portable code solution maintains such characteristics from source code to portable code and then from portable code to a variety of target computing platforms. The solution is equally applicable to military and civil avionics systems, both safety-critical and non-safety critical. Some guidelines are also provided for application to *current* legacy and obsolescence problems. A potential portable code toolset architecture for COTS and bespoke tools has been given, together with procurement guidance.

ACKNOWLEDGEMENT

The work described in this paper was funded by BAE SYSTEMS via *Center of Excellence* contracts to the Department of Computer Science, University of York, York, UK.

REFERENCES

[1] Gaul, T. *et al*. "Construction of Verified Software Systems with Program Checking: An Application to Compiler Back-Ends", in Workshop on Run-Time Result Verification, London, 1994.

[2] Louden. K. C., "Compiler Construction: Principles and Practice," *PWS Publishing*, 1997.

[3] "Architecture Neutral Distribution Format (XANDF)," *XOpen Co. Ltd.*, 1996.

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

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

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

[7] "ARINC Specification 651 : Design Guidance for Integrated Modular Avionics," *Airlines Electronic Engineering Committee*, 1991.

[8] Multedo, G. *et al*, "ASAAC Phase II Programme Progress on the Definition of Standards for the Core Processing Architecture of Future Military Aircraft," *Proc. of ERA Avionics Conference*, 1998.

[9] Field, D., *et al* "Software Techniques for IMA - System Management, System Security and Blueprints," *Proc. of ERA Avionics Conference and Exhibition*, 1997.

[10] "ARINC Specification 653 : Avionics Application Software Standard Interface," *Airlines Electronic Engineering Committee*, 1997.

[11] Grigg, A., Audsley, NC., Fletcher, MA., Wake, AS., "A Method for Design and Analysis of Next Generation Aircraft Computer Systems," *Proc. of INCOSE Conf. on Systems Engineering*, 1999.

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

[13] "Software Considerations in Airborne Systems and Equipment Certification," *RTCA-EUROCAE* DO-178B/ED-12B, 1992.

[14] Bowen, J., (ed) "Towards Verified Systems", vol. 2 of "Real-Time Safety-Critical Systems", Elsevier, 1994.

[15] Stringer-Calvert, D.J., "Mechanical Verification of Compiler Correctness", DPhil Thesis, YCST/98/05, Dept. of Computer Science, University of York, 1998.

[16] Audsley, N.C. *et al*, "Portable Code for Avionic Systems", ERA Avionics, pp.6.2.1-6.2.10, 1999.

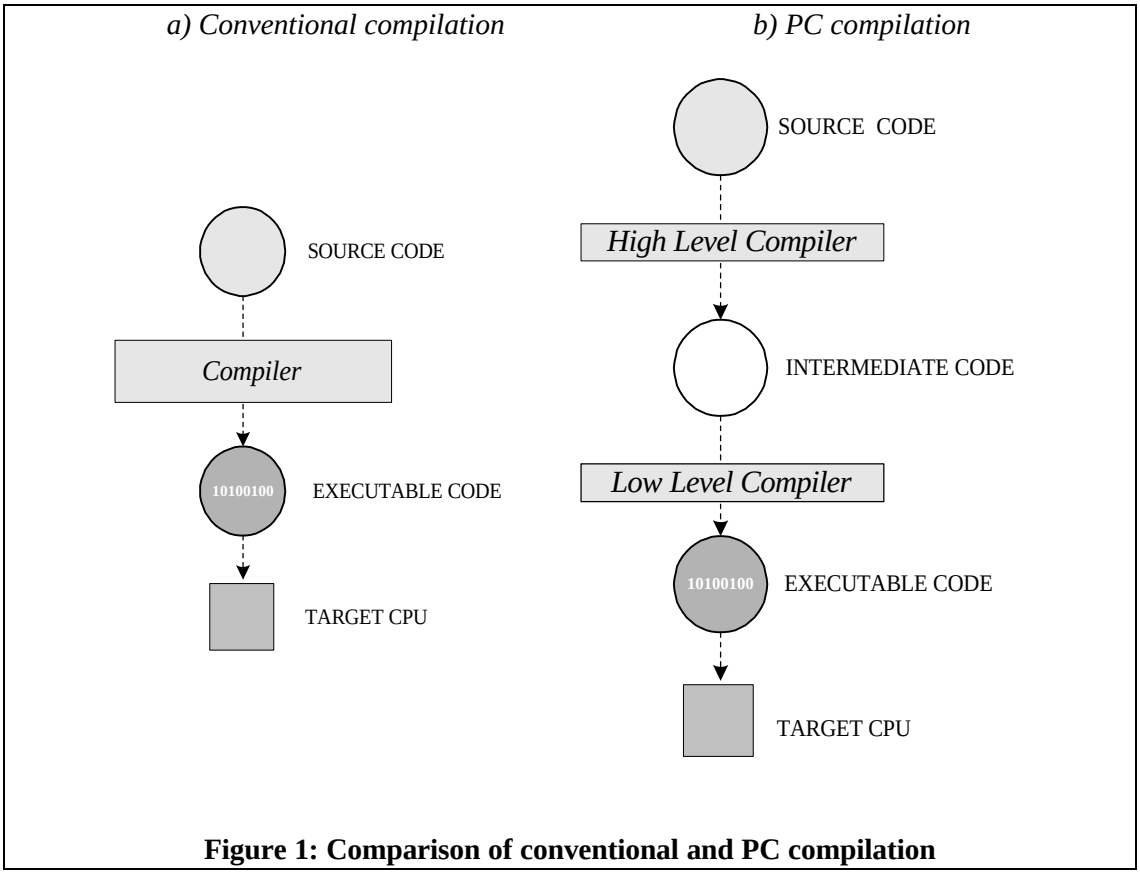

Figure 1: Comparison of conventional and PC compilation

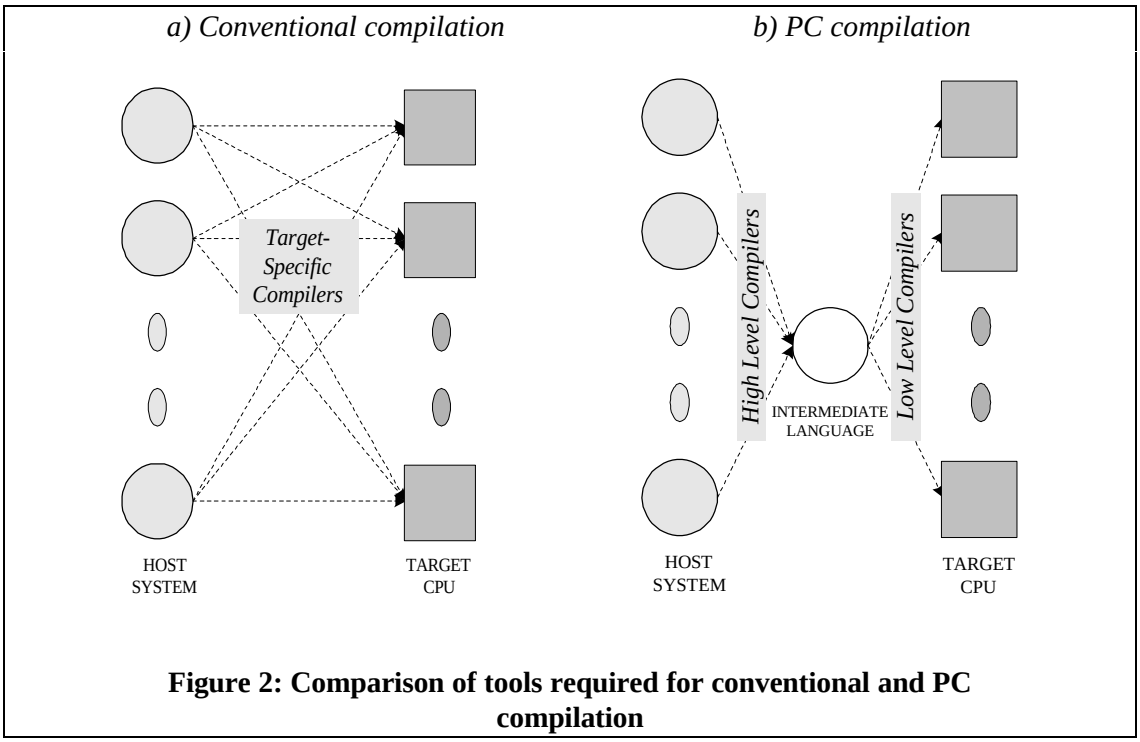

Figure 2: Comparison of tools required for conventional and PC compilation

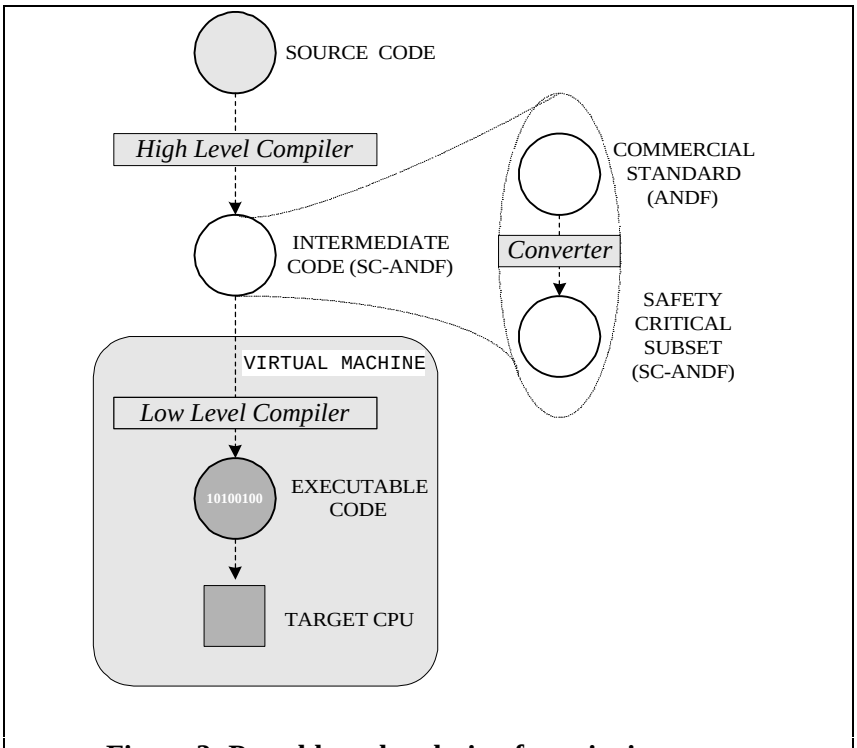

Figure 3: Portable code solution for avionic systems

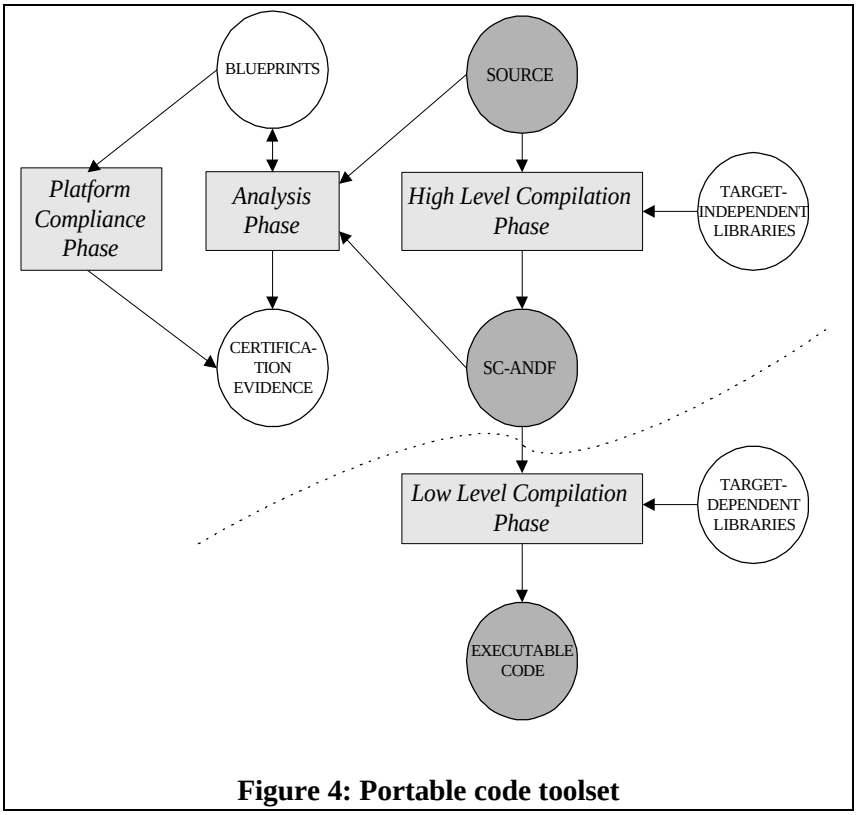

Figure 4: Portable code toolset

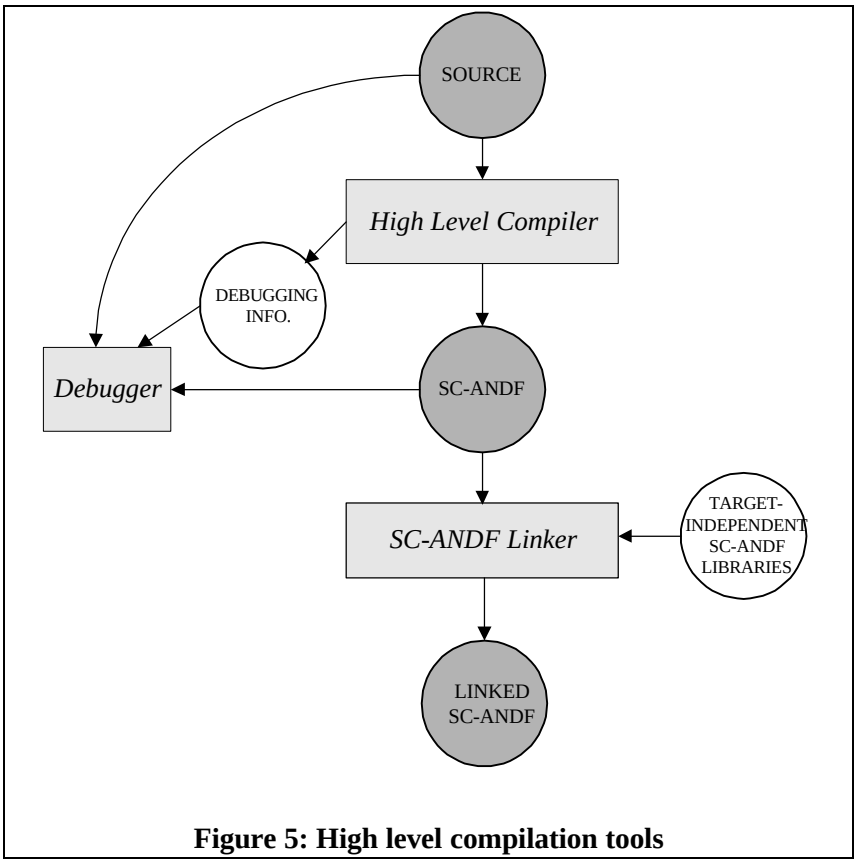

Figure 5: High level compilation tools

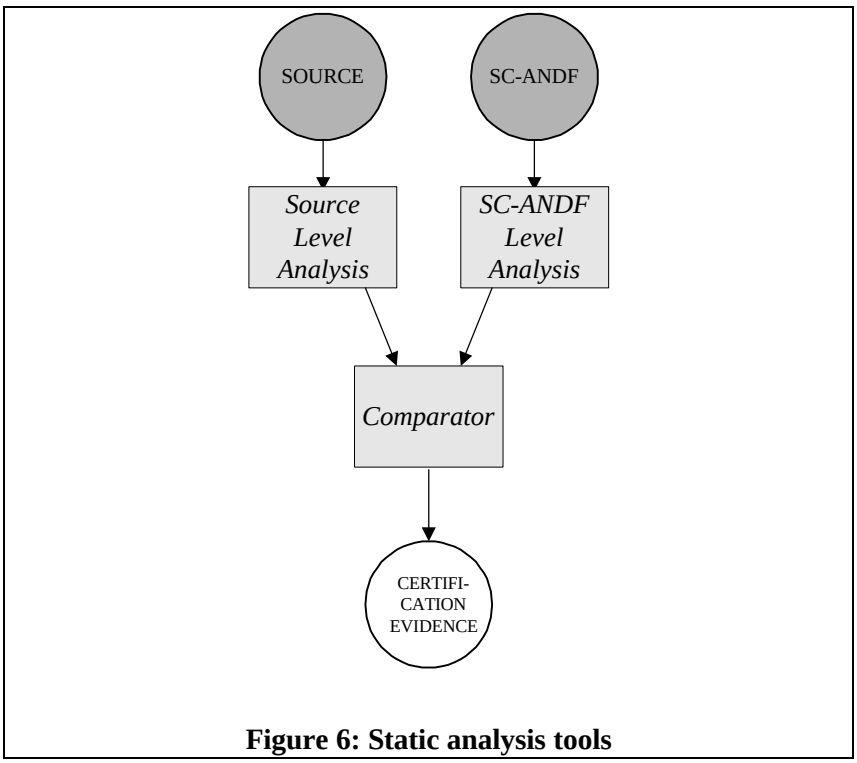

Figure 6: Static analysis tools

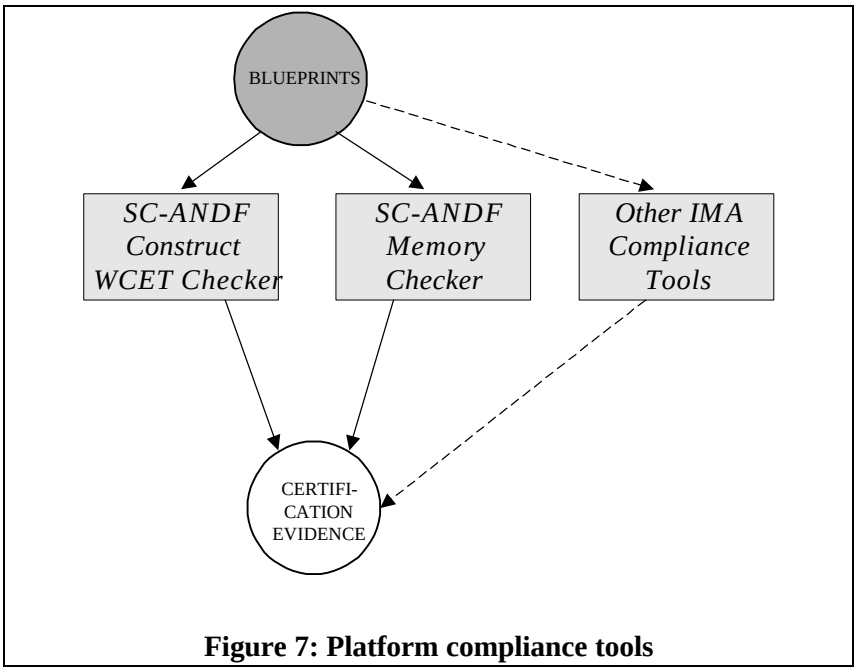

Figure 7: Platform compliance tools

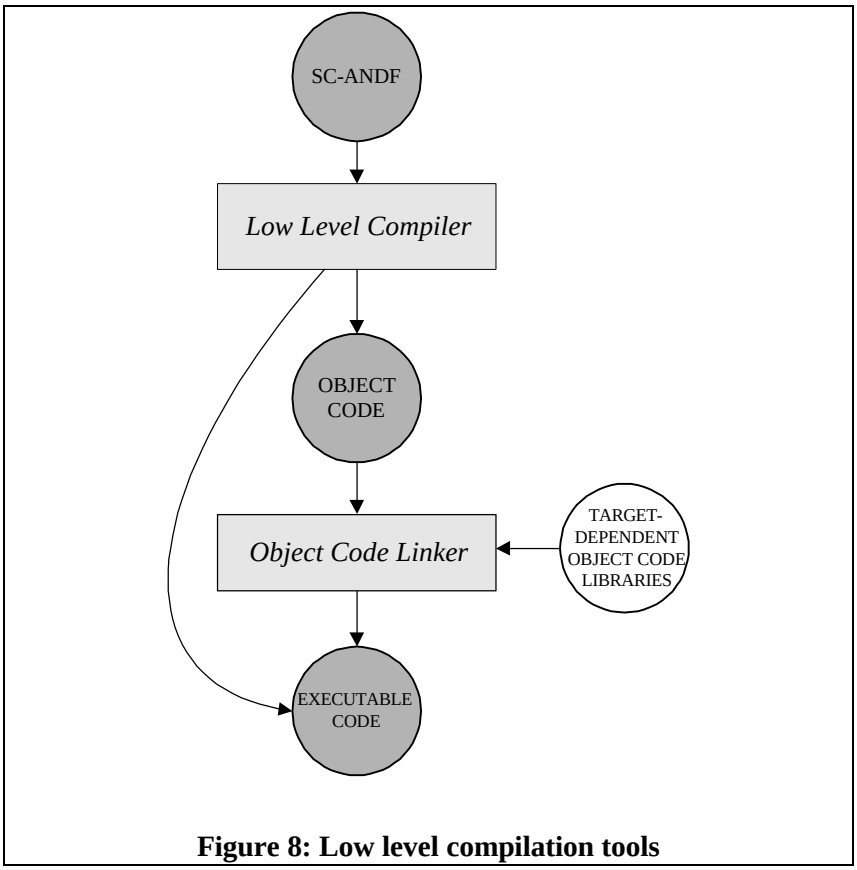

Figure 8: Low level compilation tools