Validation & Verification of an EDA automated synthesis tool

Authors: Di Carlo S., Gambardella G., Indaco M., Rolfo D., Prinetto P.,

Published in the Proceedings of the IEEE 6th International Design and Test Workshop (IDT), 11-14 Dec. 2011, Beirut, LI.

DOI: 10.1109/IDT.2011.6123100

Validation & Verification of an EDA automated synthesis tool[1]

Stefano Di Carlo*, Giulio Gambardella†, Marco Indaco*, Daniele Rolfo†, Paolo Prinetto*
†CINI
Via Ariosto 25, 00185 Roma, Italy
Email: {*FirstName.LastName*}@consorzio-cini.it
*Politecnico di Torino
Dipartimento di Automatica e Informatica
Corso Duca degli Abruzzi 24, I-10129, Torino, Italy
Email: {*FirstName.LastName*}@polito.it

***Abstract*—Reliability and correctness are two mandatory features for automated synthesis tools. To reach the goals several campaigns of Validation and Verification (V&V) are needed.**
The paper presents the extensive efforts set up to prove the correctness of a newly developed EDA automated synthesis tool. The target tool, MarciaTesta, is a multi-platform automatic generator of test programs for microprocessors' caches. Getting in input the selected March Test and some architectural details about the target cache memory, the tool automatically generates the assembly level program to be run as Software Based Self-Testing (SBST). The equivalence between the original March Test, the automatically generated Assembly program, and the intermediate C/C++ program have been proved resorting to sophisticated logging mechanisms. A set of proved libraries has been generated and extensively used during the tool development. A detailed analysis of the lessons learned is reported.

I. Introduction

In software engineering, validation and verification (V&V) methodologies are two sides of the same coin to assess high quality software. V&V consists of one or more techniques applied to software artifacts, at different abstraction layers, during each stage of the software development process (requirements elicitation, architectural design, unit coding and so on). In fact, validation concerns the evaluation of system requirements and the fulfilment of users' real needs. Verification checks the consistency of an implementation w.r.t. its specification. Depending on which V&V stage one is focusing on, implementation and specification can exchange their roles for selected artifacts [1].
In literature Myers' classic book [2] introduces for the first time the "V" model of verification and validation. The distinction between validation and verification is introduced by Boehm [3], who has described validation as "building the right system" and verification as "building the system right".
Several V&V approaches have been presented in literature, most based on modelling software artifacts. In particular, control flaw graph and state machine models are used to describe interactions between software modules [4] [5]. However, they capture just one aspect of dependency. Data flow models make it possible to describe data interactions between parts [6] [7] [8] [9].
From other perspective, symbolic execution techniques aim at defining a set of conditions under which each control flow path is executed, and evaluate their effects on the program state. This approach is useful for checking properties [10] [11] [12] [13] [14] [15] [16].
Finite state verification techniques merge principal characteristics of symbolic execution and formal verification [17] [18] [19] [20].
In this paper we present a custom approach, based on a sophisticated logging mechanism, for validating and verifying the MarciaTesta tool [21]. MarciaTesta is an automated synthesis tool that covers the lack about the generation of ASM test programs for data and instruction cache memories. The overall synthesis process goes through different translation stages. For each one, a different formalism is adopted [22].
The V&V of the entire MarciaTesta is based on verification steps of each translation level and a global validation step.
In section II we briefly introduce MarciaTesta tool. In section III we describe the adopted basic V&V methodology. In section IV, architectural details of modules for V&V and their tasks are presented, while the correctness of main inputs is highlighted in section V. In section VI some experimental results are showed, while section VII draws some conclusions.

II. MarciaTesta Tool

MarciaTesta [21] is a sophisticated automated synthesis tool able to generate assembly programs for both data and instruction cache testing, customized for specific target architecture. Cache memory software testing usually requires the adaptation of a general March Test, according to a selected Software-Based-Self-Test (SBST) methodology [23] [24] [25]. After the test algorithm selection, the test engineer is required to manually code it in assembly code. MarciaTesta has been developed to automate the overall process, thus speeding-up test program generation and providing an error-free assembly code.
To perform the test program generation, different inputs must be defined by the user, as shown in Figure 1 :

[1]This projects is partially funded by Ansaldo STS SpA and FinMeccanica within the "Iniziativa Software" (II ediz) framework

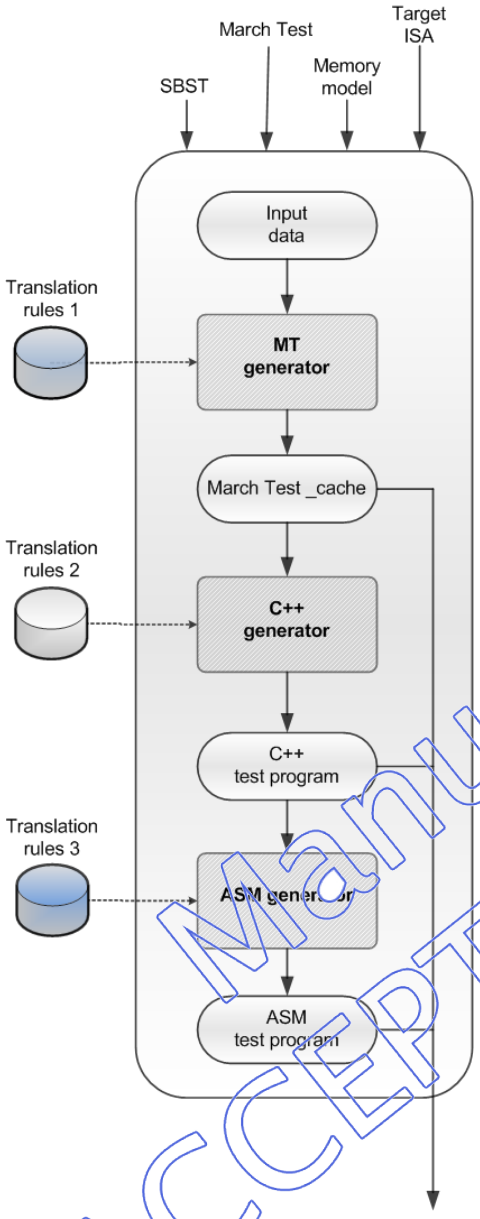

Figure 1. MarciaTesta Architecture

- *Memory model:* it contains parameters to fully customize the data cache memory architecture. These parameters include:
  (i) *Write policy* of the cache.
  (ii) *Word size* (data-width) of the target system.
  (iii) *Base address* of the cacheable memory.

  Moreover, the user must also provide the addressing schema between the main memory and the data cache memory, setting up three additional parameters: *Index*, *Tag*, and *Offset*.
- *March Test:* it contains the selected March Test, denoting with *'U'* and *'D'* respectively the ⇑ and ⇓ addressing orders. It also includes *Addressing Order (AO)* that is the exact sequence in which the cache lines must be addressed during the ascending or descending addressing orders. In addiction the *Data Background (DB) List* is provided. It contains the background patterns required for the implementation of the data and directory array test.
- *SBST:* it specifies the selected SBST methodology. It contains the basic rules to translate each March Test operation into its equivalent for data cache. MarciaTesta offers a set of libraries useful for users to describe the selected *SBST Methodology* (i.e. [22]).
- *Memory model:* it describes the architecture of the target data cache.
- *Target ISA:* it lists the ASM implementation of a minimum set of well defined Meta-ISAs [22] for the target microprocessor.

Both *SBST* and *Target ISA* inputs are initially computed by MarciaTesta to generate a database of rules (Translation rules 1 and 3 in figure 1).
The outputs of MarciaTesta are:

- *March Test_cache:* it is the output of the first translation level. It represents the input March test translated according to the provided SBST methodology.
- *C++ test program:* it is the output of the second translation level. It represents the C intermediate implementation of the translated March Test. It can be useful to test engineers for emulating the test execution.
- *ASM test program:* it is the output of the third translation level. It is the assembly program ready to be run on the target microprocessor.

Figure 1 shows the internal architecture of MarciaTesta. The synthesis process get through three translation levels, performed by the *MT generator*, the *C++ generator*, and the *ASM generator* modules, respectively. The output of each step becomes the input of the next level.
In the first translation stage, the *MT generator* gets in input the selected March Test and the SBST translation rules and it produces a new version of the March Test, compliant with the SBST approach, but still architecture independent.
In the next step, the *C++ generator* codes each march element resorting to a set of C++ methods [22].
Finally, the *ASM generator* replaces each C++ method, listed in *C++ test program*, by a set of assembly instructions.
Implementation details on a first release of the MarciaTesta tool can be found in [21].

III. Basic Approach

In order to verify the correctness of MarciaTesta and validate its results, a verification and validation strategy must be adopted.
Firstly, some preliminary key considerations about MarciaTesta's design are needed:

1) The architecture of the tool has been designed with a modular approach, assigning each task (i.e., translation level) to a specific component.
2) Components can be logically isolated.

3) Inputs and outputs of the tool are represented resorting to different formalisms, but are characterized by a shared well-defined semantic.

Based on these preliminary assumptions, the correctness of the tool has been proved by proving, for each module, the equivalence between its input and its generated output, as shown in Figure 2.

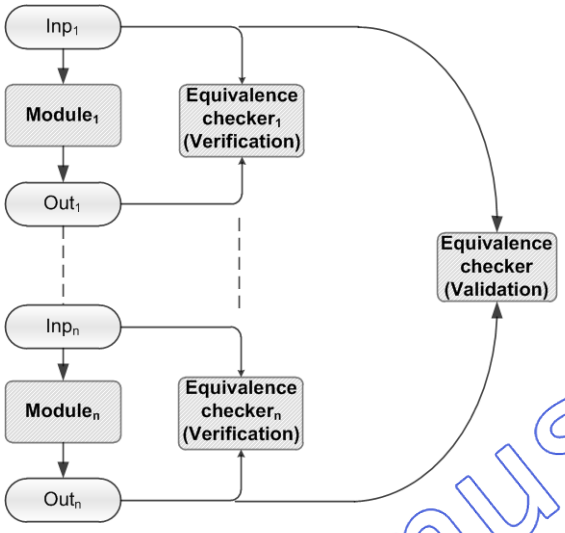

Figure 2. Verification and Validation methodology

Since the output descriptions of each module are expressed in terms of different formalism [22], the equivalence is proved extracting the semantic for each description and proving the equivalence between the two semantics.
The semantic is expressed through a list of C operators that represent the operations performed by a generic March Test for caches during its execution. We highlighted that during cache testing we don't have a direct access to the cache memory, therefore, to write/read a word from the cache a read operation from main memory is needed. The selected operators are :

- *Write memory cell(word, address)*: it writes a *word* into the main memory at specified *address*
- *Read memory cell(address)*: it reads the cache line corresponding to *address*
- *Read and verify for Data(address)*: it reads the data corresponding to address from the cache memory and verifies its correctness
- *Read and verify for Directory(DB TAG,DB OK)*: it reads the data corresponding to *DB_TAG+addressing order* from cache and verifies that it equals *DB_OK*
- *Invalidate cache line(index)*: it invalidates the cache line pointed by *index*
- *Enable_cache*: it enables read and write operations from the cache
- *Disable_cache*: it disables read and write operations from the cache

Based on these considerations, the actual check consists of the following steps performed for each module:

```
FOR i = 1 to n DO
    Extract the semantic from Inp_i and Out_i
    Perform a verification between Inp_i and Out_i
ENDFOR
Perform the validation between Inp_1 and Out_n
```

Both the validation and verification steps are performed resorting to an equivalence check on an automatically generated formal representation of the semantic. Following, the n-version programming paradigm [26] the semantic extractors have been developed by a team different from the implementation team.

IV. V&V Modules

The checker has been implemented as shown in Figure 3. The semantic extractors compute partial outcomes to extract

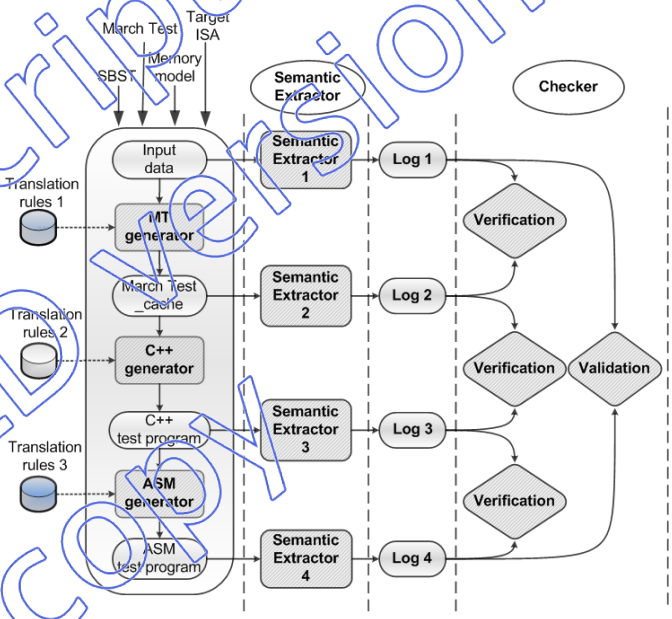

Figure 3. V&V implementation: Modular Checker

the well-defined semantic and create the log files. The V&V modules only require to check the equivalence between log files.
In particular, the checker is composed of two different types of modules. The first implements the *Verification* task, comparing two *Log* files extracted by the neighbour translation levels. The second one compares the first and the last *Log* files, generated from the main input and output of the tool, in order to *Validate* MarciaTesta.

V. Inputs Verification

A portion of the data contained into *Log* files are extracted from the aforementioned inputs of MarciaTesta. For this reason, correctness of the inputs is a crucial task to guarantee the V&V of overall tool. Therefore, in this section, we present an overview of the main inputs and adopted approaches for their validation. For a more detailed description of inputs the

reader may refer to [21].
The main inputs are:

- **March Test**: it had been checked with a visual inspection, verifying if March Test is properly written using *YAUF* [22].
- **SBST**: it had been verified by a cache memory test specialist. He verifies the translation rules correctness.
- **Memory model**: it had been verified by a designer of the target system. He had to ensure the coherency between specified parameters in *Memory configuration* input and their real corresponding with the target system.
- **Target ISA**: it had been verified by an expert on the target processor ISA. He has to verify the correctness of the assembly instructions compounding each Meta-ISA [22].
- **Translation rules**: are a set of libraries that describes the correspondence between two subsequent translation levels. The three translation rules were verified by an expert on SBST methodology, an expert on C++ algorithms and a qualified ASM programmer, respectively.

VI. Experimental results

MarciaTesta has been validated using several campaigns of Validation and Verifications (V&V).
To reach the goal, we choose different input combinations to verify any MarciaTesta operating modes. In particular we selected different March Tests, including: *MATS+* [21], *March C-*, *March U* [28], *PMOVI*, *March SR*, [29], *March LR*, *March B* [30], *March MSS* [31], *March SS* [32], *March G* and *Abraham-Thatte* [33].
Moreover, we considered two *Target processor ISA descriptions* for MicroBlaze [34] and NiosII [35], in order to verify MarciaTesta correctness when test program is generated for both write-back and write-through policies.
Microblaze is a soft core processor designed for *Xilinx* FPGAs, with a Harvard memory architecture, a RISC-like instruction set and a data cache with write-through policy.
NiosII is a 32-bit RISC embedded-processor designed for the *Altera* FPGAs with a data cache with write-back policy.
The actual boards on which we deployed the automatically generated tests are a *Virtex4 ML-403 Embedded Platform* [36] and a *Nios Development Board Cyclone II Edition* [37].
The parameters of *Target memory configuration* have been set, for both microprocessors, to target a system with $2kB$ instruction and data cache, with four words per cache line, and $128kB$ of on-chip memory.
Table I shows, for each analyzed March Test, the test length and the number of ASM rows for both the target architectures. For each V&V campaign, the Checker module has successfully verified the equivalence between the corresponding *Log* files. For completeness an extensive simulation is done to debug each test program.

Some programming bugs are detected by the test program as functional faults and they led to an error detection in the system.

Table I
Experimental results on NiosII and MicroBlaze processors

March test	*Test Length(Xn)*	*Number of ASM rows*	
		NiosII	MicroBlaze
MATS+	5n	50289	42079
March C-	10n	99453	87147
March U	13n	128121	111719
PMOVI	13n	130169	117863
March LR	14n	138365	121963
March SR	14n	140413	128107
March B	17n	164985	140391
March MSS	18n	177277	156779
March SS	22n	218237	197739
March G	23n	226433	201839
Abraham, Thatte	30n	296109	267419

```
...
MOV R3,R4
...
```
(a) Correct code

```
...
MOV R2,R4
...
```
(b) Wrong code

Figure 4. Programming bugs detected as functional faults

Figure 4 shows an example of programming bugs that behave as functional fault. This kind of bugs can be fixed by running the test on a golden environment (e.g., an emulator).
During the V&V campaign some bugs were find in the generated test programs. This kind of bugs don't show up as functional faults, so the test program ends successfully also in presence of faults.

```
...
JMP NOK
...
OK: next test
...
NOK:ERROR
...
```
(a) Correct code

```
...
JMP OK
...
OK: next test
...
NOK: ERROR
...
```
(b) Wrong code

Figure 5. Programming bugs detected by V&V

Figure 5 shows an example of this kind of bugs, founded during V&V. Figure 5 (a) lists an example of the correct code, while Figure 5 (b) lists the bugged one. As we can notice, a changing in the jump label can led to a correct results from the test program, also in presence of a fault.

VII. Conclusion

In this paper the V&V for MarciaTesta tool has been presented. Thanks to the modular implementation of the tool, that consists in different translation levels, a peculiar methodology has been implemented.

For each level of translation, a semantic extractor has been implemented, that allow to validate each module by comparing the log file generated by its input and its output.
The logging mechanism generation is designed on four different abstraction levels:

- March Test input level
- March Test for cache level
- C++ test program level
- ASM test program level

Finally, the verification has been setup comparing the log file generated by the main input of MarciaTesta and its main output. The correctness of the generated test program has been checked on two target architectures.

ACKNOWLEDGMENT

The authors would like to express their sincere thanks to the whole design team of Ansaldo STS for their helpful hints and guidelines.

REFERENCES

[1] M. Young and M. Pezze, *Software Testing and Analysis: Process Principles and Techniques*. John Wiley & Sons, 2005.
[2] G. J. Myers, *Art of Software Testing*. John Wiley & Sons, Inc., 1979.
[3] B. W. Boehm, *Software engineering economics*. Prentice-Hall, 1981.
[4] A. Pretschner, W. Prenninger, S. Wagner, C. Kühnel, M. Baumgartner, B. Sostawa, R. Zölch, and T. Stauner, "One evaluation of model-based testing and its automation," in *Proc. of the 27th International Conference on Software Engineering*, pp. 392–401, 2005.
[5] M. Pezzè, R. N. Taylor, and M. Young, "Graph models for reachability analysis of concurrent programs," *ACM Trans. Softw. Eng. Methodol.*, vol. 4, pp. 171–213, 1995.
[6] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: principles, techniques, and tools*. Addison-Wesley Longman Publishing Co., Inc., 1986.
[7] K. M. Olender and L. J. Osterweil, "Interprocedural static analysis of sequencing constraints," *ACM Trans. Softw. Eng. Methodol.*, vol. 1, pp. 21–52, 1992.
[8] A. Rountev, B. G. Ryder, and W. Landi, "Data-flow analysis of program fragments," in *Proc. of the 7th European Software Engineering Conference held jointly with the 7th ACM SIGSOFT International Symposium on Foundations of software engineering*, pp. 235–252, 1999.
[9] M. Hind, "Pointer analysis: haven't we solved this problem yet?," in *Proc. of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pp. 54–61, 2001.
[10] R. W. Floyd, "Assigning meanings to programs," in *Proc. of the 20th Symposium on Applied Mathematics*, vol. 19, pp. 19–32, 1967.
[11] C. A. R. Hoare, "An axiomatic basis for computer programming," *Communications of the ACM*, vol. 12, no. 10, pp. 576–580, 1969.
[12] R. A. Kemmerer and S. T. Eckman, "Unisex: A unix-based symbolic executor for pascal," *Softw. Pract. Ex. Trans.*, vol. 15, no. 5, pp. 439–458, 1985.
[13] S. L. Hantler and J. C. King, "An introduction to proving the correctness of programs," *ACM Trans. Comput. Surveys*, vol. 8, pp. 331–353, 1976.
[14] W. E. Howden, "Symbolic testing and the dissect symbolic evaluation system," *IEEE Trans. Softw. Eng.*, vol. 3, pp. 266–278, 1977.
[15] W. E. Howden, "An evaluation of the effectiveness of symbolic testing," *Softw. Pract. Ex. Trans.*, vol. 8, pp. 381–397, 1978.
[16] L. Clarke, "A system to generate test data and symbolically execute programs," *IEEE Trans. Softw. Eng.*, vol. SE-2, no. 3, pp. 215–222, 1976.
[17] M. B. Dwyer, L. A. Clarke, J. M. Cobleigh, and G. Naumovich, "Flow analysis for verifying properties of concurrent software systems," *ACM Trans. Softw. Eng. Methodol.*, vol. 13, pp. 359–430, 2004.
[18] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre, "Software verification with blast," in *Proc. of the 10th International Conference on Model checking software*, pp. 235–239, 2003.
[19] D. Jackson, "Alloy: a lightweight object modelling notation," *ACM Trans. Softw. Eng. Methodol.*, vol. 11, pp. 256–290, 2002.
[20] G. Holzmann, *Spin model checker, the: primer and reference manual*. Addison-Wesley Professional, first ed., 2003.
[21] S. D. Carlo, G. Gambardella, M. Indaco, P. Prinetto, and D. Rolfo, "Marciatesta: an automatic generator of test programs for microprocessors' data caches," in *Submitted to 20th Asian Test Symposium*, 2011.
[22] S. D. Carlo, G. Gambardella, M. Indaco, P. Prinetto, and D. Rolfo, "A unifying formalism to support automated synthesis of sbsts for embedded caches," in *Proc. of the 9th East-West Design & Test Symposium*, pp. 39–42, 2011.
[23] N. Kranitis, A. Paschalis, D. Gizopoulos, and G. Xenoulis, "Software-based self-testing of embedded processors," vol. 54, no. 4, pp. 461–475, 2005.
[24] S. Di Carlo and P. Prinetto, *Models in Hardware Testing*, ch. Models in Memory Testing, From functional testing to defect-based testing, pp. 157–185. Springer, 2010.
[25] Y.-C. Lin, Y.-Y. Tsai, K.-J. Lee, C.-W. Yen, and C.-H. Chen, "A software-based test methodology for direct-mapped data cache," in *Proc. of the 17th Asian Test Symposium*, pp. 363–368, 2008.
[26] A. Avizienis, "The n-version approach to fault-tolerant software," *IEEE Trans. Softw. Eng.*, vol. SE-11, no. 12, pp. 1491 – 1501, 1985.
[27] I. Mrozek and V. N. Yarmolik, "Mats+ transparent memory test for pattern sensitive fault detection," in *Proc. of the 15th International Conference on Mixed Design of Integrated Circuits and Systems*, pp. 493–498, 2008.
[28] A. J. van de Goor and G. G.N., "March u: a test for unlinked memory faults," in *Proc. of the IEEE Circuits, Devices and Systems*, pp. 155–160, 1997.
[29] A. van de Goor, G. Gaydadjiev, V. Mikitjuk, and V. Yarmolik, "March lr: a test for realistic linked faults," in *Proc. of the 14th VLSI Test Symposium*, pp. 272–280, 1996.
[30] A. Van De Goor, "Using march tests to test srams," *IEEE Trans. Design Test of Comput.*, vol. 10, no. 1, pp. 8–14, 1993.
[31] G. Harutunvan, V. Vardanian, and Y. Zorian, "Minimal march tests for unlinked static faults in random access memories," in *Proc. of the 23rd VLSI Test Symposium*, pp. 53–59, 2005.
[32] S. Hamdioui, A. J. van de Goor, and M. Rodgers, "March ss: A test for all static simple ram faults," in *Proc. of the 2002 IEEE International Workshop on Memory Technology, Design and Testing*, pp. 95–100, 2002.
[33] R. Nair, S. Thatte, and J. Abraham, "Efficient algorithms for testing semiconductor random-access memories," *IEEE Trans. Comput.*, vol. C-27, no. 6, pp. 572–576, 1978.
[34] Xilinx, *MicroBlaze Processor Reference Guide*, 2004.
[35] Altera, *Nios II Processor Reference Handbook*, v7.2 ed., 2007.
[36] Xilinx, *ML403 Evaluation Platform*, v2.5 ed., 2006.
[37] Altera, *Nios Development Board Cyclone II Edition Reference Manual*, v1.3 ed., 2007.