Chapter 2

RCS blueprint/content

this chapter is temporal which could be divided into other chapters like Introduction, Evidence, Requirements, Prototype.

2.1 contribution/final aim

2.1.1 goal

- To accelerate the development of verification technology through the development of better tools, greater interoperability, and realistic benchmarks.
- To provide a focus for the verification community to ensure that the research results are relevant, replicable, complementary, and cumulative, and promote meaningful collaboration between complementary techniques.
- To provide open access to the latest results and educational material in areas relevant to verification research.
- Collect a significant body of verified code (specifications, assertions, derivations, proofs, models, implementations) that addresses challenging applications.
- Identify key metrics for evaluating the scale, efficiency, depth, amortization, and reusability of the technology.
- Enumerate challenge problems and areas for verification, preferably ones that require multiple techniques.
- Identifying (and eventually standardizing) formats for representing and exchanging specifications, programs, test cases, proofs, and benchmarks, to support tool interoperability and comparison.
- Defining quality standards for the contents of the repository.

2.1.2 theory

The project will deliver a comprehensive and unified theory of programming, which covers all the major programming paradigms and design patterns appearing in real computer programs, and nearly all the features of the programming languages of the present and foreseeable future.The theory will include a combination of concurrency, non-determinism, object orientation and inheritance. Many disciplined software design patterns will be proven sound, and analysis algorithms will be developed that check whether a program observes the relevant disciplines [4] .

- Domains: Control systems, fault tolerance, distributed computing, networking, security, scientific computing.
- Models/Logics: Golden models, abstraction, modularity, unification.
- Programs: Semantics, refinement, types/assertions, analyses, optimization.
- Tools: Construction, verification, composition, bridging semantic gaps, methodologies.

2.1.3 content

- verified programs (which will be illustrated in chapter Evidence in detail) All available forms of machine-readable documentation – specifications, dependability analyses, design trajectories, development histories, internal interface specifications, simulators, test case generators, regression tests, and even conjectures, theorems, and proofs – will accompany the codes.
- libraries of bare specifications independent of code: These may include formal definitions of standard intermediate languages and machine architectures, communication and security protocols, interfaces of basic standard libraries, etc [4] .
- There will also be libraries of application-oriented specifications, which can be re-used by future applications programmers in particular domains (such as railway networks). Some of these will be supported by automatic code generators, perhaps implemented within a generic declarative framework [4] .
- tool
  - Integrated verification environments
  - Language front-ends
  - Static Analyzers
  - Test case generators
  - Theorem Provers

  - Model Checkers
  - Graphical User Interfaces
  - Program Synthesizers
  - Integrated Builds

- Benchmarks
- Case Studies
- Interoperability
  - Interchange formats, mappings between models, logics, glue code.
  - Models: state machines, automata, timed automata, hybrid automata, abstractions.
  - Language syntax and semantics: Logics, specification languages, programming languages.
  - Representations of proofs, test cases, counterexamples.
- Verified libraries: STL, openSSL, core Java, Bouncy castle, openPGP, glibc, GMP.
- Tutorials
- Advanced search capability
- Keywords and ontologies
- Forums
- Challenge problems: File synchronization, File system, web server, kernel, TCP/IP, SSL, compression, theorem prover kernel, cache consistency, separation kernel, compilers, virtual machines, build tools, fault-tolerant architectures, model reduction for hybrid systems, aspect extraction.
- Generic properties: Absence of runtime errors, data consistency, timing behavior, accuracy, type correctness, termination, translation validation, serializability, memory leaks, information hiding, representation independence, and information flow.

2.1.4 Deliverables

- A comprehensive theory of programming that covers the features needed to build practical and reliable programs.
- A coherent toolset that automates the theory and scales up to the analysis of large codes.
- A collection of verified programs that replace existing unverified ones, and continue to evolve in a verified state.

- An archive for verification tools, benchmarks, models, specifications, and verified software.
- A laboratory for the verified software project.
- A portal for attracting participation and exchanging research results.
- Main idea: Drive problem-centric research through published standards, tools, benchmarks, metrics, and challenges.

2.1.5 the target software

- Libraries: Software implementing a family of functions invoked through a programmer interface are packaged as libraries. The formal specification of these interfaces is a key challenge. The verification challenges range from demonstrating the absence of runtime errors to conformance with the interface specification. Examples of libraries include OpenSSL, the Eiffel and Java class libraries, scientific software, and computer algebra systems. Verification tools themselves rely on libraries for multi-precision arithmetic, binary decision diagrams, and polyhedral manipulations.
- embedded devices: These are devices containing software that interact with the physical world in the form of device drivers, smart cards, medical devices, airbag systems, and controllers. Lightweight runtime environments supporting schedulers, file systems, database, networking, and scripting, are also suitable candidates for formalization and verification.
- systems: Libraries and embedded devices involve precise and well-specified interfaces, whereas the verification of systems runs squarely into the complication of designing software with respect to diverse and complex requirements. Here the formal architecture of the software is critical to the success of any verification effort. Candidate examples include a hypervisor and the MINIX3 operating system.
- applications: Various safety or security critical applications such as an airtraffic control system, a secure communication system built on a hypervisor architecture, a certifying compiler, and a web server. These applications depend critically on the properties of the underlying systems.

2.1.6 Source of verified software

- Some of the code and specifications may be collected from critical projects that have already undergone a formal analysis
- Some of the code will be developed by rational design methods from specification, using interactive specification checkers and development aids
- Other code will be generated automatically, together with its assertions and its proofs, by special-purpose program generators, taking an application-oriented specification as input

- Some of the code will be taken from existing class libraries for popular programming languages
- Some even from well-written open source application codes

Early examples may be selected from embedded applications (e.g., smart-cards and sensors); from critical control systems (reactors, flood gates); and from open sources of off-the-shelf software (e.g., the Apache web-server).

2.1.7 Develop Steps

We expect that the plans for the project will include a specification phase, an integration phase and an application phase, each lasting around five years [4]. At any given time during the course of the project, the repository will contain all the code verified so far, which will be used repeatedly as regression tests for the evolution of the tool-set, and as a measure of the progress of the project. In addition, the repository will contain a number of challenge problems, which have been selected by the research community as worthy candidates and ripe for verification.

- During the specification phase, a network of repositories will be established and populated with challenge specifications and codes. Some of the smaller challenges will be totally verified using existing tools, and some of the larger ones will be verified to a formally specified level of structural soundness and serviceability. A formal tool-bus will establish communication at the file level between the more widely used tools.
- During the second integration phase, the tools will evolve to exploit even closer inter-working, and performance will be improved by introduction of more inter-active interfaces between them. New tools will emerge as user needs become more apparent. Medium-sized programs will be fully verified and some of the largest programs will be verified to a high level of structural integrity.
- During the final phase, the pace of progress will accelerate. A comprehensive, integrated tool suite that is convenient to use will permit the collaborative verification of large-scale applications, culminating in achievement of the million-line goal.

2.1.8 Methodology

- Specification languages: Intensional/extensional properties ranging from crash-freedom to resource usage and total correctness.
- Programming language design: Safety, integrating specification, types, and implementation, improved static analysis.
- Program generation techniques: Domain specificity, semantic equivalence, proof preservation, optimizations from annotations.

- Verification by construction: Modular refinement and abstraction, refinement theories and patterns.

2.1.9 other aspects

The project will deliver a repository of verified software, containing hundreds of programs and program modules, and amounting to over a million lines of code. Each program will have been mechanically checked by one or more of the program verifiers in the tool-set.

Application domain [4] : They will cover a wide range of applications, including smart cards, embedded software, device routines, modules from a standard class library, an embedded operating system, a compiler for a useful language (possibly smart card Java), parts of the verifier itself, a program generator, a communications protocol (possibly TCP/IP), a desk-top application, parts of a web service (perhaps Apache). The programs will use a variety of design patterns and programming techniques, including object orientation, inheritance, and concurrency.

Levels of verification [4] : There will be a hierarchy of recognized levels of verification, ranging from avoidance of specific exceptions like buffer overflow, general structural integrity (or crash proofing), continuity of service, security against intrusion, safety, partial functional correctness, and (at the highest level) total functional correctness. Each claim for a specific level of correctness will be accompanied by a clear informal statement of the assumptions and limitations of the proof, and the contribution that it makes to system dependability. The progress of the project will be marked by raising the level of verification for each module in the repository. Since the ultimate goal of the project is scientific, the ultimate level achieved will always be higher than what the normal engineer and customer would accept.

A spectrum of specification properties: No crashes/No type errors/No exceptions/No deadlock/Preconditions hold/ control and functional behavior (partial specification to complete specification)

The target levels of certification [2] will be appropriate to the criticality of the application, from basic soundness (crash-proofing), through levels of security, immunity to attack, continuity of service, observance of safety constraints, and ultimately, total conformance to functional specification.

Benefit [4] : There will remain the considerable task of transferring the technology developed by the project into professional use by programmers, for the benefit of all users of computers. This is where all the educational, social, commercial, legal and political issues will come to the fore. The scientists will have made their main contribution in demonstrating the possibility of what was previously unthinkable. Scientific research will continue to deliver practical and theoretical improvements at an accelerated rate, because they are based on the achievement of the goals of the Grand Challenge project.