metaware.

the complete IT modernization solution

The Origin of Refine®

A Metaware White Paper

EXECUTIVE SUMMARY

Refine® emerged as a widely used technology for software analysis and re-engineering in the 1990s by integrating several groundbreaking technical innovations:

- A flexible representation of software source code as annotated abstract syntax trees (ASTs) in a persistent object repository
- A powerful grammar system for rapid generation of code parsers and printers to translate between ASTs and source files, preserving comments and formatting
- A very-high-level, wide-spectrum language for analyzing and transforming ASTs, with support for set theoretic types, first order logic, program transformation rules, syntax-directed pattern-matching and conventional procedural constructs
- A runtime environment with support for generic operations on trees including copying, comparison, substitution and application of transformation rules
- A GUI toolkit with tables, diagrams, hypertext, outlines, and a flexible multi-window hyperlinking mechanism
- A development environment with dynamic linking, unit-level incremental compilation, debugging, object inspection, and profiling/metering tools

With Refine®, software engineers achieve order of magnitude reductions in the time required to develop automated tools for language translation, extraction of design models, defect analysis, coding standard checking, and consolidation of redundant code. In some cases, Refine® was used to create automated software processing applications previously considered too difficult to be economically viable.

This paper describes the history of Refine® from its origin in a Silicon Valley software research laboratory in the 1980s through its commercialization and widespread adoption in the 2000s and beyond.

1 KESTREL AND THE CHI SYSTEM

Refine® grew out of a revolutionary research system called CHI developed at the Kestrel Institute in Palo Alto.

1.1 Kestrel Institute

In 1981, Dr. Cordell Green founded Kestrel Institute, a non-profit computer science research institute in Palo Alto, to pursue research towards formal and knowledge-based methods for automation of the software development process. Dr. Green, formerly a professor in the Computer Science department at Stanford University, had already been recognized as a top-flight computer scientist based on his key contributions in the field of logic programming, for which he was awarded the Grace Murray Hopper Award by the Association for Computing Machinery.

Kestrel Institute quickly attracted top-flight researchers in the field of automated software development, including faculty and graduate students from Stanford University, the California Institute of Technology (CalTech), the University of California at Santa Cruz (UCSC) and the University of Illinois. Research grants were provided by a several US government agencies including the Defense Advanced Research Projects Agency (DARPA), the Office of Naval Research (ONR), and the Air Force Office of Scientific Research (AFOSR).

1.2 The CHI System

In the early 1980's, Kestrel Institute created the CHI system to explore the possibility of automating software development using knowledge representation and inference techniques from the fields of artificial intelligence. Much of the seminal work in designing and creating CHI was done by Jorge Phillips and Stephen Westfold, two graduate students at Stanford whose PhD dissertations provided the intellectual foundation underlying the CHI system. Other key members of the CHI team included Gordon Kotik, a graduate student at UCSC, Allen Goldberg, a professor at UCSC, and Thomas Pressburger, a graduate student at Stanford.

Four of the key ideas in CHI were:

- A very-high-level (VHL), wide-spectrum specification language named V
- Representation of software in an object-oriented repository
- The explicit representation of programming knowledge as program transformation rules
- Self-application of CHI in order to allow CHI to implement and optimize itself

1.3 The V Language

The goal of the V language was to represent a wide-spectrum of programming language concepts, from VHL concepts such as mathematical set theory and first-order logic to conventional low-level constructs such as if-then-else statements, for loops, and assignment statements. The use of VHL constructs enabled short, clear specifications of algorithms, which are faster to write and easier to read and maintain. The lower-level constructs served two vital purposes. First, it allowed the V language to express transformation rules that described how to implement VHL constructs (eg, sets) in terms of low-level constructs (eg, lists or arrays). Second, it provided a natural means of describing inherently state-oriented algorithms, which in some cases were better described using low-level language constructs than the VHL constructs.

A crucial aspect of V was that it was designed to be an *executable* specification language. This differentiates V from several other high-level specification languages for which a human was required to translate the specification into an implementation. One of the key goals of the CHI project was to automate the translation of specifications into implementations, and allow ongoing modification and maintenance of an application to take place at the specification level. This pioneering aspect of V was very influential, as manifested in the variety of executable specification languages in use today, and in the increasing demand for executable versions of specification techniques such as Executable UML.

1.4 Program Transformation Rules

One of the most important language constructs in V was the *transformation rule*, which is an if-then rule of the form:

```
rule Rulename (p1, p2, …)
      Precondition(p1, p2, ..) -->   Postcondition(p1, p2, ..)
```

where *Precondition* and *Postcondition* are logical expressions involving parameters p1, p2, .. and optionally global variables. The meaning of such a rule is "test to see if the Precondition is true, and if so, modify the state of the system so that the Postcondition becomes true".

The power of if-then rules to represent domain knowledge had been demonstrated in expert systems, and had shown significant potential, particularly in *analysis* applications such as medical diagnosis (MYCIN) and mass spectra analysis (DENDRAL). These expert systems used the knowledge embodied in their if-then rules in order to reason *backwards* from observations such as disease symptoms to deduce underlying causes, eg, diseases.

In contrast, the primary use of transformation rules in V was to represent programming knowledge that could be used in automated *synthesis*, that is, the creation of low-level software implementations from VHL specifications. Such rules are termed program transformation rules, and have the general form:

```
rule Rulename (p)
   Precondition(p) --> Postcondition(p)
```

where:

- *p* is an element in a program (i.e., a statement, expression, declaration etc)
- *Precondition(p)* describes the current state of *p*, for example, by asserting that *p* is a variable whose data type is "set of integers".
- *Postcondition(p)* describes the next state of *p*, for example, by asserting that *p* is a variable whose data type is "linked list of integers". In this case, the rule represents the programming knowledge that an abstract set can be implemented as a linked list.

The precondition and postcondition of the rule can also describe conditions on objects related to p. For example, if p is a variable declaration, a rule could also involve conditions on the references to p within the program. This allows program transformation rules to take into account the *context* of program element, instead of being limited to the element itself. This contextual information would allow, for example, a synthesis application that decides how to implement a set based on whether it is important to optimize the speed of membership tests or the size of the data structure used to implement the set.

Below is an example rule that describes how to simplify an If-Then-Else statement with a negated condition by switching the Then and Else branches:

```
rule simplify-if (a)
   a = 'if ~@cond then @stat1 else @stat2'
      -->  a = 'if @cond then @stat2 else @stat1'
```

The rule makes use of syntax-directed pattern matching to express the precondition and postcondition of the transformation using the syntax of the language being transformed.

An important benefit of the rule formalism is the separation of the *representation* of knowledge (as transformation rules) from the *use* of the knowledge in a particular application. For example, a single rule could be used in multiple synthesis algorithms without any modification of the rule itself. The synthesis algorithm could express decisions such as which order to visit the elements in a program, which rules to apply and which order, etc.

1.5 Representation of Software in an Object-Oriented Repository

CHI was designed to automate software development by codifying and applying software knowledge expressed as program transformation rules. To meet this goal, CHI had to provide a detailed

representation of the software source code being transformed—this representation served as the data structures to be transformed by the transformation rules.

The CHI team designed a novel software representation that combined the three key concepts: *objects*, *abstract syntax trees (ASTs)*, and *logic programming*. The result is an object-oriented AST representation of software in which nodes of the ASTs were objects. Moreover, each object class is interpreted as a logical predicate on objects (a class test), and inheritance is modeled as logical implication. For example, if the object class *Expression* has a subclass named *AdditionExpression*, then the following implication holds:

```
AdditionExpression(x) => Expression(x)
```

The attributes of object classes (also known as slots or fields) are modeled as maps (functions) whose domains were restricted to the corresponding class. For example, the mapping from an *IfStatement* to the *Expression* that models its condition is defined as:

```
Condition: map(IfStatement, Expression)
```

This object-oriented AST representation combined elegantly with the V language to allow the succinct description of programming knowledge as transformation rules. Moreover, the representation was extensible and proved to be an excellent means to represent not only source code, but also related software information such as control flow graphs, test cases and documentation. With the advent of Refine®, this representation formed the cornerstone of the language modeling abilities that make Refine® extensible to analyzing and transforming code in any programming language.

1.6 Self-Application of CHI

A fundamental principle of CHI was that it should be *self-applied*: the specification of CHI should be written in the V language, and the implementation of CHI should be produced automatically from the specification by CHI, using programming knowledge expressed in CHI's knowledge base of program transformation rules. This principle and its consequences were described in the groundbreaking 1981 paper "Knowledge-Based Programming Self-Applied" by Cordell Green and Stephen Westfold.

One motivation for this self-application was to allow CHI to improve its own implementation as its knowledge base of programming expertise was expanded. Another important benefit was to make the developers of CHI become users of CHI, to stress test the technology and ensure that the developers were motivated to make CHI reliable, easy to use, and applicable to the real-world challenge of building a complete programming environment.

The self-application of CHI was achieved by the bootstrapping technique often employed by developers of compilers. First, a compiler for the V language was written in Lisp. Then a new version of the compiler was written as V transformation rules, and the initial compiler was used to compile the V version of the compiler. Finally, the V version of the compiler was used to compile itself, completing the bootstrapping procedure. The target language for the V compiler was Lisp, and a Lisp compiler was used as the final phase of the V compiler.

2 REASONING SYSTEMS

By late 1984, the CHI project had achieved impressive results in automating software development, and the CHI team realized there was an opportunity to create a commercial version of the CHI technology. However, Kestrel Institute, as a non-profit, could not conduct a for-profit business selling such a product. For this reason, Cordell Green, Gordon Kotik and other members of the CHI team founded a new for-profit company named Reasoning Systems in January 1985 to commercialize the CHI technology.

The initial members of Reasoning Systems included several members of the CHI team and also additional computer scientists and business developers.

As the Director and Chief Scientist of Kestrel Institute, Cordell Green was hired as the Chairman of the Board of Directors. Dr. Green was presented the Grace Murray Hopper Award by the Association for Computing Machinery (ACM) for establishing the theoretical basis for the field of logic programming. Green was named the recipient of the 8th International Stevens Award for contributions to methods for software and systems development. Dr. Green developed the foundation theory for Logic Programming, which also formed the foundation for the Deductive Data Base field, as well as many formal, inference-based AI systems. He has made several seminal contributions to the field of program synthesis, including a paper that provided the basis for the Refine® language. Dr. Green has served at the Darpa Information Processing Techniques Office as Research and Development Program Manager for artificial intelligence, planning the DARPA Speech Understanding Research Project and serving on the steering committee. At Darpa, he also served as an assistant to Dr. Larry Roberts while he was creating the Arpanet. Dr. Green was Chief Scientist and Program Manager for Computer Science at Systems Control Inc. He served as Research Mathematician in the Artificial Intelligence Group at Stanford Research Institute. He has served as Lecturer and Assistant Professor of Computer Science at Stanford University. Dr. Green received a BA and BS from Rice University as well as MS and PhD from Stanford University.

John Anton was hired to serve as President. Anton has expertise in the areas of control theory, signal processing, software technologies, and their application. As VP for Advanced R&D at Systems Control, Inc., he led a team that built the Reconfigurable Inflight Control System (RIFCS) for McDonnell Aircraft – built using technology from CTRL C (the predecessor to today's Matlab), which was also built under his leadership. Anton was an Adjunct Professor at Santa Clara University where, for 10 years, he taught courses in linear systems theory, optimal and stochastic control, and decision theory. He received a Ph.D. in Applied Mathematics from Brown, a B.S. from Notre Dame, and was a Fulbright Fellow at the Technische Hochschule, Germany.

Gordon Kotik served first as the development manager and later as Vice President of Engineering. Kotik holds a BA in Mathematics from Princeton University and a Masters Degree in Computer Science from the University of California.

Reasoning Systems was founded with the goal of selling a general-purpose programming environment based upon the use of a VHL language and knowledge-based development environment. The first two commercial customers were Lockheed Corporation, an aerospace company now known as Lockheed Martin, and Thomson-CSF, a French electronics and defense contract now known as Thales. These customers entered into contracts with Reasoning Systems to purchase software licenses and consulting to apply the technology to their application areas.

Reasoning Systems started to productize part of the CHI system. The name *Refine®* was chosen for the resulting product, to emphasize the use of program transformation rules to Refine® VHL software specifications into running implementations. The VHL language was also renamed from V to Refine®.

The first version commercial version of Refine® was delivered to Lockheed and Thomson-CSF in mid-1980s, running on the Symbolics Lisp Machine hardware platform, which provided perhaps the most advanced and full-featured programming environment of the day. Refine® was designed to add VHL language and knowledge-based programming capabilities into the Lisp Machine environment as a seamless extension that preserved and leveraged the features of the underlying Symbolics Lisp environment. The resulting environment provided a tightly integrated editor, compiler, and debugger, as also a GUI using a mouse and windowing system, which was very innovative in 1985.

3 THE SOFTWARE REFINE®RY

Reasoning Systems continued to expand and develop Refine® from 1986-1988, but it proved more difficult than anticipated to market a radically different model of software development based on a VHL language and Symbolics hardware. In addition, Reasoning Systems was simultaneously promoting Refine® for many different application domains including aerospace, communications, and software re-engineering.

Reasoning recognized customers were seeing huge value in applying Refine®'s unique software transformation capabilities to process existing legacy applications written in conventional languages such as COBOL and Fortran. Kotik therefore advocated a strategic focus on a single application domain: *software-processing applications for legacy systems*. Software-processing applications were defined as applications for which source code was the input and/or output of the application, for example:

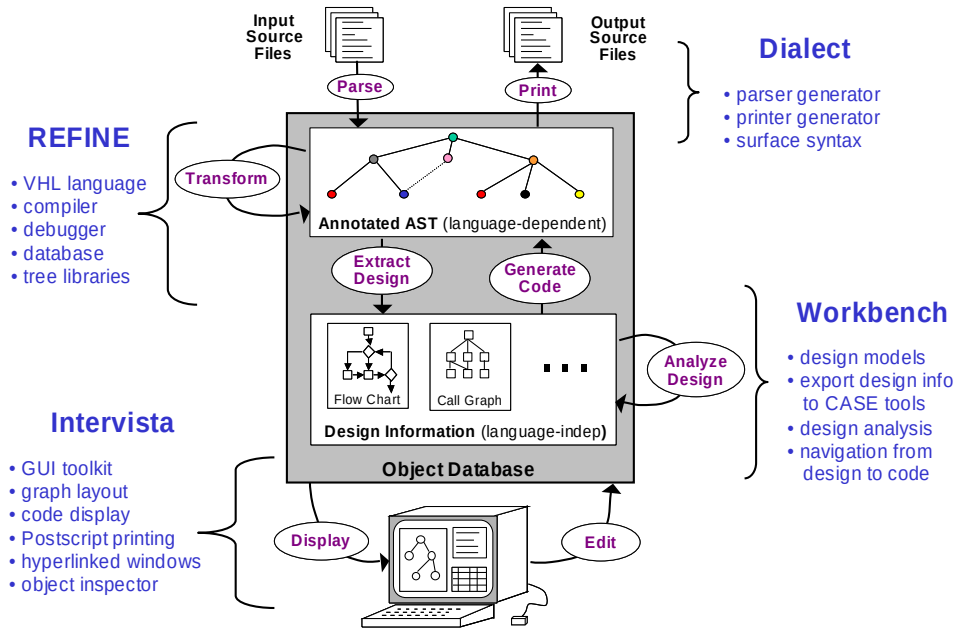
- *Re-engineering applications* to translate legacy systems to new languages, platforms, and databases
- *Reverse engineering* applications to extract design information such as structure charts and control-flow graphs from legacy source code
- *Software quality* applications including generation of test cases from legacy source code and automated code analysis to identify bugs and other defects.

Refine®'s VHL language and object-oriented AST representation of source code provided a powerful mechanism for automatic code analysis and code re-engineering. The productivity benefits of Refine® in this application domain were so compelling that customers were willing to use of a novel VHL language and specialized Symbolics hardware in order to realize the order-of-magnitude or greater improvements in development time.

In 1988, Reasoning Systems focused its strategy and efforts on software processing, extending and specializing Refine® as a development environment for rapid creation of software processing applications. The improvements included:

- Introduction of a more powerful LALR(1) parsing technology called *Dialect* to allow more rapid development of *language models* for legacy languages. Each language model included a parser, printer, internal AST representation and static semantic analyzer for source code in a particular language
- Introduction of a graphics toolkit named *Intervista* to allow automatic and/or interactive creation of diagrams, tables, outlines and hypertext source displays
- Creation of *Workbench*, a set of language-independent representations for common design abstractions including structure charts, control flow graphs, and data flow information, including GUI visualizations using the graphics toolkit
- Modifying the Refine® compiler to generate Common Lisp, and porting Refine® to run on other hardware platforms including Unix workstations from Sun, HP, and IBM.

The improved product was named *Software Refine®ry*, and included Refine®, the new parsing system *Dialect*, the graphics toolkit *Intervista*, and the *Workbench* component for language-neutral design abstractions. The diagram below describes its components:



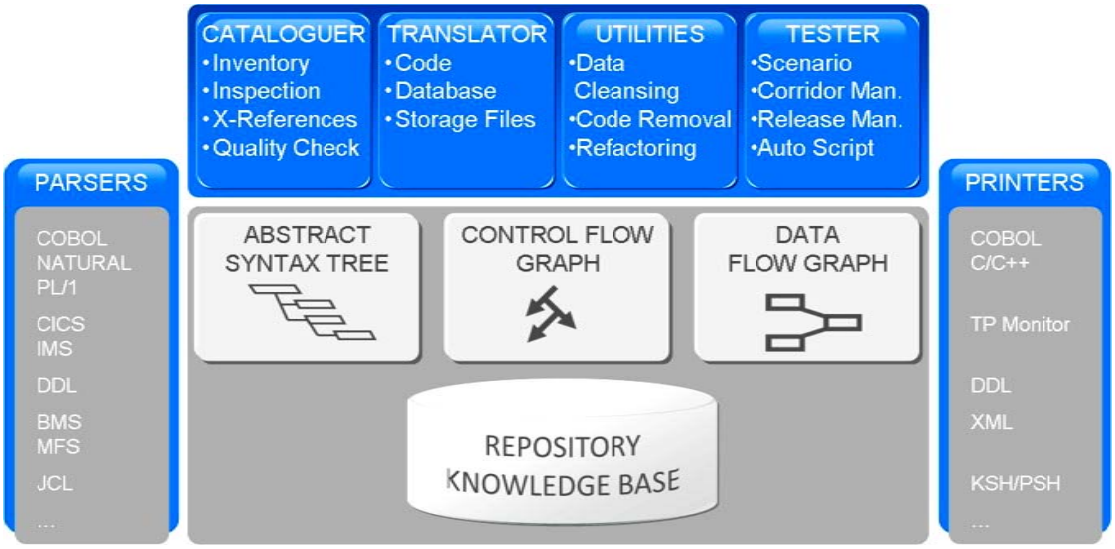
Released in June 1990, Software Refine®ry was the industry's first commercial transformation software. Reasoning Systems grew its customer base rapidly due market demand for a flexible and powerful toolkit for software analysis and re-engineering of legacy systems. As of the mid-1990s, Software Refine®ry encapsulated the result of roughly 100 man-years of software development effort.

4 REFINE® BY METAWARE

From the start in 1995, Metaware has developed and used Refine® as an OEM license. In 2004, Metaware acquired the whole Intellectual Property of Refine® to respond increasing requirements of enterprise legacy modernization across Europe.

With the most powerful and scalable reverse engineering technology commercially available, Metaware developed Refine® Solutions upon Refine® to provide comprehensive solutions legacy modernization, primarily for Bull GCOS, HP3000 and IBM MVS and z/OS.

Refine® Solutions uniquely provides highly accurate analysis and transformations, as measured by a defect injection rate in range of 10^{-5} to 10^{-6}. Using Refine® enables the highest levels of productivity, quality and standardization, regardless of volumes and complexity, and results in higher ROI.



5 SUCCESS STORIES

From 1989 to the mid 1990's, Refine® has been adopted by more than 200 organizations, public and private IT laboratories, for the development of software-processing applications. During this time, more than 100 papers were published describing applications created with REFINE®. Some of the highlights of this era of REFINE® are described below.

5.1 Boeing

Boeing was one of the earliest and most prolific of the companies that adopted REFINE® for automated code processing. Philip Newcomb, a researcher working in a software R&D lab in Boeing's Research and Technology division, spearheaded this adoption. During this time, Boeing contracted Reasoning to create language models for C, Fortran, COBOL, JCL, Ada, Jovial and other languages. Reasoning retained the rights to this software, and introduced some of these language models as separate products in product line called Refine® Language Tools. These products served as REFINE® add-ons to enable faster development of software-processing applications for those languages.

Using the language models, Boeing and Reasoning collaborated to create dozens of groundbreaking code processing applications including:

- A test verification system for Ada that pioneered the use of theorem proving technology to automatically determine test case feasibility. This system was capable of generating test case specifications for 21 structural test coverage criteria based on control- and data-flow analysis of Ada routines.
- A tool that supports interactive modularization of large COBOL programs by replacing a single large compilation unit with a functionally equivalent collection of smaller units. The tool was able to properly preserve all dependencies among data declarations, linkage and working storage sections, CALLs and entry points.
- A tool that derived an object-oriented application data model from COBOL source code, to facilitate re-engineering COBOL into an object-oriented language. This tool used a broad range of analytic and transformational techniques including program slicing, alias analysis, and merging of redundant (copied) routines. The tool was applied to COBOL programs with more than 100K LOC.

5.2 IBM

IBM adopted REFINE® in multiple divisions in the 1990s and created several advanced software-processing applications, the most significant of which was a program understanding tool for the PL/AS language, an IBM proprietary derivative of PL/I.

This tool was used extensively to improve the quality of SQL/DS, a large relational database management system with over 3,000,000 lines of PL/AS code and a large customer base. The tool's capabilities included checking coding standards, defect finding and filtering, design-quality metrics analysis, and identification of dead code. IBM was able to use the tool to correlate design errors and defects, showing that design errors led to 43% of the total product defects. Analysis was also performed to correlate defects with code complexity, code modification history, and the size of individual code modifications.

5.3 Accenture

Accenture was an early and significant adopter of REFINE®, and used REFINE® to create several re-engineering applications.

The most significant tool created by Andersen Consulting using REFINE® was a software re-engineering workbench named BAL/SRW for recovering the designs from IBM 370 BAL assembler programs. Its capabilities included design extraction and presentation, program slicing and dicing, and recognition of stereotypical assembler coding patterns, unreachable code elimination and program design editing. BAL/SRW was used successfully on a number of commercial engagements with major insurance and telecommunications customers of Andersen Consulting.

Accenture also created an innovative tool for recognizing higher-level programming *concepts* in COBOL programs and performing concept-based transformations. An example of such a concept is a sequential array search. The tool supported declarative specifications of concepts using patterns and constraints, and fully automated identification of concepts within COBOL programs.

5.4 EPRI

EPRI (Electric Power Research Institute) is an independent, nonprofit center for public interest energy and environmental research, focused on electric power. EPRI and Reasoning partnered to create a system called EPRI Software Quality Measurement System (ESQMS) to measure the conformance of C and Fortran code to a set of application specific coding standards covering diverse aspects of code including commenting style, formatting, identifier naming, language standard, coding idioms,

file organization, and use of libraries. The ESQMS was applied to over 3,000,000 lines of C and Fortran code written by contractors to validate conformance with the standards.

5.5 Formal Systems

Formal Systems was founded to provide re-engineering services for NATURAL applications. The company first created an automated translator from NATURAL 1.2 to NATURAL 2.0 using REFINE®, and then sold migration services to several commercial organizations.

Based on the success of these services, Formal Systems used REFINE® to create several other re-engineering tools services for NATURAL. These included the NXL2000 tool for Y2K inspection and remediation, and translations between other NATURAL versions.

5.6 Universities

Reasoning Systems introduced a discounted university license for REFINE® in the 1990's, to encourage use for research and provide a supply of new engineers trained in the technology. Among the universities that licensed REFINE® for research were Stanford, MIT, University of California at Berkeley, New York University, Carnegie Mellon, Rutgers University, Georgia Institute of Technology, University of Michigan, University of Linkoping, Queensland University

metaware.
the complete IT modernization solution

www.metaware.fr
Tel. +33 1 30 15 60 00
Fax. +33 1 30 15 06 71

Global Headquarter:
1. Parc des Grillons
60, route de Sartrouville
78230 Le Pecq – FRANCE