An Avenue for High Confidence Applications in the 21st Century

Timothy W. Kremann William B. Martin
Frank Seaton Taylor

{timk, bradm, frankt}@sundown.ncsc.mil

National Security Agency
9800 Savage Road Suite 6718
Fort George G. Meade, Maryland 20755-6718
410.854.6917
fax: 410.854.6939

Point of Contact: Timothy W. Kremann

Type: Paper

February 12, 1999

Abstract

Ensuring that an implementation has certain desired properties in the past has meant the incorporation of an overly expensive and time consuming process. The push to avoid this expense has resulted in the denigration of costly formal methods. However, formal tools are now beginning to emerge that make cost effective high confidence developments possible. In addition, techniques for concealing aspects of formal methods within an application designers/programmers development tools are allowing application engineers to acquire high confidence in their application's implementation without becoming proficient at formal methods, making formal methods eminently practical. Based on these developments and tangible results, the design and implementation of future high confidence applications is promising.

The vision which will be described in this paper is based on a unifying theory that allows all aspects of software engineering to be handled within a single formal framework. As motivation, recent successes of the application of this formal framework will be described. Furthermore, this paper will introduce readers to the vision and supporting technology by looking at three levels of knowledge capture and reuse: blocks, designs, and domains.

Moreover, it is our hope that formal methods will be a significant addition to the software engineer's toolbox and become common place for security and safety critical developments. Although the ideas outlined in this paper are not necessarily new, they are, however, embodied in a practical and evolutionary implementation. This paper, therefore, will describe a vision of how this technology could pave *An Avenue for High Confidence Applications in the 21st Century*.

1 Introduction

Ensuring that an implementation has certain desired properties in the past has meant the incorporation of an overly expensive and time consuming process. The push to avoid this expense has resulted in the denigration of costly formal methods. However, formal tools are now beginning to emerge that make cost effective high confidence developments possible. In addition, techniques for concealing aspects of formal methods within an application designers/programmers development tools are allowing application engineers to acquire high confidence in their application's implementation without becoming proficient at formal methods, making formal methods eminently practical. Based on these developments, and tangible results, the design and implementation of future high confidence applications is promising.

The vision which will be described in this paper is based on a unifying theory that allows all aspects of software engineering to be handled within a single formal framework. As motivation, recent successes of the application of this formal framework will be described. Furthermore, this paper will introduce readers to the vision and supporting technology by looking at three levels of knowledge capture and reuse: blocks, designs, and domains.

As a vehicle for understanding this vision, the SPECWARE[1] environment and its evolution will be traced. Formally, SPECWARE is a system for writing, composing and refining specifications. This is analogous to assembly language in computer science. However, as it is with assembly language so it is here, building applications in SPECWARE is tedious at best. In order to displace this tediousness, analogous to high-level languages in computer science, Designware was envisioned. Designware allows a designer to apply complex combinations of the basic building blocks which occur repeatedly in various designs in order to develop structures and solutions. These structured developments can then be collected and used at the next level of knowledge capture in an environment called Application-ware. At this level, Application-ware takes in a problem description and generates a solution using knowledge and constraints already captured in the system. This effectively hides the formal underpinnings. Thus SPECWARE and Designware provide a firm foundation for Application-ware front-ends, providing a user friendly front-end to the design and implementation of problem specific high confidence applications.

It is our hope that formal methods, especially those embodied in SPECWARE and its follow-on systems, will be a significant addition to the software engineer's toolbox and become common place for security and safety critical developments. The ideas in SPECWARE are not necessarily new, they are, however, being embodied in a practical vision. This paper, therefore, will describe a vision of how this technology could pave *An Avenue for High Confidence Applications in the 21st Century*.

2 Success Story

> The interplay between generality and individuality, deduction and construction, logic and imagination—this is the profound essence of live mathematics. Generally speaking, such a development will start from the concrete ground, then discard ballast by abstraction and rise to the lofty layers of thin air where navigation and observation are easy; after this flight comes the crucial test of landing and reaching specific goals in the newly surveyed low plains of individual reality. In brief, the flight into abstract generality must start from and return again to the concrete and specific.
>
> —Richard Courant

Before proceeding with the abstract theoretical underpinnings which support the vision and technology suggested in the introduction of this paper, it seems both motivational and instructional to begin by discussing a recent

[1] SPECWARE is a registered trademark of Kestrel Development Corporation.

success related to the application of the formal framework to be elaborated in the sections that follow. Specifically, Motorola's development of a Mathematically Analyzed Separation Kernel (MASK) provides a very clear view of the relevance of the SPECWARE technology in the design and implementation of high confidence applications. Let's begin this examination by answering the following questions before highlighting the MASK project:

2.1 What are separation kernels?

Operating system research has concentrated on organizing basic operating system functions into a collection called a kernel. The kernel presents abstractions of the fundamental resource management mechanisms to other, less primitive, service providers. In operating system implementations that attempt to provide a basis for secure information processing, the kernel software is carefully constructed and evaluated. To aid the evaluation process, the kernel functions are implemented as relatively small programs that are independent of one another.

Moreover, a separation kernel is charged with the critical task of providing separation among process spaces by manipulating the protection features of the system. In [6], John Rushby states that a separation kernel creates "an environment which is indistinguishable from that provided by a physically distributed system: it must appear as if each regime is a separate, isolated machine and information can only flow from one machine to another along known external communications lines". Figure 1 provides a pictorial view of separation in a physically distributed system. It highlights Rushby's suggested environment of separation given that the boxes and arrows are separate physical entities, providing a clear view that the only influence exerted on the processing inside one box is due to the wires that connect it to other boxes.

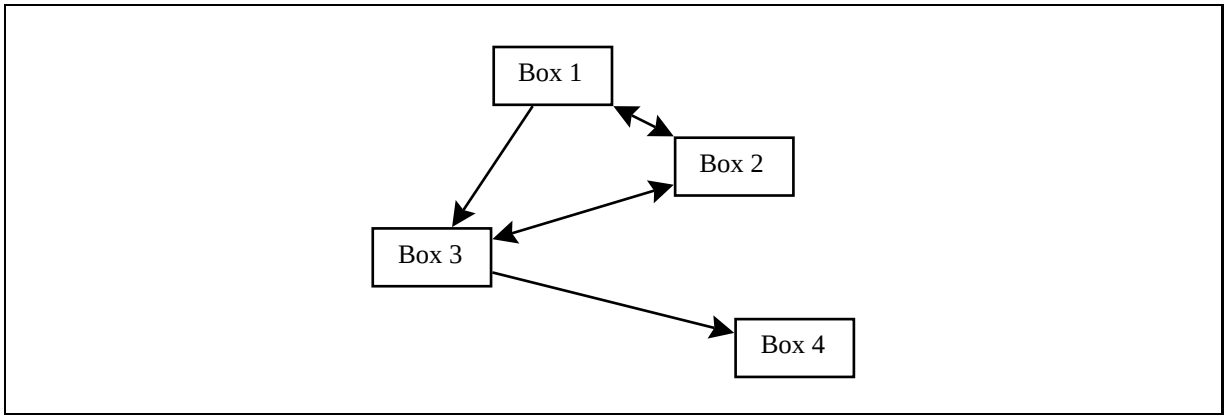

Figure 1: Physical Separation

This view becomes somewhat more clouded when the functionality of these boxes and arrows is implemented inside a single box placed in the context of an operating system and hardware. Figure 2 provides a view of this context and the possibilities of unintentional information flow due to the platform substrate. This type of information flow is often referred to as a covert channel. That is, a mechanism which is used to establish communication between two entities, where the mechanism was not intended to be used for communication.

2.2 Why are separation kernels desirable?

The primary benefit of a separation kernel in critical systems is derived from the kernel's ability to confine processes, data, and resources in different information domains. In this instance, the separation kernel serves as the

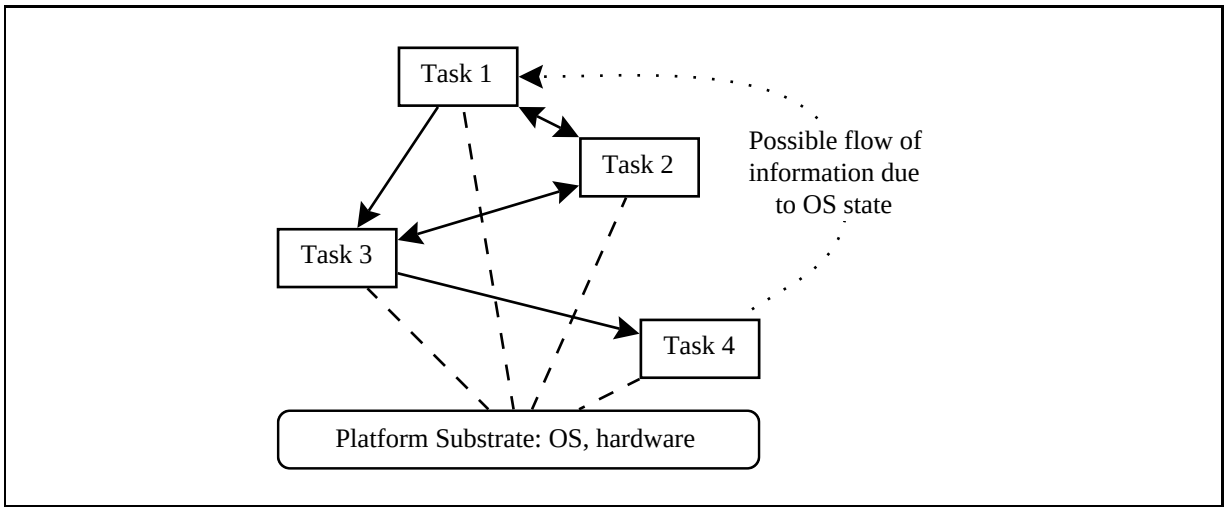

Figure 2: Single Box Implementation—Covert Channel[5]

ultimate security policy enforcement function by constraining all use of the basic information system resources. This role of the separation kernel is critical to the development of high assurance security products.

Another benefit to critical systems provided by a separation kernel architecture is the simplification of the security evaluation. As previously described, a separation kernel explicitly describes the communication between tasks which are allowed. With this in mind, the security evaluator need only examine the known communication lines between the tasks being evaluated, thereby limiting the costly evaluation of the entire system.

2.3 The Mathematically Analyzed Separation Kernel Project

The National Security Agency set out in 1997 to formally construct a kernel which would regulate communication between processes based on a separation policy, namely a separation kernel. It is to this end that the National Security Agency in conjunction with Motorola SSTG outlined the development of a separation kernel with the use of the formal methodology supported by the SPECWARE system.

The primary objective of the MASK project was to formally construct a separation kernel that could be used in the development of secure embedded systems. In an effort to formally construct such a separation kernel the following major tasks were outlined and completed: (1) the development of a security architecture; (2) the development of an abstract description of a separation kernel; and (3) formal implementation of this abstract description.

Upon completion of this initial prototype development by Motorola SSTG and NSA, Motorola extended this initial design, developing an extended separation kernel which is currently being employed in Motorola's Smart Card Technology. More recently, Motorola has made additional extensions to this separation kernel which has been introduced into Motorola's Advanced Infosec Machine (AIM).

Moreover, AIM is a high speed, high assurance crypto-module that can serve as a platform for many secure communications applications, including Type 1 encryption services. MASK, therefore, offers complete separation on board the AIM chip, providing confidence that an operation by one entity running under the control of AIM cannot influence another entity running under the control of AIM, unless communication between the two entities is explicitly allowed in the separation policy—*guaranteeing separation of all data providing for multi-level security on the AIM chip!*

3 Blocks

> Solving a problem is similar to building a house. We must collect the right material, but collecting the material is not enough; a heap of stones is not yet a house. To construct the house or the solution, we must put together the parts and organize them into a purposeful whole.
>
> —George Polya

SPECWARE, under development at Kestrel Institute in Palo Alto, California, is an environment for the specification and formal development of software with its primary objective being the correct development of entire systems. In the language of Polya, SPECWARE is the mechanism which enables builders to organize building blocks (specifications) into a purposeful whole. It achieves this goal by incorporating ideas from mathematical logic which help to integrate several notions in software engineering. More specifically SPECWARE provides for the formal composition of specifications and software components and the refinement of specifications to code.

Within SPECWARE a specification is described in logic and can describe problems, architectures, programs, data structures, and application domains. In addition SPECWARE provides a composition operator which enables a user to compose specifications from several smaller specifications, and to describe exactly how each of the smaller specifications relates. For example, a specification for arrays of integers can be constructed using the SPECWARE tool by composing component specifications for arrays and integers.

Next, software development is accomplished in SPECWARE by refining one specification into another. Refinement is a process in which all the elements of one specification are represented in terms of another. For example, if a specification for lists is refined into a specification for arrays, every operation on lists is described in terms of operations on arrays. In addition, refinement can be applied to the development of system architectures, the design of algorithms, and the selection of data structures. Each refinement has an explicit representation in SPECWARE, and refinements can be manipulated by the system. A complete refinement may also be regarded as a history of the system design process. The SPECWARE system, with its utilization of refinement, has capabilities for generating many desirable target implementations. Currently available target implementations are Lisp and C++.

Hence, the SPECWARE system which provides the aforementioned specification and refinement capabilities for generating specific implementations from high level specifications, along with its composition capability, provides the necessary mechanisms for constructing a *solution from a heap of specifications*.

In an effort to improve the reader's view of the basic building blocks of the SPECWARE system and its mechanisms for organizing them into a purposeful whole let's take a look at a recent effort in the area of Java security.

Beginning in October of 1997, the National Security Agency initiated a research effort with Kestrel Institute to explore the application of the SPECWARE technology in the development of reliable Java applications. This effort began by addressing the need for a high confidence Java bytecode verifier. Within the Java Virtual Machine (JVM), the bytecode verifier performs static and dynamic checks. The static checks insure that the class definition can be parsed to yield well-formed code. The dynamic checks verify type safety and insure that local variables and object instances are initialized before use for all possible execution paths of a method.

The approach which was followed in the development of this Java bytecode verifier was to formalize the verifier as a constraint satisfaction problem on lattices. Therefore, the main work of building SPECWARE specifications for the verifier centered around the development of the JVM lattice. The JVM lattice describes the information that the bytecode verifier maintains at each program point. In addition to formalizing the JVM lattice, transfer functions, class files, constraints, and the solution to a set of constraints were formalized.[3]

To provide for a better examination of the building blocks of SPECWARE and its apparatus for manipulating the same, lets take a closer look at the development of the JVM lattice. The JVM lattice used in the JVM bytecode

development was done in a very structured manner, building upon very rudimentary specifications (such as lists, stacks, sets, arrays, partial-orders, booleans, etc.) that reside in the SPECWARE library. Utilizing this library of specifications and SPECWARE's composition capabilities, more complex specifications were built, such as, primitive-semilattice, reference-semilattice, judgments, assumptions, and assertions.[4] Figure 3 depicts, essentially, the structure of the JVM lattice development which was carried out in SPECWARE. This effort illustrates the practical use of SPECWARE's basic building blocks and composition mechanisms in the development of high confidence applications.

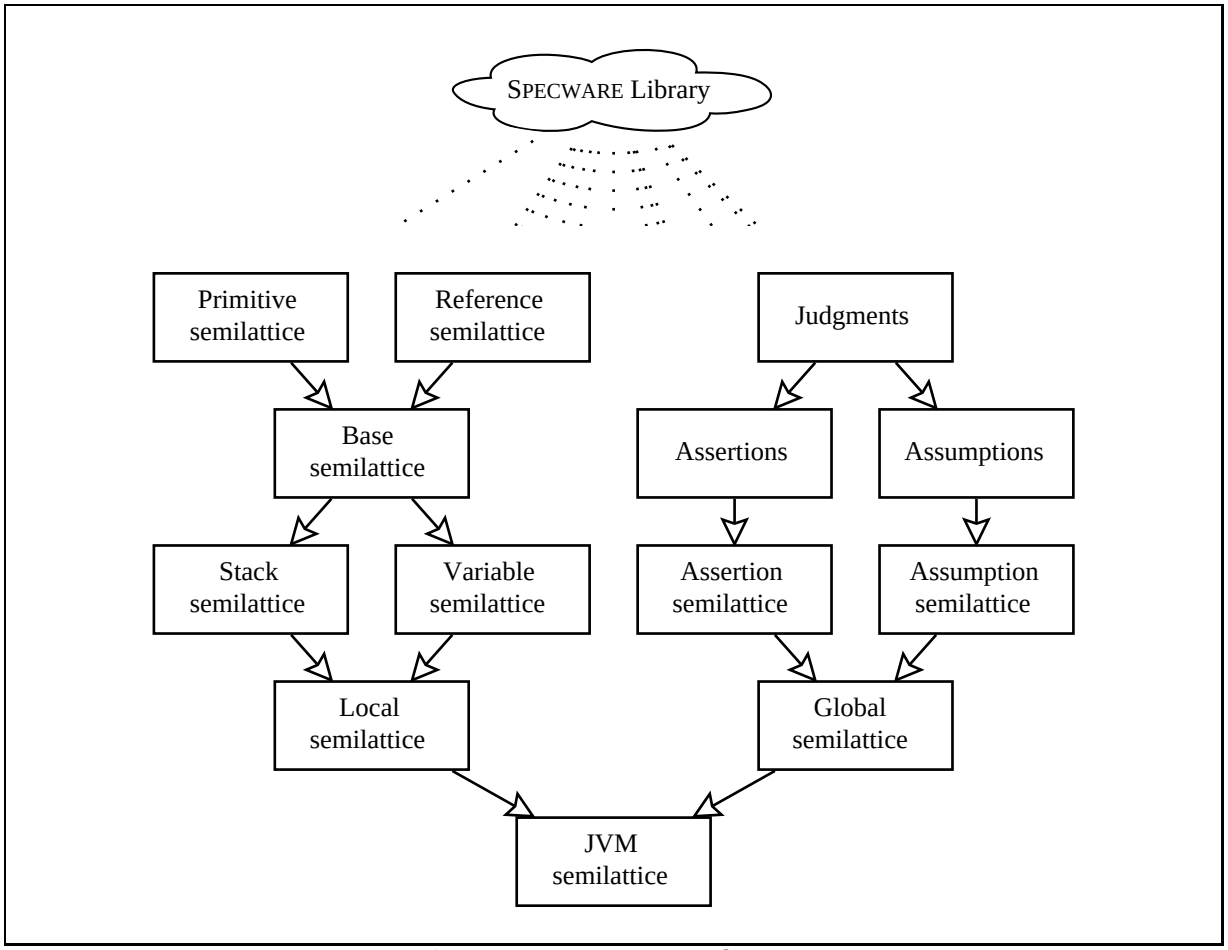

Figure 3: JVM Lattice Development

4 Designs

> To see what is general in what is particular and what is permanent in what is transitory is the aim of scientific thought.
>
> —Alfred North Whitehead

In the previous section, the process of refinement was presented as the process in which all elements of one specification are represented in terms of another. This view of refinement provides an expressive setting for

applying a library of abstract reusable refinements to a specification. Therefore, the goal of Designware is to provide the designer with a library of abstract reusable refinements and automated operations to complete his design. To date, refinements in Designware have centered upon design knowledge for algorithms, datatypes, and optimizations. It is the view of the designers at Kestrel Institute that other types of design knowledge can be captured in a similar fashion. In fact, the next section of this paper will outline a view of exploiting domain-specific design knowledge for a designers domain in general, and specifically for the planning and scheduling domain. For now, let's concentrate on design knowledge for algorithms which will reside within the Designware environment and explore how it will be applied toward the design of high confidence systems.

At the heart of the application of design knowledge is an idea developed in the early 1990's by Doug Smith of Kestrel Institute called the *classification approach*.[7] The classification approach solves the exact problem of constructing refinements of a requirements specification. The effect of this classification approach is to constrain the design by reducing the set of possible implementations. In an effort to be productive in the application of design knowledge, Designware must facilitate the development of organized libraries of abstract and reusable refinements, including a facility for their application by the designer. First, organization within a library of refinements is necessary, and is traditionally organized into a taxonomy. This taxonomy provides an ordering of how refinements increasingly constrain the design. In examining algorithm design knowledge, Kestrel Institute has developed an extensive taxonomy. This taxonomy is provided below in Figure 4.[8]

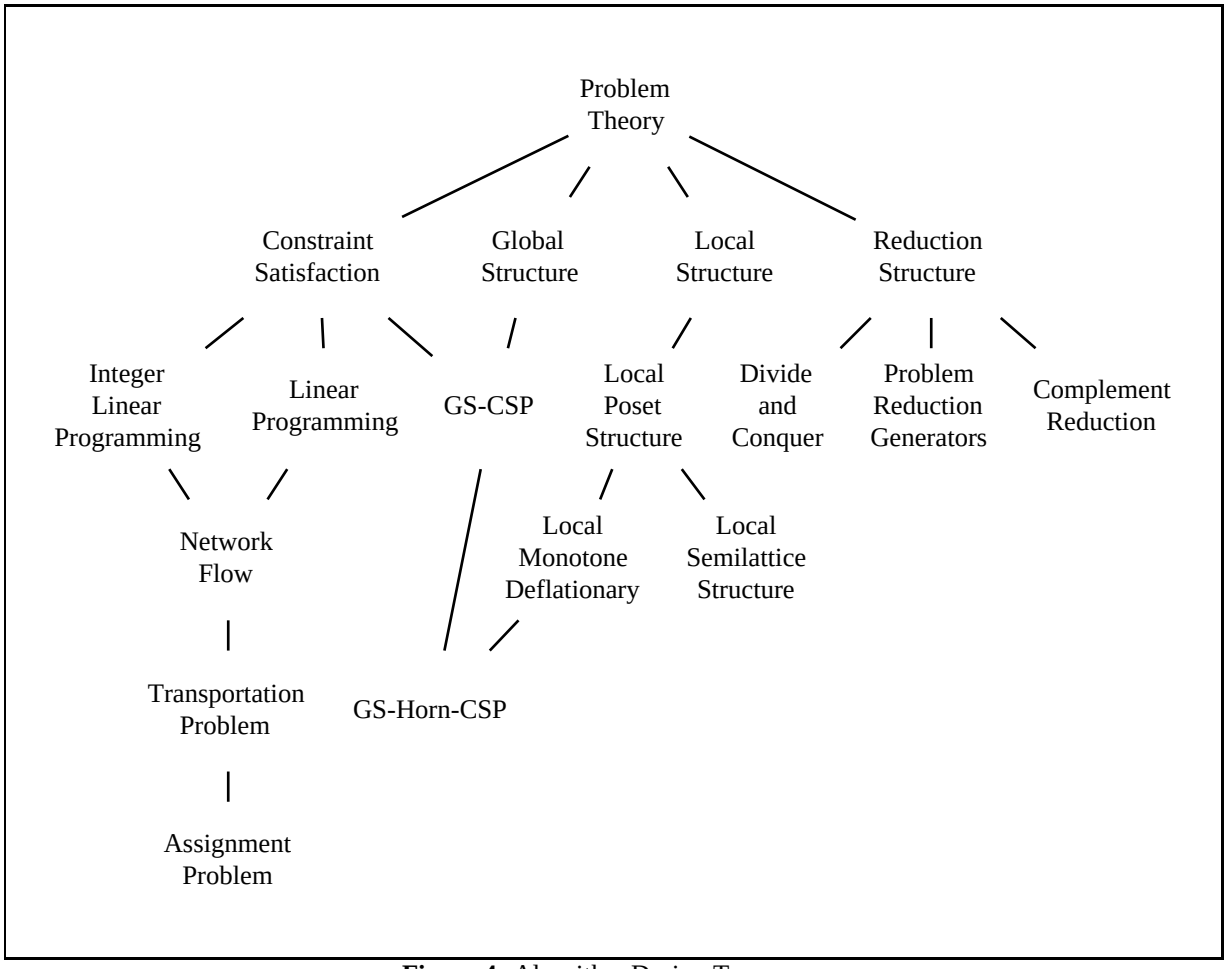

Figure 4: Algorithm Design Taxonomy

Now, with an extensive algorithm design taxonomy, and the foundational idea of classification, a process for incrementally constructing a refinement hierarchy can be achieved. Doug Smith coined this incremental refinement of the requirements specification as *ladder construction*.[7] Figure 5 depicts the ladder construction, providing for a view of generic ladder design, where DT's are the hierarchical design specifications, and the S's are the refined requirements specifications.

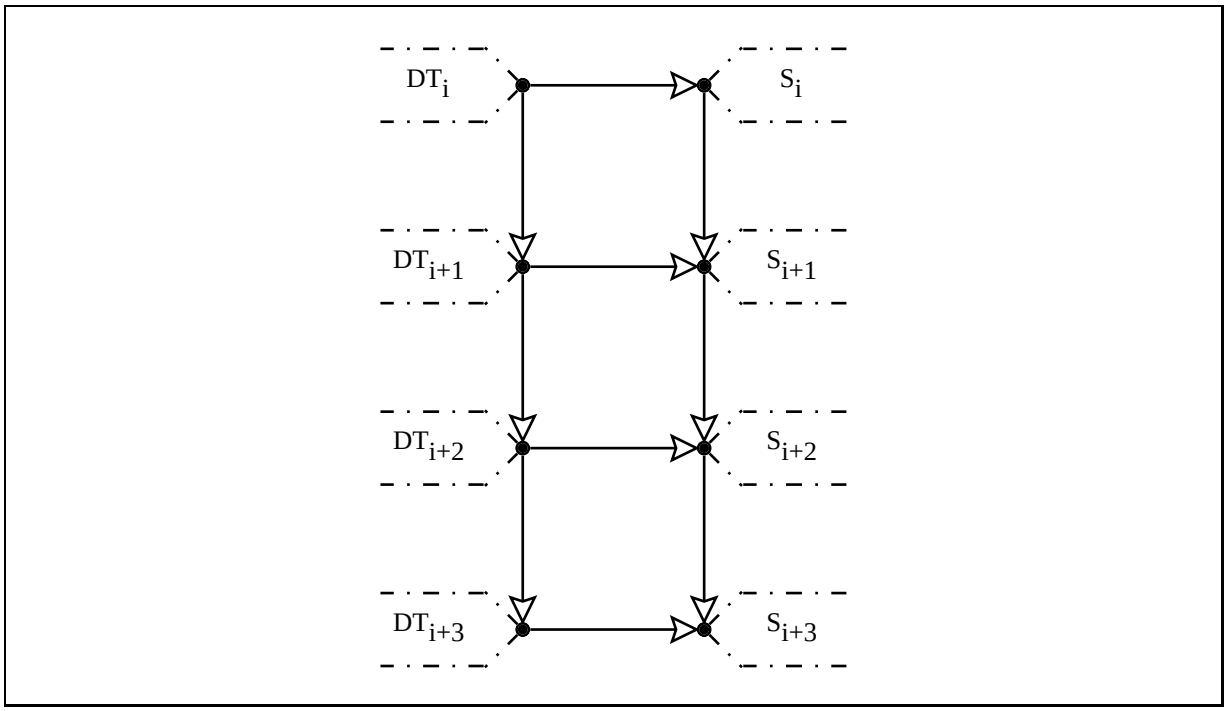

Figure 5: Ladder Construction

To solidify these thoughts, let us examine an exemplary problem where incremental construction of a refinement hierarchy is utilized in the development of an algorithm. The problem is that of placing k queens on a chessboard so that no queen can capture another, the well-known Queens problem. Moreover, this ladder construction development will provide a broader view of the application of design knowledge than merely algorithm knowledge. This example will incorporate the application of datatype and optimization knowledge, as well.

Observing the aforementioned ladder construction strategy, a depiction of the Queens specification refinement is presented in the graphical progression outlined in Figure 6. Figure 6(a) depicts the classification of the Queens problem in the form of a global-search solution. Global search is the process of repeatedly extracting, splitting, and eliminating sets of candidate solutions until no sets remain to be split. (Binary search and backtracking are well known examples of global search.) Additionally, the bottom rung of the ladder in Figure 6(a) shows the addition of constraints to the design theory. This additional information comes in the form of derived filters which have the effect of removing infeasible solutions from the search space.

Next, Figure 6(b) shows the conditioning of the constructed queens global-search algorithm. The conditioning operation, in effect, prepares the specification for an optimization design choice called finite differencing. In this case the act of conditioning amounts to the grouping of expressions. This section of the ladder transforms the queens global-search algorithm constructed in the last step of Figure 6(a) into a form presentable for finite differencing.

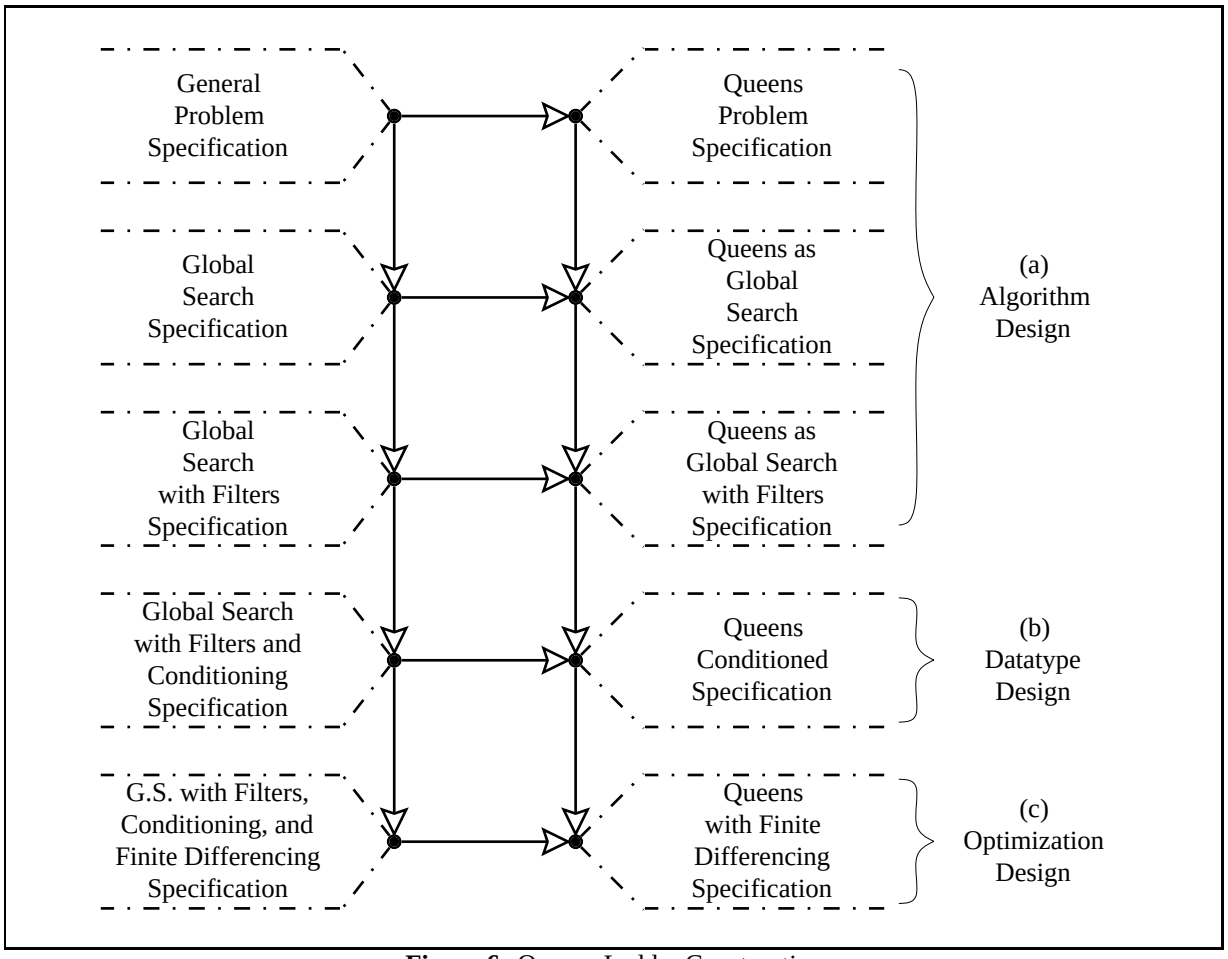

Figure 6: Queens Ladder Construction

Finally, in Figure 6(c) the refinement of the conditioned global-search algorithm for the Queens problem is completed with the application of finite differencing. Finite differencing is an optimization technique in which recurring expensive calculations are replaced by inexpensive incremental equivalents.

Although the example provided above is exemplary in nature it does have a more practical basis. The Queens problem is the focus of an ongoing high performance computing research initiative. The objective of this initiative is to demonstrate the use of the classification approach in the design of solutions for high performance computing benchmarks. This technology, which provides automated support for the formal derivation of programs, will explore the rapid and extensive exploration of the design space of alternative solutions and problem variants. This initiative will be driven by developing and applying this classification technology to selected problems from a set of high performance computing benchmarks and the Baskett Benchmark.[1] The problems which have been chosen will follow two classes of algorithms, backtracking and sorting. Tasks will include backtracking (Queens), integer sorting, and the Baskett Benchmark.

The intent of this effort is to demonstrate “local robustness” in the sense that the classification technology can support the generation of a variety of implementations for each of the benchmark problems and can handle variants of each benchmark problem.

5 Domains

> The whole of science is nothing more than a refinement of everyday thinking.
>
> —Albert Einstein

The essence of providing tools that *everyday thinkers* can use is to provide a tool which communicates with them in their language. Furthermore, every mature discipline of science and engineering has developed precise notations that facilitate the communication of essential concepts between experts. These notations, which include control diagrams and process flow charts, are concise, unambiguous, and may be effectively reviewed by experts. As such, they represent the most likely source of high quality, authoritative specifications in complex systems. Domain-specific languages (DSLs) are computer languages designed to capture and extend this powerful, essential communication pattern of experts.

Therefore, since the practitioner in a problem domain is an expert in his domain language a tool that understands the same language becomes extremely useful. In addition, the knowledge captured through domain analysis, using tools such as SPECWARE and Designware, which require both domain experts and knowledge engineers, is made available to the domain practitioner in his terms, thereby hiding the complex formal methods underpinning of the tools.

The fact that formal methods are hidden is essential in making these tools useful in practice. Many times a formal description of a problem in an esoteric language has been useful for understanding the problem but provided little to the actual developers. Many practitioners refuse to look at formula that contain a backward "E" ($\exists$) or an upside down "A" ($\forall$). Even though the formalization may have cleared up some ambiguities in the requirements, many insights that the formal methods specifier obtained are not in a form that the practitioner can use. This can be changed by expressing the results in the terms of the practitioner.

Additionally, domain knowledge, once captured, enables the building of solutions to problems in domains using familiar terms. The knowledge of how to build solutions from problem descriptions is developed during the analysis of that domain, accomplished by complex and cumbersome formal analysis. However, this analysis, once done can be reused across problems within that domain. Hence this complex work can be put into a reusable and automated form, reducing the need for the practitioner to understand this analysis.

Automation works because the difficult domain analysis provides the basis for an application generator, and is not redone by the practitioner. The input to the domain analysis includes the expertise of domain practitioners and the generic knowledge of computer science and systems engineering. The output of the domain analysis is an application generator. The application generator is then available to the practitioner who inputs aspects of his particular problem in terms of the domain and the output is, then, an application that will generate solutions to his problem. Expressed another way, the application generator is the embodiment of the knowledge of the problem domain and supporting knowledge and is reused across applications. The application output of the generator is an application that produces a solution to a specific problem within the domain. With this narrative in mind, a working example developed by Kestrel Institute, called Planware[2], will be discussed to further illustrate this approach.

The problem domain for Planware is planning and scheduling. The domain language is in terms of resources, tasks, reservations and their attributes. In this instance, users simply choose the type of resources from a taxonomy of resource types. Users are then presented with a spreadsheet where they can select constraints on the tasks and resources. Each of the fields is restricted to appropriate values. Without the user realizing it, Planware constructs a formal specification from the spreadsheet. This domain-specific specification of the users problem is then used by Planware as the start of an ladder construction as seen in the previous section, all without further user input. The output is a domain specific scheduler that when given a set of scheduling requests will output a valid schedule.

This may seem mundane, however the domain analysis embodied allows Planware to generate schedulers that produce solutions where previous schedulers have failed, and more impressively, generates schedulers that are orders of magnitude faster. The spreadsheet in Figure 7[2] provides a view of the typical constraint choices to be made by the user after having selected 'transportation resource' from the taxonomy.

Parameter	Lower Bound	Exact Value	Upper Bound
Start Time	Task.release	Finish - Duration	Task.pick-up
Resource-type		Multi-choice menu	Sum of task req'd resources
Instantaneous demand	min-cap	Sum of task demands	max-cap
Duration	0	Finish - Start, Dist(orig, dest)/speed	
Finish Time	Task.ead	Start + Duration	Task.due-date
Max-capacity		r.r-type.max-cap	
Separation	0	r.r-type.separation	
Origin		Task.poe	
Speed		r.type.speed	

Figure 7: Transportation Resource Spreadsheet for Constraining Reservations

Planware will use this input and its domain knowledge to produce a scheduler. All of the knowledge about schedulers is automatically used in generating that schedule, a great example of reuse.

The significance of this development environment is that a complex formal process and a treasury of domain knowledge was used to produce Planware. This domain specific knowledge is reusable across most scheduler problem domains. Moreover, the beauty of Planware is that the practitioner did not have to learn formal methods to use this tool.

Planware is not the only example of domain specific languages that have achieved success within academe and industry. However, it is the view of the writers that the ability to formalize problem domains using the same underlying methodology of knowledge capture and reuse that is presented in the SPECWARE and Designware technology, will allow for easier reuse of generic knowledge across domains.

Finally, efforts are currently underway to develop a taxonomy of abstract and reusable specifications for the design and implementation of cryptographic service providers (CSP). The hope is to formalize the CSP domain through formal knowledge capture, and then to provide an expressive domain specific interface to the CSP implementor.

[2]This spreadsheet is courtesy of Kestrel Institute.

6 Conclusions

It is clear that there are certain applications where failure is simply too costly and investment in formalization at the outset is mandated. Indeed, the technologies presented here create the greatest benefit if they are used at the very earliest stages of design. The successes herein and their requisite rigorous analyses of problem domains are indicative of the track we should follow.

Moreover, it is reasonable to expect that the cost and intellect required to use these tools will decrease dramatically, while the size and complexity of future systems will steadily increase. The combination of these forces will eventually lead to outgrowths of the ideas in this paper being integrated into the development process, making designers unaware that their tools rely upon advanced mathematical underpinnings. Furthermore, designers will come to depend heavily on tools descended from the ones mentioned herein.

References

[1] Michael Beeler, *Beyond the Baskett benchmark*, Computer Architecture News (1984).

[2] Lee Blaine, Limei Gilham, Junbo Liu, Douglas R. Smith, and Stephen Westfold, *Planware—domain-specific synthesis of high-performance schedulers*, Proceedings of the Thirteenth Automated Sortware Engineering Conference (Los Alamitos, CA), IEEE Computer Society Press, October 1998, pp. 270–280.

[3] A. Coglio, A. Goldberg, and Z. Qian, *Toward a provably-correct implementation of the JVM bytecode verifier*, Tech. Report KES.U.98.5, Kestrel Institute, August 1998.

[4] Kestrel Institute, SPECWARE *specification of the JVM bytecode verifier*, August 1998, Contract MDA904-98-C-B448.

[5] Motorola Space and Systems Technology Group, *Mask top level specifications*, November 1996, Contract MDA904-96-C-089.

[6] J. M. Rushby, *Design and verification of secure systems*, Proceedings of the Eighth Symposium on Operating Systems Principles, vol. 15, December 1981.

[7] D. R. Smith, *Classification approach to design*, Tech. Report KES.U.93.4, Kestrel Institute, Palo Alto, California, 1993.

[8] ———, *Mechanizing the development of software*, Calculational System Design: Proceedings of the International Summer School Marktoberdorf (Amsterdam) (Ed. M. Broy, ed.), ASI, NATO, IOS Press, 1999.