Mechanical Tool Support
for
High Integrity Software Development

Michael Mehlich Ira D. Baxter
Semantic Designs, Inc.
12636 Research Blvd. C-214
Austin, TX 78759-2200
{mmehlich,idbaxter}@semdesigns.com

Abstract

Since the failure of a high integrity software system has a huge cost, it is important to develop reliable, trustworthy software for such systems that is very unlikely to fail due to coding errors. This high level of software reliability has to be preserved for the life-time of the system, even as its requirements and implementation change. One method for achieving this level of reliability is to formalize the software development process, and provide mechanical tools to support it, to ensure that no process errors are induced in the final code developed from a formal specification. Mechanical modification of the steps for constructing the system then can ensure that this high degree of reliability is preserved. We sketch a tool, the Design Maintenance System (DMS), which we are constructing. DMS provides the type of support required to perform these activities mechanically with interactive support, and discuss how this tool can help constructing and maintaining reliable, trustworthy software for high integrity systems.

1. Introduction

Software-based systems can be found in more and more application areas, including high-risk areas where a system failure induces huge costs, e.g. serious injury, loss of life, or destruction of resources. For such high-risk application areas it is important to construct and maintain trustworthy software with an extremely small probability of an error occurring that would cause a serious system failure.

In traditional, i.e. current, software engineering practice *informal* requirements are somehow converted into an *informal* and *semi-formal* specification using generic domain notations without underlying precise semantics from which the program then is constructed manually. The final program is validated by performing tests using manually derived test cases. In practice, the delivered software will have to be changed due to changed context, requirements or errors actually found in the program. Maintenance is then done by modifying the code manually, often without modifying the specification as well.

This approach to software construction has several obvious drawbacks:

- The *gap between the informal requirements and the specification* in generic domain notations is too large. Thus, it is difficult, if not impossible, to validate the correctness and completeness of the specification.
- The specification *does not have a precise semantics*. Thus, it is ambiguous, i.e. subject to different interpretations by different engineers that are involved in the construction of the software.
- There is *no precise relationship between the specification and the final program*. Information about how the specification is realized in the program or why it is realized that way cannot be found or, at best, is documented informally. This makes it impossible to trace the impact of the requirements to

the construction process and the final program, or to trace an error found in the program back to the requirements or to a design decision.

- *Testing the final program* with manually or even automatically constructed test cases *is neither exhaustive, nor can it be used to prove the program is error free*. Testing can only show the presence of errors.
- *Maintaining the software* by *modifying the code* is difficult and often *introduces new errors* into the program. Failure to reflect changed requirements in the specification also leads to an increasing gap between the original specification and what the code really does.

The difficulties with the conventional software method make the reliability of its products highly suspect, especially if the software in question has a long lifetime and is subject to the usual pressure for many changes over time. Thus, conventional methods seem like a poor approach for constructing and maintaining reliable software systems for high-risk areas.

The fundamental basis for overcoming the deficiencies of the conventional method is to have a specification of the requirements using domain notations that are specific for the problem area, thus reducing the gap between the informal requirements and the specification. Giving the domain notations a precise underlying semantics ensures that there is no ambiguity in interpreting the, now *formal*, specification. Analysis and simulation tools based on such formal domain semantics then can support the validation of the specification using terms understandable to the specifiers.

Such a formal specification of the requirements then would theoretically allow proof of the correctness of the program with respect to the specification. However, in general the specification as well as the final program code are large and there are many hidden design decisions made during the manual construction of the program. Such hidden design decisions have to be recovered from the code and encoded in the proof. This is as hard, if not harder, than designing the program in the first place. Program scale thus makes it essentially impractical to carry out these proofs in an economical fashion. To alleviate the recovery problem, we have to reduce the gap between the formal specification and the program. This can be done by introducing intermediate specifications that contain smaller, easy to detect design decisions. Ideally, these design decisions are so small that their respective proofs are trivial.

A transformational approach to software construction essentially provides us with such small design decisions in form of transformations (cf. [Nei84, Par90]). If the transformations have been proven to be (functionality) correctness-preserving in advance, then the program, constructed by applying the transformations successively, then is *correct by construction* with respect to functionality. The transformations then represent decisions on how to implement function. Satisfaction of the non-functional part of a full specification (see [Bax90]) is a consequence of “performance side-effects” of the chosen transformations. To achieve desired performance effects, one must have and use explicit performance criteria to select transformations (cf. [McC87]). The selection of (a set of) transformations to achieve a performance effect represents a decision on how to achieve performance. The applied transformations together with their rationale for application exactly describe the relationship between the specification and the program.

The transformations and their rationale can be captured in a *transformational design* (see [Bax90]). Such a design record allows tracing requirements from the specification to the code, and tracing back from code fragments to the specification and/or the transformations (and their rationale) that are responsible for deriving that piece of code. The transformational design can also be used to guide re-implementation of the software when the requirements change, by reusing recorded design decisions that are still valid rather than develop the new implementation from scratch.

Capture of a transformational design enables the whole software construction and maintenance process from the specification to the program to be supported by a semi-automatic system. Construction and modification can be performed incrementally to adapt the software system to changed requirements (cf. [Bax90]). This *avoids process errors.*

Effective use of such a semi-automatic system requires considerable infrastructure in the form of predefined domains, each capturing knowledge about some problem domain or some implementation technology domain (database, graphical user interface, file systems, communications, etc.). Each domain consists of notations, concepts, and transformations relevant to that domain as well as maps to lower-level domains. To acquire this domain knowledge is expensive.

Unfortunately, even for high-risk applications there is only a limited budget available for constructing the (software) system. Therefore, as there is a strong relationship between the reliability of a system and its cost, a high cost of knowledge acquisition means a low reliability of the system developed within this budget limit (cf. Figure 1). It is unfortunate that reducing knowledge acquisition also leads to either poorly performing or unreliable systems.

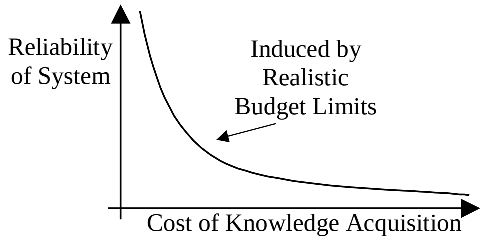

Figure 1. Reliability vs. KA cost

Successful risk reduction consequently requires reduction in the cost of domain knowledge acquisition used by tools. This can be achieved by amortizing the cost of knowledge acquisition over many different applications systems and by having a highly agile tool for supporting the development of domains.

In the sequel we give an overview of DMS, a mechanical tool that supports the incremental construction and maintenance of large application systems as well as domains. We then discuss how such a tool could help developing trustworthy software for high integrity systems in high-risk areas.

2. The Design Maintenance System (DMS) — An Overview

DMS (cf. [Bax95]) is fundamentally a transformational synthesis system, i.e. a semi-interactive system for deriving code from specifications by repeatedly applying transformations. The transformation knowledge is organized around knowledge clusters called "domains". DMS captures a transformational design, and aids in the revision of an application (see Figure 2) using the captured design as a guide (see [Bax90], [Bax92]).

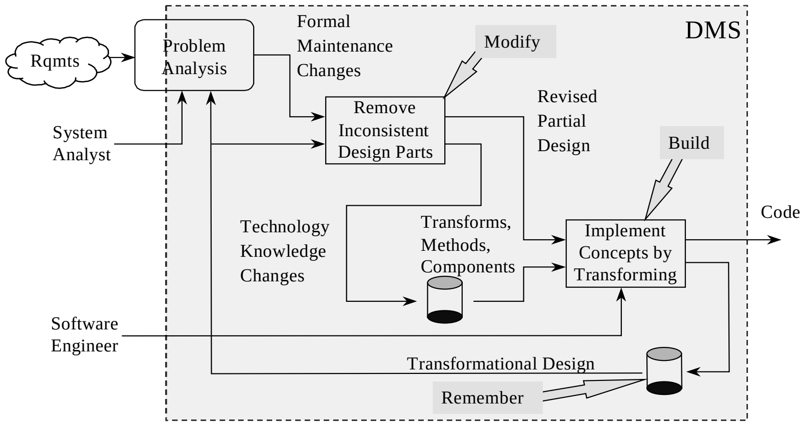
DMS
Rqmts
Problem Analysis
Formal Maintenance Changes
Modify
Remove Inconsistent Design Parts
Revised Partial Design
System Analyst
Build
Code
Technology Knowledge Changes
Transforms, Methods, Components
Implement Concepts by Transforming
Software Engineer
Transformational Design
Remember

Figure 2. The Design Maintenance System — Concept

Using this system, software could be constructed by first developing a "functional" specification of the software together with performance specifications that describe the non-functional requirements (e.g. time complexity of algorithms to achieve and/or the implementation language to use). The program code then is derived by successively applying transformations that are semi-automatically selected from a repertoire of domains to approach and finally realize the performance specifications while satisfying the functional specification. The performance specifications serve as the justification for the implementation decisions made, i.e. the transformations applied.

A major practical problem is that during the life-time of a software system its requirements are constantly changing (see [Boe81, Gui83]), often even while the software is still under construction. DMS is intended specifically to address this issue. Repeatedly performing an in general huge derivation is expensive both computationally and in terms of interactive steps (which no one wants to revisit, especially if their outcome does not change). It is desirable to reuse as much of an existing derivation as possible.

Thus, DMS records the transformations applied, their point of application, and the rationale for applying them (i.e. the performance requirements that are approached). Additionally, DMS builds on a theory of transformational maintenance to revise the captured transformational design according to formal changes made in the functional specification as well as in the performance specifications. This allows DMS to reuse and revise major parts of a derivation for the changed system specification often without a need to revisit all the decisions. In particular, this approach avoids the need to ask the software engineer to revisit unaffected interactive decisions. A sample of this process applied to a data processing program can be found in [BP97].

2.1 Domains

A significant number of different notations, transformations, methods, and software components are needed to construct and maintain large systems. The SINAPSE system (see [Kan91]) for the generating mathematical modeling codes had several thousand chunks of synthesis knowledge, used for a narrow domain and producing relatively small (5K source lines of code) programs. It is not practical to manage such complex information as an amorphous collection. In DMS this knowledge is organized into a network of domains (cf. [Nei84]).

Each domain consists of at least the following parts:

- A *specification (or program) syntax* that allows DMS to read (i.e. transform the syntax into an internal form), process (the internal form), and display (i.e. transform the internal form back to the syntax) documents written in the domain specific notation.
- *Parameters*, which are uncommitted details of a domain having properties dictated by the domain, in the same spirit as parameters in algebraic specifications. (For a parallelism domain, the parameters may represent arbitrary side-effecting actions).
- An (optional) *machine-interpretable semantics* describing the meaning of documents written in that notation. Such a semantic description provides the basis for domain independent analysis capabilities such as symbolic simulation and deriving specification properties. Semantics are also a necessary precondition for being able to support mechanical proof the correctness of the transformations within the domain as well as between domains although this is not required to use DMS.
- Domain specific *transformations* that can be classified as
    - *Optimizations* that can be used to simplify or elaborate a specification within the domain,
    - *Refinements* that describe how to transform concepts of the domain to concepts of other domains at a lower level of abstraction, and
    - *Jittering transformations* that modify the specification in order to make transformations of the other two classes applicable.
- *Methods* that comprise rules to apply certain (sets of) transformations. Such methods tell DMS how to select groups of transformations to synthesize codes by providing a semi-procedural plan to approach, achieve, or preserve performance properties. Often methods are non-deterministic plans that use refining transformations to map a configuration of concepts (i.e. a set of related concepts together with their relations) in one domain into another configuration in a domain at a lower level of abstraction.
- Reusable, explicitly defined *software components*, each component consisting of a set of methods that can be applied to the same (or a similar) abstract concept. Such components are more reusable than conventional code-based components as they tailor themselves to the context in which they are used. This is especially useful for component compositions which occur naturally when an abstraction is refined into other lower level abstractions, each of which has its own components that may be applied to obtain even lower level abstractions. The explicit definition of such components allows domain engineers to compile their knowledge into the domain definition, which in turn allows application engineers, who usually have less knowledge about the domain, to take advantage of this knowledge.
- *Analyzers* that measure "interesting" properties of a specification (or program). Such analyzers could e.g. compute performance values of the specification or, even more important, information that is needed in preconditions of transformations.
- *Procedures* that provide efficient implementations for complex methods or software components (which may but need not be included explicitly as methods and software components, respectively) or provide additional functionality (e.g. test case generation).

A specification written in one domain can be repeatedly refined into specification(s) in other domains to achieve an implementation, under the guidance of domain methods and interaction with the software engineer. (We call this forward engineering.)

2.2 Domain Networks

The domains are related by virtue of specifications from abstract domains being refinable to specifications (or programs) in domains of lower level of abstractions. This implicitly establishes a *domain interconnection network* (an example of which can be found in Figure 3) with specific application domains at the most abstract level, generic application domains, computer science domains, and execution model domains at intermediate levels of decreasing abstraction, and target execution languages at the lowest level(s) of abstraction. DMS provides a collection of reusable domains for the lower levels of abstraction (gray area) as the cost of acquiring these can be amortized across many applications. This allows domain engineers to concentrate on defining the value-adding generic and specific application domains that are defined and implemented in terms of other more implementation-oriented domains.

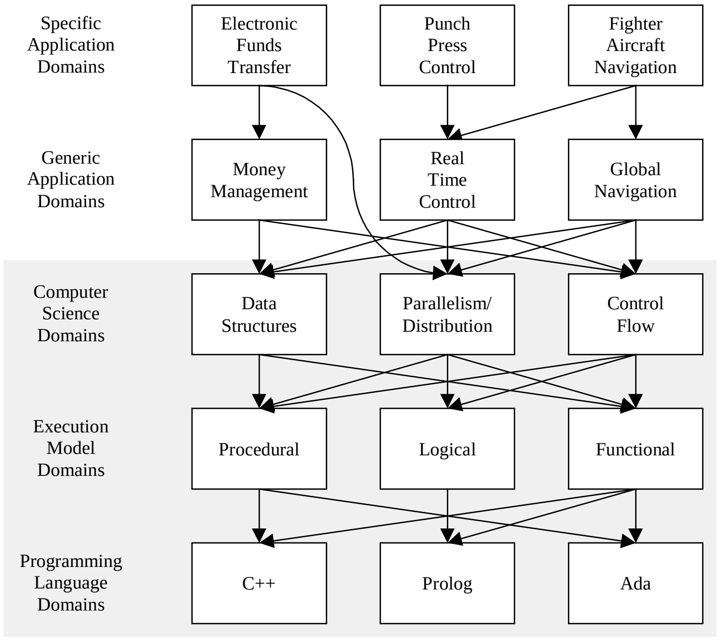

Figure 3. A Domain Interconnection Network

This domain network provides all the knowledge needed for constructing software mechanically. It provides application domain specific notations to describe the functional and performance specifications, software components, methods, and transformations to derive lower level (functional and performance) descriptions from higher-level descriptions, and analyzers to validate and simulate descriptions on different levels of abstractions.

The domain knowledge in the network can be heavily (re)used in the construction process of many different software systems, and can consequently be well tested. Most of the errors that originally might have been compiled in the domain knowledge are eliminated over time. This contributes to higher reliability of newly constructed systems (re)using these domains, as well as to systems already constructed with the possibly erroneous domain knowledge via maintaining them.

2.3 Design Capture, Reuse and Incremental Modification

As already mentioned briefly DMS records the automatically chosen as well as the interactively enforced steps, i.e. software components, methods, and transformations that have been applied successfully during the software construction process together with the rationale that led to applying them. DMS also records how an overall possibly complex performance specification is broken into smaller and simpler subspecifications over in general smaller regions of a functional specification. This recorded information forms the *design history* of the software system. We consider this design history as *the product* of the software construction process whereas the resulting code is “only” a byproduct. The design history provides a complete explanation of the code byproduct.

Capturing the design history is important for explanation, traceability and to save effort during later software maintenance. Both the software engineer and DMS work hard to discover which software components, methods, and transformations to apply, to determine exactly where to apply them, and to achieve the performance requirements. Redeveloping even only a small fraction of an application is expensive.

Many, especially small, modifications to the software specification (*and*, presumably less-frequently, to the domain definitions) that have been used during the construction process usually have only a small impact on the code. It is therefore worthwhile to incrementally modify, i.e. revise, the design rather than to reconstruct the code from scratch again. DMS provides this capability.

The DMS revision mechanism for the design history can take advantage of the ability of transformations to commute application order in the design history (cf. [Bax90]), and the flexibility of methods and especially software components to adapt to changes in the concepts and/or their performance requirements. In essence DMS tries to preserve as much of the design history (including the interactively chosen steps) as possible and reorders, replaces, or removes transformation steps that are no longer applicable.

The result of the revision process is a partial design history for the modified software specification. By switching over to a transformational software construction process this design history can be completed to produce a new implementation. DMS may be able to use the steps removed from the design history to guide the construction process by proposing analogous transformations. In the course of repairing a partial history, DMS uses the same tested domains it uses for pure forward engineering.

The newly completed design history has the desired modified code as an outcome, and can be revised again by the same process for future modifications.

2.4 Reverse Engineering to Maintain Legacy Systems

Ideally, all software would be constructed and maintained within a system like DMS. However, there exist many legacy systems for which one often has only the system code with some informal, inaccurate documentation. Legacy systems are by definition successful, and since successful systems suffer a continual demand for enhancement; they have to be maintained. Thus, DMS makes a concession to reality and provides support for reverse engineering legacy systems recover the lost design, or at least a plausible design.

To obtain such a possible design history DMS interactively runs its engine backwards by applying the transformations, methods, and software components in reverse direction (see [BM97]). DMS tries to recognize more concrete realizations of concepts and to abstract them to their respective concepts on a more abstract level of description. The software engineer participates as guide, arbiter among proposed abstractions, and on-line domain engineer to provide missing domain abstractions and/or implementation knowledge. The recognized steps are recorded as leading from the abstract concept to the concrete realization. They can be revised to maintain the system specification obtained by the reverse engineering effort.

It is not necessary to recover the entire design to maintain an existing system. One only needs to recover those parts that will be affected by the changes to be made. It is further sufficient to raise the level of abstraction only as far as needed for the software engineer to perform the changes reliably.

2.5 Scale

Many software systems consist of hundreds of thousands of lines, some even of millions of lines of code. For DMS to be useful, it has to scale correspondingly in several aspects:

- the size of the application system
- the number of engineers constructing, respectively maintaining the system
- domain knowledge acquisition

DMS faces these scale issues by several means:

- Incrementality in the modification process.
- DMS is implemented in PARLANSE, a parallel processing language. PARLANSE provides efficient support for forking and synchronization of small-grain parallelism as well as sophisticated software engineering support such as modules, abstract types, and robust exception handling. This provides computational power to support the expensive symbolic processing involved.
- Design histories are not treated as linear histories of transformations steps between specifications but rather as non-linear networks of dependencies between effects of transformations. Thus, major fractions of the design history need not be revisited for small changes of the software specification.
- DMS is a client-server system allowing the software engineers to work on their workstations whereas the design history database is held on a server. An individual engineer implicitly locks those parts of the design history he may be changing. For large systems these are small portions allowing multiple engineers to work with small interference.

- Large application systems are coded using many different languages. DMS supports this by its agility in handling domains.
- Domains are themselves definable and testable within DMS. This aids knowledge acquisition and validation.

3. DMS–Support for High Integrity Software Development

In previous sections we provided a short overview of a sophisticated mechanical tool for the construction and maintenance of software. Such a tool can support the development of trustworthy software for high integrity systems by several means:

- *Specification in domain terms*, enabling access by non-technical domain experts, reducing encoding errors and easing specification inspection.
- *Use of domain semantics and analyzers*, enabling specifiers to validate behavior or properties at various stages of implementation, and enabling off-line verification of critical components.
- *Modularization and (re)use of layers of domain (implementation) knowledge*, simplifying development and amortizing the knowledge acquisition cost required to capture or develop reliable components.
- *Traceability from specifications to code*, enabling fault analysis and reliable software modification.

We discuss each of these topics in more detail.

3.1 Specification in Domain Terms Reduces Modeling Errors

DMS allows the use of arbitrary domains with their specific notations for specifying a software system. Choosing a *domain notation* that is close to the actual need, i.e. *close to the "natural" model that has to be specified*, reduces the number of errors that are made during analyzing and describing the requirements and increases the chance to find possible specification errors early. Consider e.g. the (nearly) self-explanatory specification of a production cell (with a three stage conveyor belt) given in Figure 4 (ignore the gray patch for now). It should be easy to imagine the actual physical production cell. This specification is intended to describe (a system of) controller(s) that ensures that the production cell eventually and continuously produces the desired stream of output parts provided it gets an acceptable stream of input. The specification is considered consistent, if and only if there exists such a controller.

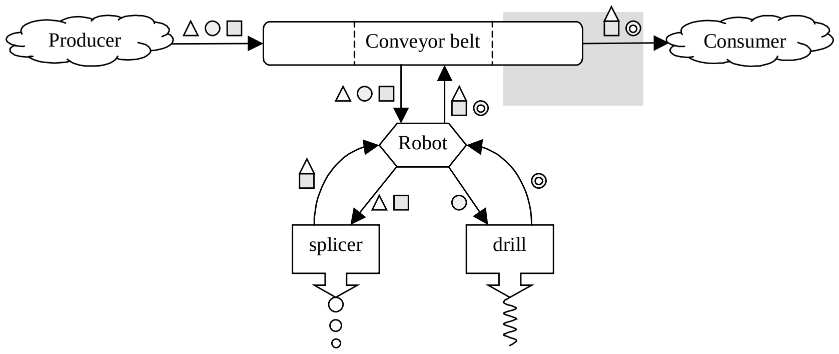

Figure 4. A Simple Production Cell

3.2 Domain-level Semantics/Analysis Enables Early Problem Detection

For such abstract domains it is possible to associate a *machine-interpretable semantics* for its notation. DMS supports describing the domain semantics in several different ways to accommodate the needs for the specific domain:

- Transformational semantics ("equivalence" transformation into another "lower level" domain having a machine-interpretable semantics)
- Denotational or algebraic semantics,
- Operational semantics

Having a domain specific simulator, or a machine-interpretable semantics for the domain with a generic symbolic simulator, enables *simulation* of the specification. Using such a simulator, a software engineer and/or domain specialist can observe and validate the specification's behavior by symbolically "running" the specification. In our example the domain semantics could be described by using denotational semantics. A simulation then likely would show that the production cell may not be able to continue its work, e.g. if the producer provides three squares one after another.

Providing other domain specific analyzers, or applying a generic analyzer again using a machine-interpretable semantics for the domain enables the *analysis* of certain properties of a specification with the goal of validating it. As an example, a liveness check could determine whether each component of the production cell eventually and continuously gets input parts necessary to perform its work and whether it continuously produces an output part. This again is only possible under certain conditions the producer has to satisfy.

Both *simulation and analysis help validating the specification*, and thus aid the detection of potential errors early in the software development process, which reduces the overall cost for the software construction.

3.3 Domain Layers Simplify Implementation and Increase Robustness

DMS refines specifications through intermediate domains ultimately into implementation domains. The domain interconnection network *modularizes the (DMS transformational) development problem*. The smaller differences between different levels of descriptions for the system makes the whole *process* easier to mechanize and *easier to perform*.

Each of these domains come with a possible semantics and/or simulator procedures/analyzers. This enables analysis of partially-implemented specifications. Thus, potential errors in the specification and/or development steps chosen may detected earlier rather than later.

A suitable intermediate domain for our example of a production cell would be a domain of colored Petri nets (cf. [HD95]) which is parameterized by the colors for the states and transition conditions. With such a domain and suitable domains for the colors the three stage conveyor belt could be realized by the Petri net as depicted in Figure 5 (where p, q, and r denote the state of the respective conveyor stage). The controller triggers the belt to move forward only if the first two sections of the belt are ready, and certain conditions are satisfied that ensure the overall system continues in a safe and live state, i.e. that no part falls off the conveyor belt and that the whole system can continue permanently (provided the conveyor belt is fed appropriately by the producer).

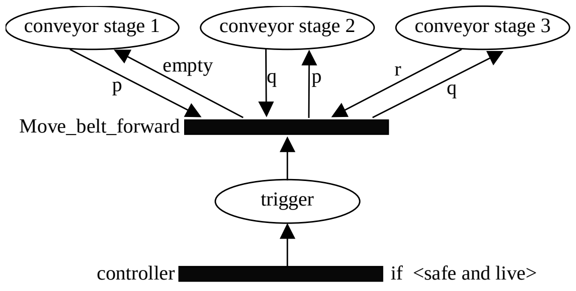

Figure 5. A Petri Net for the Conveyor Belt

The final implementation of the controller may have to be written in C. For the conveyor belt this code may look as depicted in Figure 6, in which the Petri net has been implemented as a polling device, and the safety and liveness conditions have been folded into the code. Analysis (e.g. for safety) even on this low level implementation may detect errors that are difficult to detect by testing alone.

The refinement steps from more abstract domains to more concrete lower level domains are done by applying transformations, methods, and software components to the system description mechanically. This ensures that there are *no process errors* during construction of the code. The code is *correct the first time* and there is *no error reinjection* caused by making design decisions.

For our abstract production cell domain we would expect to see transformations and methods that refine the abstract components (e.g. N-stage conveyer belt and robot together) with their relationships into a colored Petri net. The refined components have a controller triggering the functionality of the components under circumstances derivable by an analyzer or given interactively by the software engineer. The triggering condition itself may be described using yet another domain of predicate logic using statements over the states of the colored Petri net. This domain is the concrete binding for the transition coloring parameter used to instantiate the colored Petri net for this example.

Using such existing domain implementation knowledge that describe how to realize the abstract concepts effectively and efficiently in terms of lower level concepts, we get *more reliable, and often more effective algorithms for the initial implementation*. Thus, there is *less need for rework*.

The description of the semantics of the source and target domains that participate in a transformation (or method or software component) makes it *possible to verify transformations* (see [Win94]) within a domain (used to optimize the system description) as well as between domains (used to implement high level concepts by lower level concepts). The machine-interpretability of the semantics in DMS is the enabling condition to build tool support for actually performing the proofs. If all transformations applied to construct a software system are proven to be correct the resulting code obviously is correct by construction.

Even if no attempt is made to prove correctness of transformations, *test cases may be derived from the system specification and the transformations applied* to test the final code with critical values.

```
void conveyor_belt_controller()
  { for (;;)
      { // wait for a new state of the conveyor belt, that is for addition or removal of a part
        wait_for_conveyor_belt_state_change();
        // move conveyor belt if possible
        if (conveyor_belt_empty())
          continue; // moving conveyor belt would not change state
        else if (conveyor_belt_last_position_empty()) // faulty inverted condition
          continue; // there is a part at the end of the conveyor belt; moving is unsafe
        else if (conveyor_belt_middle_position_not_empty())
          { if (   (== conveyor_belt_middle_position_part() square)
                !! (== conveyor_belt_middle_position_part() triangle)
               !! (== conveyor_belt_middle_position_part() disk)
               )
              continue; // moving violates liveness as a part at the end of the belt cannot leave belt
          }
        if (== conveyor_belt_first_position_part() square)
          { if ((splicer_has_square())
              { if (!(splicer_has_triangle()) && !(conveyor_belt_second_position_part() triangle))
                  error_conveyor_belt_fed_incorrectly(); // liveness condition violated
                else
                  continue; // moving would violate liveness condition
              }
            else
              { if (win_race_condition()) // try to win race condition to move conveyor belt
                  { move_conveyor_belt_forward(); // we won race condition
                    allow_race();                // allow next race
                  }
                else
                  continue; // someone won race, which results in modifying the conveyor belt state
              }
          }
       else if (== conveyor_belt_first_position_part() triangle)
          ...
        else if (== conveyor_belt_first_position_part() disk)
          { if (drill_has_disk() !! drill_has_torus())
              continue; // moving would violate liveness condition
            else
              { if (win_race_condition()) // try to win race condition to move conveyor belt
                  { move_conveyor_belt_forward(); // we won race condition
                    allow_race();                // allow next race
                  }
                else
                  continue; // someone won race, which results in modifying the conveyor belt state
          }
        else
          error_conveyor_belt_fed_incorrectly(); // wrong part
      } //end for
  }
```

Figure 6. A Control Program for the Conveyor Belt in C

3.4 Traceability Aids Fault Analysis and Inevitable Software Maintenance

Regardless whether the transformations have been proven to be correct, it is important to *trace defective knowledge* whenever an error is detected in some level of description of the system. Candidates for such defective knowledge are:

- the specification of the requirements of the system,
- erroneous transformations, methods, and software components, and
- the semantics of the domains involved in the development (resulting in validity proofs of wrong transformations, methods, and software components and/or wrong analysis results that may have provided preconditions for applied transformations, methods, and software components)

DMS allows traceability between any of the specification fragments, implementation fragments (at any level of abstraction derived from the specification fragments), and the transformations, methods, and software components used in the derivation process. This supports finding defective knowledge involved in an implementation fragment, and/or determining incorrect implementation fragments related to errors found in a specification.

In our production cell example suppose there is an error in the parts interchanged between the conveyor belt and the consumer. Implementation code to ensure the correctness of this interchange can be found in the conveyor belt controller and the consumer controller as well as some controller(s) of the other subsystems. The tracing capability of DMS allows the location of all these places in the C code, each of which is a candidate cause of the problem. In Figures 4 and 6, we have highlighted in grey the requirement for synchronization and its implementation with an incorrectly negated condition.

In any realistic application context, customer demands will change and engineering errors will be made both during the development and the life of the product. Changes to the requirements for the software system, and well as defective implementation knowledge cause the necessity to modify, i.e. *maintain*, the system. For high integrity systems it is unacceptable to modify the code directly as this would compromise the quality of the software. Instead one should modify the defective knowledge or specification and then redesign to ensure the final modified code being trustworthy.

DMS supports maintaining software by recording its original design history and using the traceability to determine the impact of a change, preserving as much of the original design as possible. This reduces not only the overall amount of resources needed to construct the modified system code but also makes it more likely that the outcome of the maintenance process is correct. The *quality* of the code is *not compromised by the need to maintain* it.

Considering that the major effort in software engineering is spent during *maintenance* such a design revision *is* a *key capability* for each practical model of software engineering. For high-integrity software, this is especially important, as successful software must inevitably evolve. Since DMS, in repairing a design, continues to use tested domain knowledge, the error re-injection rate caused by conventional software maintenance methods can be largely avoided.

This is done within a framework of domain knowledge that is (re)used in the construction and maintenance of many different systems. This increases the reliability of the domain knowledge, and thus of the software developed using it, by virtue of the fact that the knowledge is validated by using it in many different places successfully. Each error in domain knowledge detected by even only one of the systems failing results in increasing the reliability of all systems using that domain knowledge by maintaining these systems and replacing the defective knowledge with corrected information. *Reusing the domain knowledge* in the construction and maintenance of different systems *increases the reliability* of each of the systems in virtue of the fact that the knowledge is validated by using it in many places.

3.5 Reverse Engineering and High Integrity Systems

Existing "legacy" HIS systems constructed by more conventional methods still require maintenance. DMS can be applied to such a system for maintenance purposes by first reverse-engineering to obtain a plausible design. Reverse engineering using well-tested domains can help *validate the correctness of the legacy system*, locating errors, and can *also help validate domain knowledge* used in follow-on applications.

It is not necessary to reverse engineer back to a fully abstract problem specification, or to reverse-engineer the entire application at once. For legacy applications, raising the abstraction level only a little may provide considerable insight into the system, as one might obtain by reverse-engineering our sample C program back to colored Petri nets. Further, a long-lived application provides a long time window in which the reverse engineering activity can take place. The only requirement for a software modification is that the relevant part of the design be reverse-engineered.

3.6 Summary of Impact of DMS on HIS

DMS supports the development of high integrity software in many ways. DMS uses problem specific domains for the specification of system requirements. Moreover, layered domains ease the construction process of the system by reducing the size of steps. Simulation and analysis of specifications, i.e. system descriptions, on many levels of abstraction reduce specification errors. Validated domain knowledge in the form of transformations, methods, and software components that are (re)used for many systems increases their reliability. Explicit domain semantics allow proof of the correctness for domain knowledge. Mechanical application of domain knowledge avoids process errors. Traceability of defective knowledge promotes quick identification and recovery. Test case extraction from specifications enables comprehensive mechanical testing. Maintenance of systems is a value adding activity without compromising quality. DMS is grounded in the practical software engineering realm by supporting reverse engineering to maintain existing systems as well as to validate domain knowledge. DMS scales to handle practical sized applications and interactions between multiple engineers. We consequently believe that systems like DMS would be effective support for HIS lifecycles.

4. Status of DMS

As of late 1997, the DMS tool suite is rather more architecture than reality. A team of 8 engineers are presently working full-time on the system. The parallel processing language, PARLANSE (the implementation domain for DMS), is running robustly on SMP Windows/NT systems, and is being used to develop components of DMS. The initial transformational core is nearing operation, including an SMP-capable hypergraph foundation, supporting a sophisticated rewrite engine based on associative-commutative completion routines. The rewrite engine is intended for use on rewrites extracted from algebraic specifications, which in turn enables symbolic analysis and simulation using domain denotational semantics. An initial target domain of full ANSI C++ is under construction.

We hope to have a domain-definition domain operative by 1998, along with an incremental editor driven by the domain syntax. Other domains will be defined in 1998, and the key capability, traceability, will be implemented, providing the basis for the modification capabilities outlined here. A base of domains is required to support reverse engineering capabilities, and we expect these to grow over time.

5. Related Work

Formal Methods

In [PW95] an evolutionary process model for manual program development is presented, where the correctness of all development steps is checked by suitable verification tools. The model centers around a development graph which contains the specifications, programs, and proofs together with relations between them. Due to the huge proof obligations for large application systems this approach is not practical for developing reliable software. It also does not allow mechanical support for system modifications.

Different methods have been applied to a case study for constructing a control system for a reactive system, a production cell (see [LL95] for a description of 18 of currently 35 approaches) many (if not all) of which can benefit from the DMS tool support for software construction.

Another approach for developing reactive control systems, the Abstraction-Synthesis-Transformation methodology, is described in [Win96]. The central idea of this approach is to describe the real world as directly as possible, abstract away unnecessary details, synthesize a possible controller from the abstracted description, and finally optimize the synthesized code by applying optimizing transformations. Such an approach would be well supported by the DMS technology.

Maintenance Tools

In [Wil83] transformational metaprograms, not code, are proposed to be the major software product. Implementation decisions can be replayed in the exact order they were made on a previous specification. This is naïve replay and fails when some specification change invalidates an implementation decision.

The *Programmer's Apprentice* (see [Wat88, RW90]) constructs code from an abstraction by applying and completing interactively chosen clichés to it. Code modification is done by abstracting the code into a higher level specification, allowing arbitrary changes, and then reimplementing. The transformational derivation and its justification are not recorded; they are lost.

The *Maintainer's Assistant* (see [WCM89, WB95]) uses input/output behavior preserving, possibly abstracting transformations applied to a wide spectrum language to maintain the software. Apparently the system does not support non-behavior-preserving modifications of the software. It appears to be used only for porting and translation rather than explanation and modification.

Kestrel has built a replay system that attempts to match implementation decisions with changed circumstances by using a heuristic approach (see [Gol90]). However, this may result in reapplying an old decision under inappropriate circumstances, which in turn may cause cascading spurious effects. This means that the resulting implementation may not have the expected desired properties. Thus, these properties have to be reverified manually.

6. Concluding Remarks

We have argued, unsurprisingly, that the construction of high integrity software (HIS) requires formal methods. More importantly, we argue that realistic deployment scenarios for HIS requires formal support for *modification*. We described a developing software engineering environment, the Design Maintenance System (DMS), that supports domain-based incremental, transformational construction and maintenance of large application systems. We discussed the qualitative effects of using DMS for HIS, and conclude that DMS has many of the necessary properties for formally maintaining High Integrity Software.

This work has been supported by the National Institute of Standards and Technology (NIST), Advanced Technology Program (ATP) under the Component-Based Software program.

References

[Bax90] I.D. Baxter. *Transformational Maintenance by Reuse of Design Histories*. PhD thesis, University of California at Irvine, 1990.

[Bax92] I.D. Baxter. *Design Maintenance Systems*. *CACM* 35(4), Apr. 1992, pp. 73-89.

[Bax95] I.D. Baxter. *Design (Not Code!) Maintenance*. ICSE-17 Workshop on Program Transformation and Software Evolution, 1995.

[BM97] I.D. Baxter and M. Mehlich. *Reverse Engineering is Reverse Forward Engineering*. Working Conference on Reverse Engineering, 1997.

[BP97] I.D. Baxter and C.W. Pidgeon. *Software Change Through Design Maintenance*. Proceedings of International Conference on Software Maintenance '97, 1997, IEEE Press.

[Boe81] B. Boehm. *Software Engineering Economics*. Prentice-Hall, 1981.

[Gol90] A. Goldberg. *Reusing Software Developments*. ACM SIGSOFT Symposium on Software Development Environments. Kestrel Institute Technical Report KES.U.90.2, 1990.

[Gui83] T. Guimarares. *Managing Application Program Maintenance Expenditures*. Communications of the ACM 26(10):739-746, 1983.

[HD95] M. Heiner and P. Deussen. *Petri Net Based Quality Analysis – A Case Study*. Technical Report I-08/1995, Institut für Informatik, Brandenburgische Technische Universität Cottbus, 1995.

[Kan91] E. Kant, F. Daube, E. MacGregor, and J. Wald. *Scientific Programming by Automated Synthesis*. In: M.R. Lowery and R.D. McCartney (eds.). Automating Software Design. MIT Press, 1991.

[LL95] C. Lewerentz and T. Lindner. *Formal Development of Reactive Systems: Case Study Production Cell*. Lecture Notes in Computer Science 891. Springer, 1995.

[McC87] R.D. McCartney. *Synthesizing Algorithms with Performance Constraints*. PhD thesis, Brown University, 1987.

[Nei84] J. Neighbors. *The Draco Approach to Constructing Software from Components*. IEEE Transactions on Software Engineering 10(5):564-574, 1984.

[Par90] H. Partsch. *Specification and Transformation of Programs: A Formal Approach to Software Development*. Springer, 1990.

[PW95] P. Pepper, M. Wirsing, R. Betschko, M. Broy, S. Dick, K. Didrich, J. Faulhaber, W. Grieskamp, H. Hußmann, M. Mehlich, and W. Reif. *KorSo: A Methodology for the Development of Correct Software*. In: M. Broy and S. Jähnichen. *KorSo: Methods, Languages, and Tools for the Construction of Correct Software*. Lecture Notes in Computer Science 1009, pp. 27-57. Springer, 1995.

[RW90] C. Rich and R.C. Waters. *The Programmer's Apprentice*. ACM Press, 1990.

[Wat88] R.C. Waters. *Program Translation via Abstraction and Reimplementation*. IEEE Transactions on Software Engineering 14(8):1207-1228, 1988.

[WB95] M. Ward and K.H. Bennett. *Formal Methods for Legacy Systems*. Journal of Software Maintenance: Research and Practice 7(3):203-219, 1995.

[WCM89] M. Ward, F. Callis, and M. Munro. *The Maintainer's Assistant*. Conference on Software Maintenance, 1989.

[Wil83] D. Wile. *Program Developments: Formal Explanations of Implementations*. Communications of the ACM 26(11):902-911, 1983.

[Win94] V.L. Winter. *Proving the Correctness of Program Transformations*. PhD thesis, University of New Mexico, 1994.

[Win96] V.L. Winter. *An Overview of the AST Methodology*. Sandia National Laboratories, 1996.