Constraint Maintenance and Transformation Based Design for High-Assurance Software and Systems

Lindsay Errington[1], Bruce H. Krogh[2], John Anton[1] and Peter Coronado[3]

[1]Kestrel Institute / Kestrel Technology, lindsay@kestrel.edu,anton@kestrel.edu
[2]Carnegie Mellon University, krogh@ece.cmu.edu
[3]Varian Medical Systems, peter.coronado@varian.com

April 10, 2005

Abstract

This paper proposes *constraint maintenance and transformations* (CMT) as a basis for developing high assurance software and systems. CMT goes from requirements to implementation through the application of *transformation rules* that *(i)* guarantee constraints are satisfied at each step *(ii)* supports changes and upgrades through the modification of constraints (requirements) and their propagation through the transformation rules and *(iii)* maintains a record of the transformations and proofs as documentation that can be used for certification of the final implementation. We advocate the development of tools to support CMT methods for full life-cycle development and maintenance of medical device software and systems (MDSS). In addition to promising lower time and cost for developing high-assurance systems, CMT offers direct support of standards currently being considered for MDSS development and certification.

1 Background and Motivation

The design space for high-assurance medical device software and systems (MDSS) has many dimensions, far more than are faced, say, in conventional software engineering. Not only are there functional constraints, but also constraints on safety, timing, cost, space, time to market, security and reliability to name a few. There are also many dimensions to the solution space. One must choose from different software architectures and different algorithms, as well as different hardware platforms and components.

The task for an engineer is to navigate through the design space, assess the tradeoffs among the different design choices, and make decisions that satisfy the design constraints. This is complicated by the fact that each decision changes the design constraints. For example, choosing a bus architecture may limit the choice of DSPs downstream. Thus, managing design constraints is a major challenge

Although casting the design problem in terms of constraints and constraint refinement may seem intuitive and perhaps obvious, there are relatively few tools that assist engineers in this process. There are tools and methodologies that track requirements or introduce a degree of discipline, but nothing that actually checks for consistency of design decisions or enforces a design discipline.

Languages and tools commonly used to design embedded systems are often *model based* (as opposed to *constraint based*). These include languages like Stateflow, Statecharts, hybrid automata and data-flow languages like Simulink. The attraction of these languages is that they allow designs to be represented at a

higher-level of abstraction (than say code) and the associated tools have with simulation and sometimes verification capabilities. However, although these languages are well-suited for representing a particular design, they don't help the engineer arrive at the design. This is because these tools cannot be used to express, let alone simulate or analyze, incomplete models. Engineers are thus forced to commit to a particular design prematurely.

The fact that current tools are model-based rather than constraint-based means that the design requirements (the initial constraints) do not play a direct role in the design process. They serve only as a reference for the engineer who has to invent a design for which it is hoped the constraints are satisfied. This disconnect between requirements and design has a number of implications. First, validating, verifying and certifying that a design meets its requirements become activities that are completely separate from the development process. For safety critical systems, where a high degree of confidence is required, these are time consuming and expensive tasks. Second, it means that any ambiguity that arises from unwritten, implicit and hence unexamined requirements goes undetected until late in development. Similarly, it means that design errors are often discovered late in development (often during testing). Finally, it becomes impossible to assess the full implications of changes to the requirements. Often when the design requirements are not met or when the requirements change, there are few options other than massive redesign and redevelopment. This cycle may be repeated many times.

We conclude that failure to manage the design constraints in an effective way leads to increased development time and costs. These problems will become only more acute with the increase in *(i)* hardware capabilities, *(ii)* the demand for greater functionality in medical devices and *(iii)* the use of these devices in clinical settings.

2 Beyond Model-Based Design

The discussion above suggests we consider an alternative paradigm for the development of medical device software and systems (MDSS), where the designer begins with the requirements for a system expressed as a collection of constraints. These constraints would deal with both functional and non-functional attributes of the desired system and might include timing, reliability, and uncertainty. Then, as much as possible, a system would be designed through the application of a collection of *transformation rules*. Thus, the two key themes in this paradigm are *constraint maintenance* and *transformations* (CMT).

Each transformation rule would be a precise and formal encoding of some domain-specific design knowledge, essentially a "design pattern." The application of a rule would reify the abstract specification into something more concrete. A rule might, for example, factor the problem into an architecture or select a component, data-structure, or algorithm scheme. The tool supporting the paradigm must ensure that the selected rule is applicable in the current context, manage the design constraints on behalf of the engineer, and ensure that consistency is preserved with each transformation step. In CMT, the requirements and "implementation" languages are combined, whereas hitherto they have been separate.

It is important to note that that a tool supporting CMT records not only the current design and constraints, but also the abstraction hierarchy and the history of steps that brought one from the initial requirements. What is finally delivered is not simply a collection of code, state machines and paper documents, but a highly structured formal, query-able document that weaves together the requirements and the design history and hence the rationale for the design.

Note also that CMT is not simply a "top-down" design methodology. At any point, an engineer can backtrack

and explore and assess different design scenarios and tradeoffs in the design space. The transformation records also support subsequent maintenance, including upgrades and changes in the design requirements. Thus, CMT provides full life-cycle support.

Encoding design patterns as transformation rules is a key feature of CMT. Codifying patterns means that the art of domain-specific design will be captured in the transformation rules. Creative effort will be spent creating patterns rather than specific designs. These patterns will then be applied to specific cases. Using design patterns also make it possible for knowledge to persist beyond a particular engineer's involvement in a project.

3 Certification of MDSS

At present, certification is not an active concept in the medical device world. Nevertheless, a number of international standards are relevant to medical devices. These include AAMI/SW68 [1], IEC 60601-1-4 [2] and the draft standard IEC 62304 [3].

Like the others, the draft standard *IEC 62304 Medical device software - Software life cycle processes* emphasizes levels of abstraction, verification of each level of abstraction, (informal) refinement, the development process, and traceability (similar in some respects to DO-178B, but with fewer levels). The developer must provide written documentation for the entire process.

It is interesting to note that formal methods play no role in any of these standards. Indeed, the draft standard states "This standard does not require the use of a formal specification language". In other words, certification artifacts are to be written in natural language. This contrasts with the Common Criteria (CC) where the highest assurance level (EAL 7) explicitly requires formal mathematical models and proofs. While few systems are likely to be designed to meet level EAL 7, it is interesting that the CC at least has provision for formal methods and associates such methods with the highest level of assurance.

We believe that standards and certification are likely to become more important for MDSS (in the same way that DO-178B is for avionics). Moreover, we believe that as certification is imposed, formal methods will be introduced to reduce the cost of developing high-assurance software and systems, despite the current perception that formal methods are expensive. It will not be economically feasible to produce all the tests and documentation required by current standards practices as systems become more complex and performance criteria become more stringent.

In anticipation of this, it would be useful to transcribe and extend the IEC standard in a way that assumes the use of formal methods. Wherever possible, it should prescribe that certification artifacts, such as proofs of correctness, should be machine generated. Formal artifacts other than proofs should also be considered.

The CMT paradigm delivers *certification by design*. The requirements for an embedded system are used directly in the design process. This, plus the fact that the correctness proof for each refinement step is recorded as the transformation rules are applied means that certification can be integrated with the design process rather than be carried out through separate tasks.

4 Comparison of CMT to Formal Verification

It is useful to contrast the proposed CMT approach with formal verification. In verification, a theorem prover or model-checker is used to establish that a system or component has, or does not have, some property of interest. The challenge with verification technologies has been, and remains, scaling to larger problems. Even with recent advances in model-checking technology and state-space abstraction techniques, post verification of real-world systems is intractable. Systems are simply too large for exhaustive analysis.

By contrast, we anticipate that the reasoning necessary to support a transformation and refinement paradigm will be relatively modest. Rather than applying post verification to a complete system, one must establish only the correctness of a relatively small *change* to a design brought about by a transformation. In this case, proving correctness means proving that a design decision does not violate the current collection of constraints. In this sense, the reasoning is *incremental*. The correctness of the final design rests on the *composition* of a sequence of verified refinement steps.

5 Conclusions

This position paper advocates the development of a new approach to the development of high-assurance software and systems that emphasizes contraint maintenance and transformation-based design. We also propose that a certification process be developed for MDSS that uses artifacts from the application formal methods as the principal supporting documents in the certification process. CMT offers an attractive way to generate these artifacts.

References

[1] *ANSI/AAMI SW68, Medical device software - Software life cycle processes*, 1 edition, June 2001.

[2] *IEC60601-1-4, Medical electrical equipment, Part 1: General requirements for safety, Part 4: Programmable electrical medical systems*, 1.1 edition, April 2000.

[3] *IEC 62304, Medical device software - Software life cycle processes*, December 2004. IEC Draft Standard.