Design of safety critical systems by refinement

Alex Iliasov, Arseniy Alekseyev, Danil Sokolov and Andrey Mokhov
Newcastle University, Newcastle upon Tyne

***Abstract*—An increasingly large number of safety-critical embedded systems rely on software to prevent and mitigate hazards occurring due to design errors and unexpected interactions of the system with its users and the environment. Implementing a safety instrumented function in the way advocated by the traditional software methods requires an intimate understanding and thorough validation of a complex ecosystem of programming languages, compilers, operating systems and hardware. We propose to consider an alternative where a system designer, for each individual problem, creates in a correct-by-construction manner both the design of a system and its compilation and execution infrastructure. This permits an uninterrupted chain of a formal correctness argument spanning from formalised requirements all the way to the gate-level characterisation of an execution environment. The past decade of advances in verification technology turned the mechanical verification of large-scale models into a reality while the pressure of certification makes the cost of a formally verified development routine increasingly acceptable.**

The proposal fits the Grand Challenge for Computer Research posed by Hoare in 2003, namely, development of a Verifying Compiler which not only mechanically translates a given program from one language to another but also verifies its correctness according to a formal specification. This allows meeting the most stringent software certification requirements such as SIL 4. We illustrate the vision with a small case-study developed using the Event-B modelling notation and tools.

I. Introduction

Presently, the design of safety-critical embedded systems is based on the principles of mainstream software development: a collection of requirements posit a problem a solution to which is encoded in one or more high-level programming language artifacts; the final product is obtained in the form of compiled software executed by an off-the-shelf processor module. The correctness of such product depends on the accuracy of requirements, the correctness of a program, the correctness of compilers employed to construct machine code, and, finally, the correctness and operational properties of hardware running compiled code. No amount of testing would provide a definite guarantee that the behaviour exhibited by a product is the intended behaviour as stated in the original requirements document.

It is widely recognised that the introduction of a modelling stage, in particular formal modelling, may considerably improve the quality of delivered product. A range of industrial applications demonstrate that building a large-scale formal specification is possible and, in certain application domains, delivers tangible economic benefits. Formal reasoning has been fruitfully applied at various development stages such as validation of requirements, early design, code-level and hardware-level verification. However, the techniques vary considerably between the stages resulting in the lack of continuity in the correctness argument constructed with a formal model. At the practical level, this makes formal modelling less appealing.

978-3-9815370-2-4/DATE14/ © 2014 EDAA

In this work we put forward an approach to the construction of safety critical systems that advocates a strict, top-down, formally verified derivation of an implementation starting from a set of requirements and ending with an RTL-level hardware description. Taking into the consideration the advances, during the last decade, in automated verification technology, we believe that it may be feasible, technologically, and viable, financially, to develop a safety critical information system, such as an embedded control system, completely by formal refinement without involving a programming language, and, to a certain extent, off-the-shelf CPU components. One may expect such a development method to help to meet the most stringent software certification requirements, e.g., SIL 4 of the European IEC EN 61508 standard. A formal proof of model consistency and refinement by construction subsumes the MCDC testing procedure (modified condition/decision coverage testing) recommended for IEC 61508 SIL 4 as well as DO-178B, level A.

One novelty of the work is a two-stage transition into a final product. To reduce the gap between a specification and its implementation we first develop a virtual machine (VM) tailored to the problem at hand and encode the specification as program to be run by the VM. Then we proceed with the mapping of a VM definition into a deployable hardware format such as FPGA or ASIC. The program and hardware thus obtained are formal derivations of a high-level specification with explicit proofs available for an independent check. This leads to an extremely compact trusted base potentially reduced to solely a proof checking routine. This also makes the last stage - the hardware synthesis - independent of the overall system scale since hardware complexity correlates only with number of VM instructions.

II. Overview

Our proposal consists in replacing the dominant approach to embedded system development based on informal transition from requirements to a non-specific implementation and then to final product by a design flow based on small, verifiable transformation steps spanning across requirements and final, deployable product (Figure 1, *Existing flow*).

By a non-specific implementation we understand a hardware-software complex where the hardware is abstracted in the form of executable software and the software has few explicit constraints on the prospective execution environment. To realise our vision we advocate the use of assisted formal refinement steps. This forces us to abandon use of traditional compilation, programming languages and invites to explore the

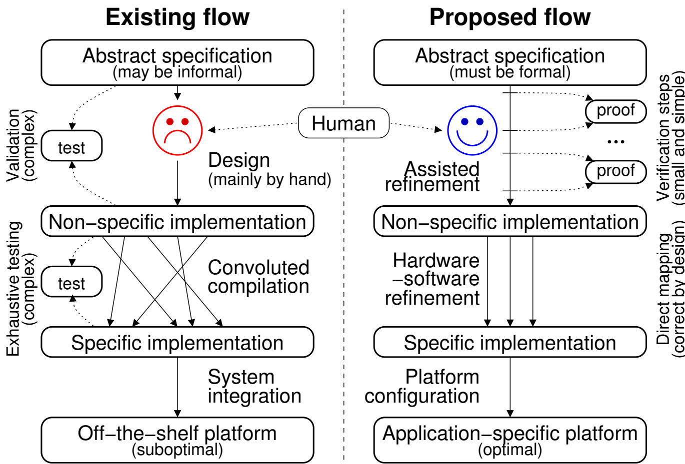


Fig. 1. Existing and proposed design flows.

benefits of application-specific hardware platforms (Figure 1, *Proposed flow*).

As a starting point, we assume the existence of a structured requirements document written in a natural language.

a) Abstract specification: The first step is the analysis of requirements with a formal notation and verification of high-level properties with the aim to uncover any requirement inconsistencies. An abstract design gradually incorporates system requirements in a succession of refinement steps.

b) Non-specific implementation: At this stage a modeller starts incorporating concrete solutions. The initial refinement steps concentrate on the architectural design, perhaps employing a form of graphical notation. The design grows to include concrete behaviour and instances of algorithms and protocols. Once a modeller is convinced that the design is precise enough - nothing of importance is left under-specified - the focus shifts to the design of the prospective execution environment.

c) Virtual machine design: Now, with a good understanding of the concrete design, a modeller needs to construct a VM that would run a program realising the concrete design. He needs to identify an optimal architecture (e.g. LOAD/STORE or a stack machine) and an instruction set to match the prospective implementation. It is necessary to balance the effort of refining a concrete design into a program of a VM and the cost of synthesising provably correct hardware realising a VM. A VM description comes in the form of a formal specification of an interpreter executing a program made of VM instructions.

Further concerns, such as performance, power efficiency, and hard real time requirements may affect the choice of architecture and instruction set.

d) Refinement patterns: The final refinement step of the concrete design is likely to be a large model, perhaps many thousands of statements. This model needs to be refined into the virtual machine specification running a certain program. A set of *refinement patterns* is developed to transform the specific instance of concrete design into a model made of exactly the execution steps provided by the VM. This patterns may be statically checked (shown correct for all possible inputs) or have their output verified as in the case of a manual program construction.

e) Compilation-by-refinement: This stage is concerned with the construction a provably correct program implementing the concrete design from step *d)*. First, a generic procedure converts a concrete design into a program to be run by the developed VM; another procedure generates a special version of the VM specification, instantiated with the generated program, to construct a refinement proof. This VM specification must be proven to be a refinement of the concrete design. Technically, this is a tricky case of refinement combining large-scale data and behaviour refinement in one step: all the variables of concrete design disappear and become memory cell addresses or register names; the abstract behaviour is encoded in a data structure - a program of the virtual machine. From this point, the focus shifts to the construction of hardware realising a virtual machine.

f) Specific implementation based on VM: A modeller needs to expand the definition of each instruction into a circuit netlist and provide a proof of the implementations being equivalent to the specifications. This is fulfilled in a combination of the following techniques. The instruction operations are implemented by conventional logic synthesis tools (e.g., Synopsys Design Compiler) and then checked against specifications by the equivalence checking tools (e.g., Synopsis Formality) [1]. The latter are based on SAT solvers and can handle components with thousands of gates [2]. To further simplify the equivalence checking effort, a library of certified efficient implementations may be provided for common functionality such as arithmetic operations. The top-level logic is subsequently constructed using a generic template for the control flow - a set of multiplexors to select instruction operation and its parameters. This template is instantiated with the previously verified operation blocks, resulting in a verified netlist for the whole circuit. The key idea is to separately prove the refinement of a generic control flow template and the refinement of each individual instruction operation, thus making this method linearly scalable with the size of instruction set. Ultimately the equivalence is shown by existence of refinement from the abstract virtual machine interpreter constructed at step c) to the hardware implementation.

III. Example

We consider a subsystem of fire safety monitoring and response logic concerned with monitoring several temperature sensors. The subsystem implements the following functions:

- information is collected from an array of temperature sensors;
- on a request, the average temperature is reported;
- the controller may be reset to make it forget any accumulated information.

We use the Event-B method [3] and the Rodin Platform[4] to capture abstract specification and carry out formal refinement and verification. The end result of the development are formally defined and verified computing platform and

firmware. Our verification technology is theorem proving and constraint solving and the vast majority of the verification conditions in the development were discharged automatically.

A. Abstract design

In the abstract design we try to faithfully reproduce informal requirements. The model state captures accumulated sensor readings, the last reported average and a flag indicating the freshness of the average value: $n \in \mathbb{N} \wedge arr \in 1 \mathinner{..} n \rightarrow \mathbb{Z}, answer \in \mathbb{Z}, done \in \text{BOOL}$. Here, n and arr define a finite array as a total function over some integer range. The initial state is $n := 0 \parallel arr := \varnothing \parallel answer :\in \mathbb{Z} \parallel done := \text{FALSE}$. Each operation of the controller is formalised in a dedicated atomic state transition.

```
read  =  any x where
             x ∈ ℤ
         then
             arr := arr ∪ {n + 1 ↦ x}
             n := n + 1 ‖ done := FALSE
         end
  ...
```

This kind of model may be presented to a customer to sign off, together with accompanying explanations, and, perhaps, animation results, as a formalised requirements document from which a certified product is to be created.

B. Concrete design

The complete record of sensor readings saved in arr is superfluous. In the concrete design we elect to simply maintain the sum of reported values. Abstract variable arr is replaced by concrete variable sum; the two are linked by the *refinement invariant* $sum = \sum_i arr(i)$. The $read$ operation is now refined to become the following

```
read  =  any x where
             x ∈ ℤ
         then
             n := n + 1 ‖ done := FALSE ‖ sum := sum + x
         end
```

Other operations are changed in an obvious manner: reset sets sum to zero and mean_good uses sum in place of $\text{SUM}(arr)$. The concrete design for our simple problem is now complete. The Rodin Platform automatically generates proof obligations necessary to establish that the concrete design *simulates* the abstract design.

C. Virtual machine model

To construct the final embedded system we are going to decompose the design into the specification of a virtual machine (to be refined into computing platform) and a firmware to be run by the virtual machine. We apply refinement to show that the specific combination of VM and firmware correctly implement the concrete design. The first step is to build a formal model of a VM. One can do either from a scratch, as we in this simple example, specify the ISA and memory architecture of existing platform, or use an existing model corresponding to the desired platform [5].

Our system requires little memory, few arithmetic instructions and some basic control flow and means for communicating with the environment. As a memory model we use a register file with rn registers of finite type $\text{V} \subset \mathbb{Z}$: $m \in 0 \mathinner{..} rn \rightarrow \text{V}$. Inputs i and outputs o are modelled by register banks i and o: $i \in 0 \mathinner{..} in \rightarrow \text{V}, o \in 0 \mathinner{..} on \rightarrow \text{V}$. Identifiers rn, ri, ro and V are the *parameters* of our VM specification and will to be instantiated with concrete values.

A program is stored in ROM and is encoded as a sequence $(oc, p1, p2)$ of instruction code oc and parameters $p1$ and $p2$: $oc \in 0 \mathinner{..} pl \rightarrow \text{II}, p1, p2 \in 0 \mathinner{..} pl \rightarrow \text{P}$. Finite set $\text{P} \subset \mathbb{N}$ defines the type of instruction parameter; II is the set of machine instructions: $\text{II} = \{\text{SET}, \text{MOV}, \text{IN}, \text{OUT}, \text{ADD}, \text{SUB}, \text{JLE}\}$.

A program counter $pc \in 0 \mathinner{..} pl + 1$ gives the index of the current instruction ($pc = pl + 1$ halts the machine). An individual instruction for some pc value is defined by $(oc(pc), p1(pc), p2(pc))$.

The following is an excerpt from the VM specification. For most cases, one Event-B event defines a single instruction. Each instruction contains a decoder (i.e., $oc(pc) = \text{ADD}$), a program counter increment and the instruction body.

```
add   =  when
             oc(pc) = ADD
         then
             m(p1(pc)) := m(p1(pc)) + m(p2(pc))
             pc := pc + 1
         end
jle1  =  when
             oc(pc) = JLE ∧ m(p1(pc)) ≤ 0
         then
             pc := p2(pc)
         end
  ...
```

In order to simplify the verification VM design, the integer division operation was left out. Instead, we are going to further refine the concrete design to realise division via an iterated subtraction.

D. Compilation by refinement

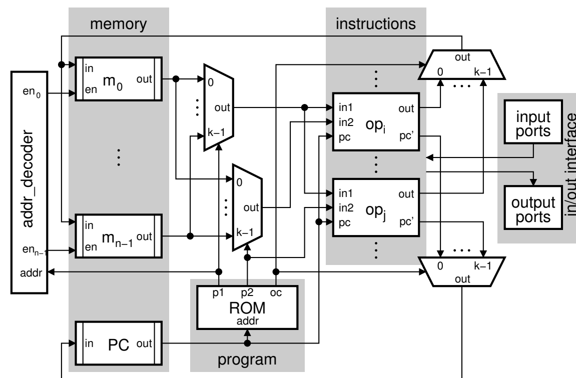
With the VM model in place, we proceed to extract the firmware from the concrete design and prove that the VM running the firmware simulates the concrete design. This is a stage, called *compilation by refinement*, when a concrete specification is turned into a program (firmware) for a given platform while proving a refinement relationship between the concrete specification and the platform initialised with a given firmware.

The complexity of a refinement link and the abundance of details make compilation by refinement challenging for a modeller to conduct manually. To a considerable extent it may be automated by defining *refinement patterns*: mechanised rewrite procedures automatically building a new refinement step. There are two major pattern classes.

The first kind 'flattens' a model by expanding complex substitutions of the form $t := a \times b + c$ into *atoms* $t' := a \times b$ and $t := t' + c$. Such expansions must provably preserve the semantics of the original statement, however, they should not require a significant extra verification effort. Atelier B BART

```
SET   zero  0           loop2:
SET   one   1           SET   b     0
SET   c     0           SUB   b     a
SET   sum   0           JLE   b     done
start:                  ADD   a     c
IN    o     d           SUB   i     one
JLE   o     reset       done:
SUB   o     one         OUT   i
JLE   o     insert      JLE   zero  start
mean:                   reset:
SET   i     0           SET   sum   0
MOV   a     sum         SET   c     0
loop1:                  JLE   zero  start
JLE   a     loop2       insert:
SUB   a     c           ADD   sum   d
ADD   i     one         ADD   c     one
JLE   zero  loop1       JLE   zero  start
```

a)



b)

Fig. 2. (a) Extracted program in the VM1 assembler; (b) a circuit architecture implementing VM1.

tool [6] is one example of mechanising such patterns; for this example we did such transformation mahnually.

The second kind constitutes patterns that seek to reduce atoms not present in the target VM (e.g., the division operation in our example) into a specification that might be compilable into the VM. One example is the integer devision operation that has to be expanded into a loop realising division via iterated subtraction. It is, essentially, a form of template-based complition though at the level of a formal model.

E. Circuit synthesis

The VM is implemented as a digital circuit which consists of (a) program - a ROM of opcodes and parameter addresses; (b) memory - a bank of registers to store program variables; (c) instructions - synthesised components for operations; (d) in/out interface - a set of ports to communicate to the environment; (e) control structure to conduct the flow of data. A high-level schematic of such a circuit is shown in Fig. 2 with components $\{a .. b\}$ shaded out and the control structure being the rest of the circuit. Formally, this circuit implements a function $(m', out) = f(m, in)$, where m is the memory state (including the pc register), in and out are the states of input and output ports respectively, and the next-state function f is calculated by program, instructions and control components.

Historically, in order to simplify the design of large systems, the circuit is assumed to change its state instantly and simultaneously for all the registers. In practice this behaviour is enforced by a periodic global clock signal. At each beat of the clock (not shown for clarity) a register, if enabled through input 'en', overwrites its stored value 'out' by the data supplied at input 'in'. If a register is not enabled it holds its previous state; this can be efficiently implemented through clock gating mechanism [1]. The period of the clock waveform is chosen long enough to allow the combinational logic (which implements the next-state function f) to complete all the switching. The minimal clock period that satisfies this requirement is estimated by static timing analysis (e.g. Synopsys PrimeTime) [1].

Note the uniformity and potential for scalability of the control logic. It uses the opcode oc and parameters p1 and p2 of the current program instruction to select which of k operations to perform and which of n registers to use. In the example of Figure 2, one can distinguish two families of instructions: those which take two registers as an input (op_i), and those which take one register and a constant (op_j). The addr_decoder component selects where to store the instruction result; in the basic case, it converts the $p1$ address from a binary form into one-hot encoding.

IV. Conclusion

We have outlined initial ideas on how to design in a formal, top-down manner a program without the use of a programming language and an accompanying compiler. The intermediate stages of a formal the development process may be presented as an objective certificate of correctness to a certifying body.

Our approach starts at the level of an abstract design, naturally flowing from a requirements study thus permitting the verification of a complete system rather then its implementation aspects. This allows to elude numerous challenges of reasoning about programs written in imperative language, verifying their transformation into runnable code (which, despite a resounding success of the CompCert project [7], remains a formidable obstacle), and dealing with intricate and often poorly documented off-the-shelf hardware designs.

The technique has the potential to scale to large problems due to the factoring of overall design into a firmware data and a VM specification.

References

[1] L. Scheffer, L. Lavagno, and G. Martin, *EDA for IC System Design, Verification, and Testing (Electronic Design Automation for Integrated Circuits Handbook)*. Boca Raton, FL, USA: CRC Press, Inc., 2006.

[2] S. Disch and C. Schollm, "Combinational equivalence checking using incremental sat solving, output ordering, and resets," in *Proceedings of the 2007 Asia and South Pacific Design Automation Conference*, ser. ASP-DAC '07. IEEE Computer Society, 2007, pp. 938–943.

[3] J.-R. Abrial, *Modelling in Event-B*. Cambridge University Press, 2010.

[4] The RODIN platform, online at http://rodin-b-sharp.sourceforge.net/.

[5] J. D. Carpinelli, *Computer Systems Organization & Architecture*. Pearson Education, 2001.

[6] Clearsy, "*AtelierB: User and Reference Manuals*," available at http://www.atelierb.societe.com/index_uk.html.

[7] X. Leroy, "A formally verified compiler back-end," *Journal of Automated Reasoning*, 2009.