On the Formal Development of Safety-Critical Software

Andy Galloway, Frantz Iwu, John McDermid and Ian Toyn

Department of Computer Science, University of York
Heslington, York, YO10 5DD, UK
{andyg, iwuo, jam, ian}@cs.york.ac.uk

Abstract. We reflect on the formal development models applicable to embedded control systems in light of our experience with safety-critical applications from the aerospace domain. This leads us to propose two complementary enhancements to Parnas' four-variable model, one elaborating the structure outside the control computer, and the other elaborating the structure inside the control computer. We then identify several challenges which illustrate why formal development in this domain is difficult, and report our own progress in meeting these challenges. Finally, we outline the residual issues, which form the agenda for our future work.

1 Introduction

It has often been argued that formal development is necessary in order to achieve the extremely low failure rates demanded for safety-critical software. Accordingly, this principle is embodied in a number of standards [1][2]. However, whilst there are good examples of the application of static program analysis techniques to safety-critical software, e.g. [3], there are very few examples of the use of "classical" formal approaches such as those based on the notion of refinement ([4] is a rare example). Indeed, there are many practical and theoretical difficulties in applying such models.

The purpose of this paper is to outline a sound technical basis for the formal development of safety-critical systems, identify recent progress in developing such a process (along with associated tools), and highlight future research challenges.

The paper starts by considering development models applicable to safety-critical systems, and uses them to reflect on the scope and limitations of classical approaches to formal development. We propose two complementary, but orthogonal, enhancements of Parnas' four variable model. The first enhancement identifies additional structure outside the control computer, whilst the second focuses on the structure inside the computer.

The analysis is then expanded by considering some of the challenges that arise in the practical development of safety-critical systems, reflecting our experience with a range of avionics applications. This is used to propose a model for formal development of safety-critical software, to outline progress being made towards realising such a model, and to identify residual research challenges.

2 Development Models

In developing safety-critical systems we need to model the environment (air, passengers, roads, etc.), the top-level system, e.g. an aero-engine, which we term the "platform", the control, or embedding, system, e.g. a Full Authority Digital Engine Controller (FADEC) and the embedded system (computing system and software). Few software development models relate the software to the embedding system/environment; counterexamples are Dave Parnas' four variable model [5] and Michael Jackson's Problem Frames [6]. Parnas' model distinguishes:

- monitored variables;
- controlled variables;
- input variables;
- output variables.

The first two represent the environment and/or platform; the control system senses the monitored variables and attempts to control the environment by influencing the controlled variables (both the sensing and influencing processes may be indirect i.e. via other real-world variables). For example a FADEC senses cockpit thrust demands, various air temperatures and pressures along with engine shaft speeds (the monitored variables), and modifies fuel flow (amongst other things) in order to influence the level of thrust (the controlled variable) in the required way.

The input and output variables are the values seen or produced by the computer – perhaps the output of an analogue to digital (A/D) converter at the input, and the contents of a register which goes through digital to analogue (D/A) conversion to produce a current to drive a motor or valve.

Abstractly, *requirements* for the control system are stated in terms of relationships over the monitored and controlled variables, whilst *specifications* for the computer system are stated in terms of input and output variables. To give a complete specification also requires a definition of the relationship between the monitored variables and the inputs, as well as between the output variables and controlled variables. (Parnas' approach does not distinguish between the environment and platform; our proposed enhancement makes such distinctions explicit, in a way which we believe adds engineering value.)

Jackson's approach is not constrained to embedded systems, and so does not identify specific classes of variables. It does however introduce the notion of domain models, which encapsulate properties of the wider system; these can be used to represent the nature of the environment, platform and embedding system. Thus, for example, a domain model could be used to explain the relationship between the monitored and input variables in Parnas' approach. Both approaches are relevant to the development of embedded systems; but experience with embedded systems such as FADECs suggests the need for an elaboration of these models.

In Parnas' approach the behaviour of the physical environment (Nature) is described by a relation, NAT. The basic model is illustrated in Fig. 1, which shows the

system decomposition on the left and the relationship of elements of the specification set on the right:

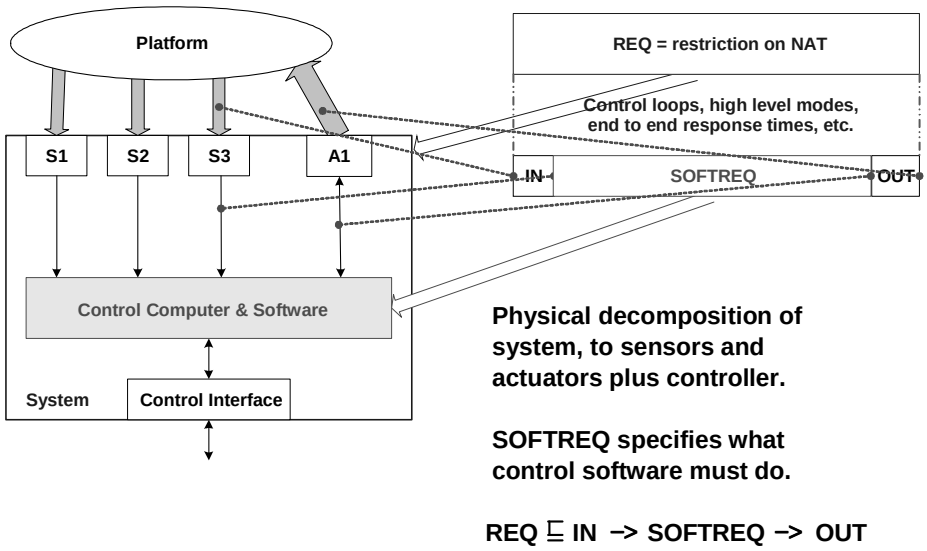

Fig. 1. Representation of Parnas' Four Variable Model

The arrows from the platform are the monitored variables; the reverse arrow is the controlled variable. In the case of an engine controller many of the inputs are environmental properties, e.g. air pressures, at defined points in the engine; other are specific properties of the engine, e.g. shaft speeds.

The input and output variables relate to the control computer and software. The sensors (e.g. S1) map the monitored variables to inputs, represented by relation[1] IN, and the actuators (e.g. A1) map the outputs to the controlled variables, represented by relation OUT. (Here we have made the decision to align IN and OUT with elements of the embedding system.) REQ gives the required behaviour in "real world" terms (environment and platform); SOFTREQ is the analogous specification at the level of computing system and software. A control interface is also shown; this would be a cockpit interface if the platform were an engine. The interface can be thought of as a further set of monitored, controlled, input and output variables, albeit with a very different inter-relationships determined by the design of other systems on the aircraft.

In problem frames, the domain models would encompass necessary properties of the environment, platform, the embedding system, the sensors and actuators – NAT (with a wide scope), IN and OUT in Parnas' terms.

[1] Note that by a relation Parnas is referring to a *trajectory* or *time-indexed* relation between variables.

3 Development Models – First Proposed Enhancement

An important practical consideration regarding domain modelling and the elucidation of NAT, REQ, IN and OUT is how to manage the considerable complexity that may be inherent. From our experience with aerospace applications, as can be seen from the above, there are certain subtleties to be addressed. A key concern is to reflect better the role of the embedding system, and to distinguish it from the environment and platform. Our view is that such distinctions provide a useful basis for abstraction, and that they need to be acknowledged and clarified within the development model. By achieving a greater separation of concerns, we believe it will be easier to develop and validate specifications and to handle change.

A further problem that we need to contend with is the difficulty of sensing key properties of the environment/platform. For example it is not practical to manage engine thrust directly – although it is a key controlled variable – instead it is necessary to use surrogates such as shaft speed or engine pressure ratio.

Thus our first proposal is to enhance the environmental model by adding additional variables. Thus, in addition to monitored/controlled variables and inputs/outputs, we might further distinguish:

- sensed and actuated variables: those real-world variables[2] which are affected directly by the system under development, and which are influenced by/influence the monitored/controlled variables;
- embeddingInput and embeddingOutput variables: those variables which represent the inputs and outputs of the embedding system.

Thus, for instance, whilst REQ might still define the high-level requirements (thrust in terms of demand), we could also distinguish EFFECTREQ over sensed and actuated variables and EMBEDDINGREQ over EmbeddingInput and EmbeddingOutput variables. We would also need to provide the equivalent of the IN/OUT relations to define how the new variables are related. For example, INEmbedding could describe the relationship between the real-world "sensed" variables and the inputs to the embedding system.

We can illustrate the above principle by revisiting the earlier engine example. The monitored variables are demands, temperatures, pressures and shaft speeds; the controlled variable is thrust. The sensed variables are the same as the monitored variables, whereas the actuated variable is fuel flow. The inputs to the embedding system might be analogue electronic signals from several sensing devices (with multiplex redundancy for some of the sensed variables). The output might be control signals to a stepper motor which changes the "throat" on a control valve. Finally, the inputs to the computer are digital representations of the analogue sensor inputs, and the output is a digital representation of the stepper motor signal. The relation INEmbedding in this context would relate the sensed input signals to the real-world variables they are sensing – this might reflect assumptions, for instance, about "noise" and also allow for

[2] E.g. shaft speed.

properties of the (true) environment[3]. It is now possible to state the relationships between the various abstractions:

$$\text{EMBEDDINGREQ} \sqsubseteq \text{IN} \rightarrow \text{SOFTREQ} \rightarrow \text{OUT}$$
$$\text{EFFECTREQ} \sqsubseteq \text{IN}_{\text{Embedding}} \rightarrow \text{EMBEDDINGREQ} \rightarrow \text{OUT}_{\text{Embedding}}$$

Where $\sqsubseteq$ is the appropriate refinement relation, and $\rightarrow$ represents composition of Parnas' relations. The above is a generalisation of the usual relationship between REQ and IN, SOFTREQ and OUT. However, once in the "real world" this generalisation, whilst valid, may be impractical to define as the relationships between sensor/actuator variables and monitored/control variables are likely to be too complex to represent as IN/OUT style relations between interface[4] variables (c.f. closed-loop control). Instead we would propose the following:

NAT is defined as a relation over all monitored/controlled and sensed/actuated variables, representing a model of the *real world*.

REQ is defined as a relation over monitored and controlled variables, with the condition that:

$$\text{NAT} \setminus ((\text{sensed} \cup \text{actuated}) \setminus (\text{monitored} \cup \text{controlled})) \sqsubseteq \text{REQ}$$

i.e. that REQ is consistent with NAT (where all sensed/actuated variables that are not also monitored/controlled variables have been hidden).

EMBEDDINGREQ is defined as a relation over sensed and actuated variables, with the condition that:

$$(\text{NAT} \wedge \text{REQ}) \setminus ((\text{monitored} \cup \text{controlled}) \setminus (\text{sensed} \cup \text{actuated})) \sqsubseteq \text{EFFECTREQ}$$

i.e. that EFFECTREQ is consistent with both NAT and REQ (where all monitored/controlled variables that are not also sensed/actuated variables have been hidden).

Finally, although we have distinguished *an* embedding system, for certain applications there may be a *hierarchy* of embedding systems. Thus, it may be desirable to distinguish more than one set of embedding system variables and requirements etc. We presented the "simple" case as an example of the general case.

4 Development Models – Second Proposed Enhancement

SOFTREQ is expressed rather monolithically. In fact there will be computing hardware, application software and also other software elements, e.g. an operating system, functions for managing faults, etc. Our second proposal is to elaborate the four variable model as shown in Fig 2:

---

[3] For example, engine pressure ratio may be modified by air temperature to produce a good model of the achieved thrust.

[4] I.e. between monitored and sensed, and between actuated and controlled.

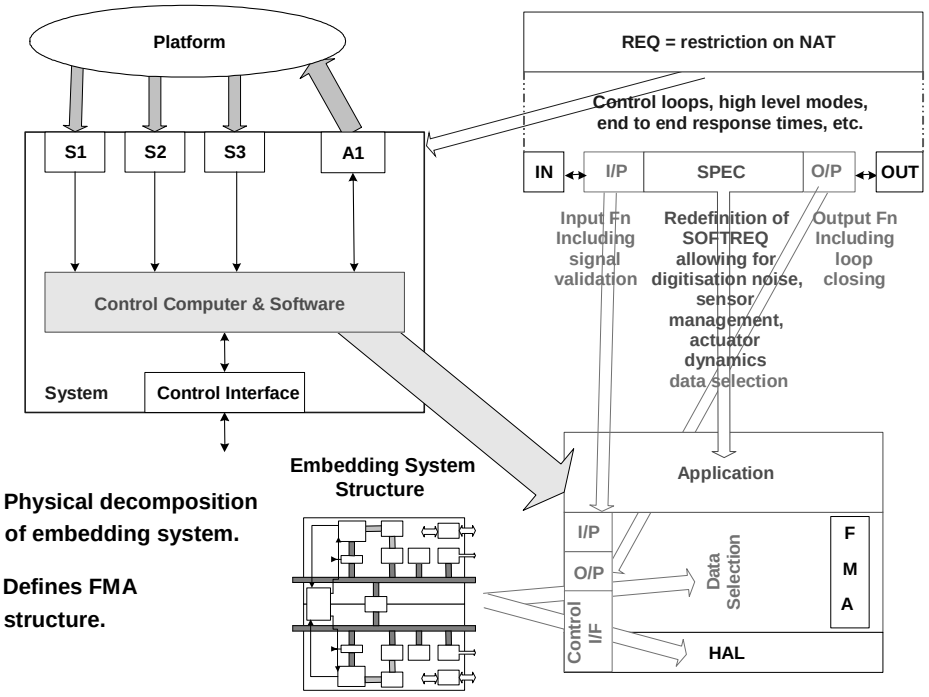

Fig. 2. Representation of Software Structure

This expanded model shows further decomposition of the software specifications, reflecting the hardware structure of the embedding system. The control system software will include device drivers (represented as I/P and O/P) which will map the output of the sensors to meaningful values in software, e.g. the output of a 6 bit A/D converter to a temperature in degrees C, represented as an Ada variable; similarly O/P represents drivers for actuators (note these may be complex and read back values from actuators, running them "closed loop"). A hardware abstraction layer (HAL), or primitive operating system, provides basic services such as scheduling, timers, etc.

The controller computer hardware is usually multiplex redundant, and there are often multiple sources of sensed data. Thus there is fault management and accommodation (FMA), or data selection, logic deriving "healthy" values from the various inputs to provide validated data to the application. A "disconnect" is shown between IN and I/P, and O/P and OUT to reflect that input/output variables may correspond to different embedding system inputs and outputs depending on which input values are selected. In highly critical applications, e.g. aircraft flight control, the validation and data selection logic (dealing with redundant processing hardware, sensors and actuators) might account for 80% or more of the embedded code.

In problem frame terms, the controller structure is another (part of the) domain model. There is another important factor in such a development, the introduction of a software architecture to structure the code. Jackson has been developing problem frames in this direction [7]; this is important, but for brevity we focus here on more "black

box" specifications. Finally the definition of HAL seems to be a "free choice"; in practice the application-programming interface (API) is likely to be defined by a standard, e.g. ARINC653 [8].

5 Challenges for Formal Development

At one level the challenge for formal development is stated simply; provide a formal process which:

- acknowledges the structure of the environment (cf. sensed/actuated, EmbeddingInput/EmbeddingOutput variables);
- respects and supports the relationship of physical design decomposition and specifications outlined in Fig 2.

It is possible to illustrate the challenge by considering some of the general philosophies of formal software development:

- refinement – the development process sits uneasily with the usual rules of refinement, e.g. weakening pre-conditions and strengthening post-conditions. For example, requirements will be met under normal conditions and under certain classes of input failure, but will be violated when inadequate input data is available. The important thing to note is that the precondition representing "inadequate input data" can not easily be expressed over the program variables; it is a "real world" property. Thus, without adequate treatment of the problem structure external to the software, one might have no recourse but to weaken the post-condition in this situation . From a development process perspective, one abstract data value (monitored), e.g. air pressure, may have multiple representations at different points in the environment and software – "real-world", "raw" values from sensors, value after fault accommodation for that sensor, value after voting between alternative data sources or derivation from other sensors etc. These "design steps" are not supported by the classical rules of refinement;
- continuous (e.g. closed loop) control – most embedded safety-critical systems use some form of continuous control, at least for part of the system. Thus the software is required to implement discrete approximations to continuous transfer functions, transforming not just the values of interest but also their integrals and differentials. The control engineers are interested in properties such as jitter, stability, etc. The issue for formal development is linking the discrete specification (e.g. SPEC) back to the continuous requirement, i.e. REQ;
- abstraction – it is hard to employ abstraction. The data being manipulated is a simple reflection of real-world properties, e.g. temperatures and pressures, making classical data abstraction of little value. Other approaches, e.g. loose or algebraic specifications, are also of limited relevance – it is necessary to specify precisely what happens under all physically permissible circumstances to

ensure safety, and so on. Some state abstraction is possible, but abstraction is a "much weaker tool" for embedded systems than in more classical "IT" systems;

- non-functional properties – the non-functional properties, e.g. timing, numerical accuracy (to ensure stability of control algorithms) etc. are crucial aspects of "correctness". Further, the functional and non-functional properties are not always separable. For instance, the functional requirements for fault detection will depend on timing requirements (e.g. the larger the interval between one reading and the next, the wider the error bands which have to be set on valid inputs).

There are some proposed approaches to these problems, e.g. the work on retrenchment [9] and some direct approaches to deriving control system specifications [10], but none of these address the range of problems outlined above.

6 Models for Safety-Critical Formal Development

Producing a formal development process which fully addressed all of the issues outlined above would be an enormous undertaking – and also, we would argue, unhelpful. To address the above issues within a single formal notation it would be necessary to formalise the relevant aspects of physics, including atmospheric and oceanic models for aircraft and ships, respectively, thermal properties of materials, e.g. fuels, sensor dynamics, and so on. Clearly this is not practical – and, in any event, there are well-established approaches for dealing with such issues in engineering practice. No one (formal) technique can adequately incorporate all of the essential features of the control system. For example, embedding *control theory* into a discrete formal method is impractical (if not impossible), and one cannot rigorously analyse discrete software elements in control theory.

Thus it seems that the strategy that should be adopted is to formalise "where formality adds engineering value", and make the links between formal development of software and the relevant aspects of domain models external to the formalism itself. However, this approach yields a secondary *meta-modelling* problem.

In many cases different dimensions to the problem space can be "separated" and targeted by different forms of analysis, e.g. the concerted use of control theory, formal specification and refinement, numerical analysis, scheduling theory and probabilistic/risk-based analysis. However, it is vital that the relationships between the various techniques are properly understood, in order for example to ensure their mutual consistency. The *meta-modelling* problem is crucial to the successful application of formal methods.

In the approach we have been developing, known as Practical Formal Specification (PFS), the interpretation of "where formality adds engineering value" has been to record assumptions which reflect the key parts of the domain models, and to conduct validation of the specifications in the context of these assumptions. The term assumption is used because these are properties which have to be assumed by the software

developers, and which cannot be "proven" as part of the (pure) software development process. These assumptions often reflect properties of the embedding system or platform, e.g. maximum rate of change of temperature (given thermal mass, and software iteration rate). Thus the assumptions bring relevant parts of relations such as $IN_{Embedding}$ into the realm of formal analysis.

7 Progress on Practical Formal Development

Our work on PFS initially started as a general analysis of where formality can add most value; more recent work has centred on Matlab/ Simulink/Stateflow (MSS), the development tool suite widely used by control systems engineers in industry. Our aim has been to add formalism to MSS specifications in a non-obtrusive manner so that the approach can be used by those already familiar with the tool without the need for substantial retraining. There are three core elements to PFS [11]:

- notational restrictions to ensure that sound specifications are produced (MSS is, in effect, a graphical programming language and it is possible to write very poor specifications in MSS);
- representation of assumptions about the domain through annotations on the MSS specifications, e.g. on the states of Stateflow (state machine) diagrams, representing the maximum rate of change of the model variables. These can be proven consistent with assumptions on the root state, which are in turn re-written (in weakest-precondition style) into assumptions on the domain variables (these need to be validated with domain experts, and can not be further analysed formally);
- rules for "healthiness" of specifications, e.g. disjointedness/completeness of transition triggers, self-consistency of specification and assumptions. These rules are checked by an analysis tool known as SSA [12] (Simulink/ Stateflow Analyser) which extracts a semantic model of the MSS specification, a representation of the assumptions and generates proof obligations for the healthiness conditions. These conditions are discharged formally using a combination of automated proof and model checking.

The PFS approach and SSA tool are influenced by the development models outlined above, and are intended to be a step towards resolving the identified challenges for formal development. In terms of these challenges, progress (within PFS and SSA) is as follows:

- refinement – the approach allows for the sorts of development steps outlined above, especially stating and relating assumptions at different levels and checking healthiness properties of the specifications. The relationships between the assumptions at different levels are checked, but the rules are probably too strict making engineering practicalities, such as requirements concessions (for example to deal with loss of sensor data), difficult to handle. There is an issue here regarding the right balance between formally justifying model assumptions and relying on validation by other means;

- continuous (e.g. closed loop) control – as stated earlier, a controller represents a *transfer* function, which maps inputs plus their differentials and integrals into outputs plus their differentials and integrals. Crucially, the PFS weakest precondition analysis is based on (discrete representations of) differential pre- and post-conditions, which allows us to do meaningful analysis of the transfer function properties of the model. Currently, we can analyse, for example, simple assumptions which can be justified in terms of transfer function behaviour (e.g. a differentiator or integrator). However, the analysis is never going to be a substitute for control theory, and again there is a trade-off to be made between formalisation and other forms of validation;
- abstraction – PFS and SSA support as much relevant abstraction as possible. Loose definition of sub-system behaviour allows for a compositional approach. Loose definitions are especially important for abstracting away from the details of continuous functions – such as those derived empirically from the domain. Such sub-systems can at one level be described relationally – as function *envelopes* – carrying enough information to ensure consistency with other parts of the model. In addition *rate of change* assumptions are particularly useful for abstraction in requirements modelled as state-machines. The assumptions effectively *scope* the set of circumstances to which the state-machine must react;
- non-functional properties – these are largely outside the scope of the method and toolset at present. There are good tools for dealing with timing properties, but this does not address the meta-modelling issue of their integration into the overall formal development process.

PFS and SSA focus on validation of specifications; they are complementary to approaches, such as ClawZ [13], which focus primarily on the verification of code against Simulink models.

8 Residual Research Challenges

There is much to be done towards an effective model of formal development; the PFS/SSA approach addresses only some of the issues identified. There remain many open problems, both at the modelling and meta-modelling levels, including:

- adequate treatment of control laws, i.e. validation of important control properties such as stability. Note that this requires addressing timing properties within the formal models;
- review of current restrictions on the approach with a view to enabling a wider class of specifications to be addressed (without imposing any unnecessary constraints on control engineers);
- dealing with non-functional properties in specifications;

- providing stronger links with the safety[5] process, including effective treatment of failure management code – for example, by using a concession-like approach such as "otherwise clauses", or by auto-generating fault accommodation code from safety analysis results, e.g. failure modes and effects analyses.

Some of these issues are being addressed by projects of which the authors are aware (many outside York); producing an integrated and usable approach remains a major challenge.

9 Conclusions

Development of safety-critical software is, in many respects, a "natural" domain for application of formal methods. Despite the dictates of standards and some successes, the use of "classical" formal techniques on embedded safety-critical code remains the exception, not the rule. This paper has tried to articulate the technical (as opposed to commercial and cultural) reasons for the limited use of formal methods on safety-critical software, and outlined some of the characteristics which a formal development process would need to have to be useful in such a domain.

We have outlined some of our work which addresses part of this broad challenge – but acknowledge that there is much to be done, and many other "pieces of the jigsaw" which need to be put in place to provide a fully fledged formal development process for safety-critical software. It is hoped that, by articulating the vision for such a process, it will help foster a better understanding of the technical challenges which need to be met in this area, and thus stimulate constructive and collaborative work on the issues.

10 Acknowledgements

The ideas presented here have been influenced by discussions with colleagues at York and elsewhere. Discussions with Dave Parnas and Michael Jackson over a number of years have been particularly influential.

We acknowledge the support of the UK MoD for the PFS project, and of the EPSRC through MATISSE (GR/R70590/01), for some of the more detailed work which underpins the philosophy set out above.

We are also grateful to our industrial collaborators, especially Airbus, BAE Systems, QinetiQ and Rolls-Royce. Without them we would not have gained the understanding of the practical safety-critical systems development issues outlined here.

[5] Including probabilistic and risk-based analyses.

References

1. UK Ministry of Defence, *Defence Standard 00-56 Issue 2: Safety Management Requirements for Defence Systems*. 1996.
2. Australian Department of Defence, *Australian Defence Standard Def(Aust) 5679: Procurement of Computer-based Safety Critical Systems*. 1998.
3. A German, Software Static Code Analysis, Lessons Learned, Crosstalk, November 2003.
4. S King, J Hammond, R Chapman, A Pryor, Is Proof more Cost-Effective than Tesing?, IEEE Transactions on Software Engineering, Vol. 26, No. 8, 2000.
5. D Parnas, J Madey, Functional Documents for Computer Programs, Science of Computer Programming, Vol. 25, No. 1, 1995.
6. M A Jackson, Problem Frames, Addison Wesley, 2001.
7. L Rapanotti, J G Hall, M A Jackson, B Nuseibeh, Architecture-driven Problem Decompositoin, in Proceedings of RE04, IEEE Computer Society Press, 2004
8. Supplement 1 to ARINC Specification 653: Avionics Application Software Stadard Interface, Standard 03-116/SWM-89, Airline Electronic Engineering Committee, ARINC, Annapolis Maryland, 2003.
9. M Poppleton, R Banach, Retrenchment, Refinement and Simulation, in Proceedings of ZB 2000, J P Bowen, S Dunne, A Galloway, S King (Eds), LNCS 19878, Spinger Verlag, 2000.
10. I Hayes, M Jackson, C B Jones, Determining the Specification of a Control System from that of its Environment, In Proceedings of FME 03, K Araki, S Gnesi, D Mandioli (Eds), LNCS 2805, Springer Verlag, 2003.
11. F Iwu, A Galloway, I Toyn and J A McDermid, Practical Formal Specification For Embedded Control Systems, in the 11th IFAC Symposium on Information Control Problems in Manufacturing, INCOM 2004 Salvador, Brazil April 5-7, 2004.
12. A Galloway, I Toyn, F Iwu and J A McDermid, The Simulink/Stateflow Analyzer, FAA and Embry-Riddle Aeronautical University (ERAU) Software Tools Workshop, Florida USA May 18 -19, 2004.
13. R Arthan, P Caseley, C O'Halloran and A Smith, ClawZ: Control Laws in Z, In Proceedings of ICFEM 2000, S Liu, J A McDermid, M G Hinchey (Eds), IEEE Computer Society, 2000.