Easy approach to requirements syntax (EARS)

Conference Paper · October 2009

DOI: 10.1109/RE.2009.9 · Source: IEEE Xplore

4 authors, including:

Alistair Mavin
Rolls-Royce
21 PUBLICATIONS **120** CITATIONS
SEE PROFILE

Philip Wilkinson
Rolls-Royce
12 PUBLICATIONS **78** CITATIONS
SEE PROFILE

Adrian R. G. Harwood
The University of Manchester
10 PUBLICATIONS **31** CITATIONS
SEE PROFILE

Accepted for RE09, IEEE, August 2009

EARS (Easy Approach to Requirements Syntax)

Alistair Mavin Philip Wilkinson Adrian Harwood Mark Novak

Rolls-Royce PLC
PO Box 31
Derby DE24 8BJ, UK
alistair.mavin@rolls-royce.com

Abstract

The development of complex systems frequently involves extensive work to elicit, document and review stakeholder requirements. Stakeholder requirements are usually written in unconstrained natural language, which is inherently imprecise. During system development, problems in stakeholder requirements inevitably propagate to lower levels. This creates unnecessary volatility and risk, which impact programme schedule and cost. Some experts advocate the use of other notations to increase precision and minimise problems such as ambiguity. However, use of non-textual notations requires translation of the source requirements, which can introduce further errors. There is also a training overhead associated with the introduction of new notations. A small set of structural rules was developed to address eight common requirement problems including ambiguity, complexity and vagueness. The ruleset allows all natural language requirements to be expressed in one of five simple templates. The ruleset was applied whilst extracting aero engine control system requirements from an airworthiness regulation document. The results of this case study show qualitative and quantitative improvements compared with a conventional textual requirements specification.

1. Company background

Aircraft engine control systems present a significant systems engineering challenge: the systems are complex, safety-critical and developed in ever-compressed timescales. To satisfy aircraft manufacturers' requirements and maintain market position, control systems must provide increased functionality and maintain the highest levels of dependability.

Rolls-Royce Control Systems develops Full Authority Digital Engine Controllers (FADECs) for civil gas turbine engines. The development and operation of FADECs present numerous challenges [1] including: operation in arduous environments, increased system complexity, ever-greater reliability, improved fault tolerance and the need to certify against European and US regulations. A typical modern control system has a dual channel design, contains thousands of components, over 100,000 lines of code and is developed with up to twenty suppliers.

2. Background to the problem

Stakeholder requirements are often written by individuals who are not experts in requirements definition. Most commonly, stakeholder requirements are written in unstructured natural language (NL). The flexibility of NL makes it an ideal medium for creative expression, such as drama, poetry and humour. However, unconstrained use of NL is inherently unsuitable for requirements definition for a number of reasons. Some of the problems that can appear in NL requirement documents are:

- Ambiguity[1] (a word or phrase has two or more different meanings).
- Vagueness (lack of precision, structure and/or detail).
- Complexity (compound requirements containing complex sub-clauses and/or several interrelated statements).
- Omission (missing requirements, particularly requirements to handle unwanted behaviour).
- Duplication (repetition of requirements that are defining the same need).
- Wordiness (use of an unnecessary number of words).
- Inappropriate implementation (statements of how the system should be built, rather than what it should do).
- Untestability (requirements that cannot be proven true or false when the system is implemented).

There are other problems that this study does not consider, including conflicting requirements and missing traceability links. There are two main reasons for their exclusion: firstly these problems are not unique to NL requirements documents, and secondly there are no occurrences of these problems in the case study.

[1] There are three common forms of ambiguity: lexical, referential and syntactical [2]. Lexical ambiguity occurs where a word or phrase, which has two or more meanings, is used in a manner that permits a sentence or phrase to be interpreted in more than one way. Referential ambiguity occurs when a word or phrase is used that can be referring to two or more things. Syntactical ambiguity arises when the order of words allows two or more interpretations.

To overcome problems associated with NL, some experts advocate the use of other notations for the specification of requirements. These include formal notations such a Z [3] and Petri Nets [4] and graphical notations such as Unified Modeling Language (UML) [5, 6] and Systems Modeling Language (SysML) [7]. There are also numerous scenario-based approaches [summarised in 8], tabular approaches such as Table-Driven Requirements [9] and ConCERT [10, 11] and pseudocode.

Advocates of some notations claim that they work for requirements at all system levels, whilst others do not claim universal applicability. UML and SysML are graphical notations used to describe systems, within which different views can be generated depending on user needs. Meanwhile, claims have been made for the effective use of scenarios in many different ways and at different system levels [8] though not necessarily within an integrated framework.

However, use of any of these non-textual notations often requires complex translation of the source requirements, which can introduce further errors. Such translation of requirements can serve to create a "language barrier" between developers and stakeholders due to unconscious communication of assumed context. There is also a training overhead associated with the introduction of many notations. Requirements authors are unlikely to seek excessive formality and complex training, and rarely do they require a software tool to help them write.

There are many hundreds of general books on requirements engineering. There are also numerous examples of published works specifically about how to write better requirements. These include two well-known papers titled "Writing Good Requirements" [12, 13] that focus on the characteristics of well-formed requirements and the attributes that should be included. There are also templates available, such as VOLERE [14] and SL-07 [15]. Despite this large body of published material, there seems to be little simple, practical advice for the practitioner.

It was hypothesised that introducing a small set of simple requirement structures would be an efficient and practical way to enhance the writing of high-level stakeholder requirements. Previous work in the area of constrained natural language includes Simplified Technical English [16], Attempto Controlled English (ACE) [17] and Event-Condition-Action (ECA) [18]. In ECA, the *event* specifies the signal that triggers the rule and the *condition* is a logical test that (if satisfied) causes the specified system *action*.

The work reported here is principally concerned with requirements syntax. Although measures were taken to improve the semantics of the requirements, they are not described in this paper. There is no claim made that this approach is universally applicable to all levels of system decomposition. The technique is most suitable to the definition of high-level stakeholder requirements.

The remainder of this paper is divided into five sections. Section 3 describes the Case Study. Section 4 defines the Method used and how it was developed. Section 5 summarises the Results. Section 6 is a Discussion of the findings and section 7 describes Future Work.

3. Case study

Certification Specification for Engines (CS-E) [19] defines what must be achieved in order for an aero engine to achieve certification. For this study, the section of CS-E most applicable to engine control systems (CS-E 50) was analysed. CS-E 50 contains a relatively small number of requirements, which was a manageable quantity for a case study.

When analysing the structure and content of CS-E, it is prudent to consider the history of the document. CS-E has evolved from incremental updates to its predecessor Joint Airworthiness Requirements for Engines (JAR-E) [20] over many years. Repeated updates have resulted in the addition of statements to form long paragraphs of prose. Many of these paragraphs contain a rich mixture of both complex and simple requirements, along with design and verification statements and supporting information. Most requirements are explicit, but there are also some implicit requirements that can be difficult to discover.

Due to the evolutionary nature of the text, care must be taken when interpreting the intent of statements within CS-E. Much of the document is written at an abstract level, relying on lists and explanatory notes to add meaning. Engineers use the requirements in CS-E during the design of an engine, but most are unlikely to be trained and experienced in requirements definition. Engineers are likely to be most comfortable when presented with a set of unambiguous, simple statements in order to validate their own work. This will reduce the likelihood of lengthy negotiations or expensive alterations later in an engine programme. In addition, if requirements are written well, they can be reused on future programmes, with obvious cost savings.

4. Method

The process and ruleset used during this investigation was developed as the work progressed. The participants started with a set of loose rules derived from their own experiences in systems, safety and requirements engineering and built on the concepts of ECA. These included basic syntactic rules of thumb such as the use of "when" for event-driven behaviour, "while" for state-driven behaviour and "if-then" statements to handle "failures".

The collaborative nature of the work led to incremental changes to the ruleset that were tested empirically during the ongoing study. Additionally, occasional serendipity helped the evolution of the syntactic rules.

A group of cross-discipline engineers, including the company's CS-E certification expert, analysed the source text of CS-E 50 in two phases. In the first phase, each

clause of the document was broken into its constituent parts. Some sentences were explicit requirements; others needed interpretation, but did imply requirements on the engine control system. Other statements were determined to be design guidance or were clearly informative text. All requirements were placed in a spreadsheet to aid manipulation.

In the second phase, each requirement was rewritten in a consistent manner using the syntactic ruleset described below. Subsequent iterations were necessary as the ruleset evolved. This led to further rewording of requirements and to the reclassification of some statements.

4.1 Generic requirements syntax

The generic requirement syntax adopted during this study is as follows:

<optional preconditions> <optional trigger> the <system name> shall <system response>

This simple structure forces the requirement author to separate the conditions in which the requirement can be invoked (preconditions), the event that initiates the requirement (trigger) and the necessary system behaviour (system response). Both preconditions and trigger are optional, depending on the requirement type, as described later in this section.

The order of the clauses in this syntax is also significant, since it follows temporal logic:

- Any preconditions must be satisfied otherwise the requirement cannot ever be activated.
- The trigger must be true for the requirement to be "fired", but only if the preconditions were already satisfied.
- The system is required to achieve the stated system response if and only if the preconditions and trigger are true.

The generic requirement syntax is specialised into five types of requirement (Ubiquitous, Event-driven, Unwanted behaviours, State-driven and Optional features), which are described in more detail below.

4.2 Ubiquitous requirements

A *ubiquitous* requirement has no preconditions or trigger. It is not invoked by an event detected at the system boundary or in response to a defined system state, but is always active. The general form of a ubiquitous requirement is:

The <system name> shall <system response>

For example: *"The control system shall prevent engine overspeed".* This is a system behaviour that must be active at all times; hence this is a ubiquitous requirement.

4.3 Event-driven requirements

An *event-driven* requirement is initiated only when a triggering event is detected at the system boundary. The keyword *When* is used for event-driven requirements. The general form of an event-driven requirement is:

WHEN <optional preconditions> <trigger> the <system name> shall <system response>

For example: *"When continuous ignition is commanded by the aircraft, the control system shall switch on continuous ignition".* This system response is required when and only when the stated event is detected at the boundary of the system.

4.4 Unwanted behaviours

Requirements to handle *unwanted behaviour*[2] are defined using a syntax derived from event-driven requirements. "Unwanted behaviour" is a general term used to cover all situations that are undesirable. This includes failures, disturbances, deviations from desired user behaviour and any unexpected behaviour of interacting systems. The authors' experiences suggest that unwanted behaviour is a major source of omissions in early requirements, necessitating costly rework. Consequently, these requirements were given their own syntax, so that they could be easily identified throughout the lifecycle.

Requirements for unwanted behaviour are designated using the keywords *If* and *Then*. The general form of a requirement for unwanted behaviour is:

IF <optional preconditions> <trigger>, THEN the <system name> shall <system response>

For example *"If the computed airspeed fault flag is set, then the control system shall use modelled airspeed".* In this example, the unwanted event (computed air speed fault flag is set) triggers the system response, which allows continued safe operation of the system.

Using the *If-Then* structure explicitly differentiates the requirements that handle unwanted behaviour. In such requirements the system response mitigates the impact of the unwanted event, or prevents the system from entering an unwanted state.

[2] The distinction between wanted and unwanted behaviour is not always clear. For example, due to the safety-critical nature of aero engine control systems, many subsystems employ multiple redundant components. This allows the system to accommodate unwanted events whilst continuing to satisfy operational requirements. In such cases, the system is behaving "normally", but the requirements would be considered as describing "unwanted behaviours" using the classification described here. Hence the distinction between wanted and unwanted behaviour is a matter of viewpoint, or even a matter of "style". Another perspective on the distinction between wanted and unwanted behaviours is provided by the concept of Misuse Cases [21]. Misuse Cases describe users with hostile intent who are likely to have wants that are in direct opposition to the wants of other system stakeholders.

4.5 State-driven requirements

A *state-driven* requirement is active while the system is in a defined state. The keyword *While* is used to denote state-driven requirements. The general form of a state-driven requirement is:

WHILE <in a specific state> the <system name> shall <system response>

For example: *"While the aircraft is in-flight, the control system shall maintain engine fuel flow above XXlbs/sec".* The system response is required at all times whilst the system is in the defined state.

To make requirements easier to read, the keyword *During* can be used instead of *While* for state-driven requirements. For example: *"During thrust reverser door translation, the control system shall limit thrust to minimum idle".* In this context, the meaning of *During* is identical to *While*, and this alternative keyword is used purely to aid readability.

4.6 Optional features

An *optional feature* requirement is applicable only in systems that include a particular feature. An optional feature requirement is designated with the keyword *Where*. The general form of an optional feature requirement is:

WHERE <feature is included> the <system name> shall <system response>

For example, *"Where the control system includes an overspeed protection function, the control system shall test the availability of the overspeed protection function prior to aircraft dispatch".* This functionality only makes sense (and therefore is only required) for a system that includes the specified feature.

4.7 Complex requirement syntax

For requirements with complex conditional clauses, combinations of the keywords *When*, *While* and *Where* may be required. The keywords can be built into more complex expressions to specify richer system behaviours. For instance, the same event may trigger different system behaviour depending on the state of the system when the event is detected.

For example: *"While the aircraft is on-ground, when reverse thrust is commanded, the control system shall enable deployment of the thrust reverser".* The triggering event (reverse thrust command) triggers the system response only when the system is in a particular state (aircraft on-ground).

The keywords *When*, *While* and *Where* can also be used within *If-Then* statements to handle unwanted behaviour with more complex conditional clauses. For example the requirement to handle thrust reverser commands during the in-flight state (an unwanted and potentially catastrophic event) is handled as follows: *"While the aircraft is in-flight, if reverse thrust is commanded, then the control system shall inhibit thrust reverser deployment"*. In this situation the trigger (reverse thrust command) is unwanted whilst in-flight and the required system response prevents the system from entering an unwanted state.

Similarly, in the requirement *"When selecting idle setting, if aircraft data is unavailable, then the control system shall select Approach Idle",* the unwanted behaviour (aircraft data is unavailable) should result in the stated system response only when the triggering event (selecting idle) is satisfied.

4.8 Testing the hypothesis

The hypothesis was tested against a number of criteria, based on the eight problems associated with NL requirements identified in section 2. For each problem, instances were counted in the both the raw requirements from the CS-E document and the interpretations. The count for each set of requirements was compared to assess the effectiveness of the EARS ruleset.

5. Results

The majority of the requirements could be written in one of the EARS templates with little difficulty. Those that could not were either manipulated to fit the ruleset or the ruleset was evolved to incorporate the additional requirement types.

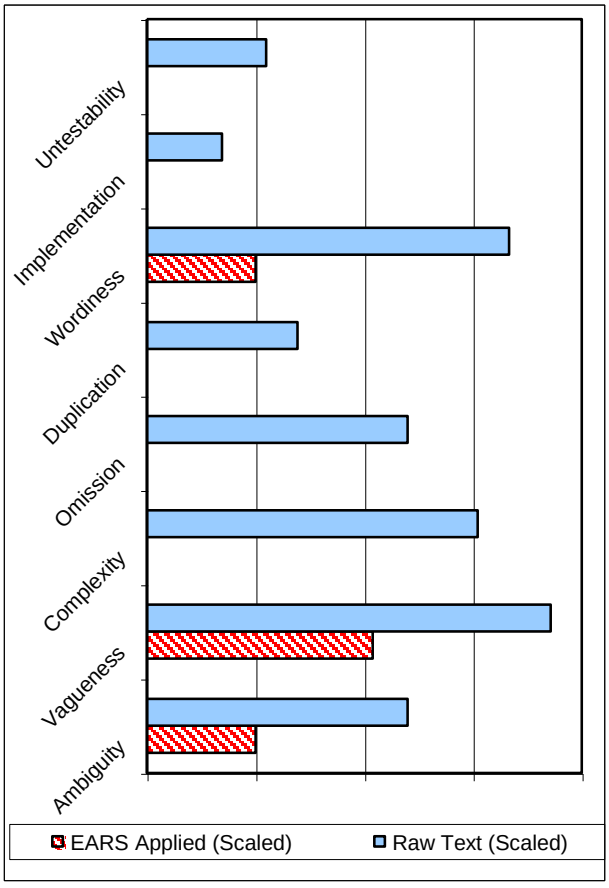
Raw extract of CS-E	Interpretation	Type
It must be substantiated by tests, analysis or a combination thereof that the Engine Control System performs the intended functions in a manner which does not create unacceptable thrust or power oscillations.	The Engine Control System shall not cause unacceptable thrust or power oscillations.	Ubiquitous
It must be demonstrated that, when a Fault or Failure results in a change from one Control Mode to another, or from one channel to another, or from the Primary System to the Back-up System, the change occurs so that the Engine does not exceed any of its operating limitations.	When the Engine Control System changes operational mode, the Engine Control System shall maintain the engine within approved operational limits.	Event Driven
Single Failures leading to loss, interruption or corruption of Aircraft-Supplied Data, must not result in a Hazardous Engine Effect for any Engine.	If a single Failure leads to deficient Aircraft-Supplied Data, then the Engine Control System shall not cause a Hazardous Engine Effect.	Unwanted Behaviour
The Engine Control System must be designed and constructed so that in the Full-up Configuration, the system is essentially single Fault tolerant for electrical and electronic Failures with respect to LOTC/LOPC events.	While in a Full-Up Configuration, the Engine Control System shall be Essentially Single Fault Tolerant with respect to LOTC/LOPC event.	State Driven

Raw extract of CS-E	Interpretation	Type
When over-speed protection is provided through hydro-mechanical means, it must be demonstrated by test or other acceptable means that the over-speed function remains available between inspection and maintenance periods.	Where over-speed protection is provided through hydro-mechanical means, the frequency of Engine Control System inspection and maintenance periods shall be consistent with the required availability of the feature.	Optional Feature

Table 1. Examples of raw extracts from CS-E with interpreted control system requirements

Several patterns emerged as the rules were applied to the raw extracts from CS-E. Common examples of these patterns were:

- Logical restructuring to increase clarity and understanding.
- Reduced wordiness to create simpler statements.
- The separation of complex triggers resulting in two or more atomic requirements.
- Reusable statements and reusable formats.



Graph 1. Quantitative results for case study

Consultation with a team of specialists, including Safety Engineers and Airworthiness Engineers, uncovered the contextual intent of CS-E, allowing an accurate set of requirements to be defined. Table 1 shows some examples of raw extracts from CS-E and the resulting interpretations. As part of the interpretation process, it was occasionally necessary to examine associated advisory material [22]. This material contains information, verification and design statements in addition to requirements.

Following translation of the requirements, the raw requirements and interpretations were compared against the criteria described in section 2. Graph 1 shows the differences for the eight criteria on a log scale[3]. Overall, the interpretations scored significantly better than the raw requirements. Against five of the criteria, the interpretations contain none of the problems present in the raw requirements. For the other three criteria – ambiguity, vagueness and wordiness – a substantial reduction in problems was observed. The number of requirements and the average words per requirement for the raw requirements and the interpretations is shown in Table 2. The total number of requirements has increased, while the average number of words per requirements has reduced.

	Raw extract	Interpretation
No. individual requirements	36.0	47.0
Av. words per requirement	36.9	25.6

Table 2. Word count of raw requirement extracts and interpretations

6. Discussion

The results show that the modified notation has a number of advantages over the use of unconstrained NL. All of the text from the regulation, once formatted as requirements, was successfully translated into one of the EARS templates. Where problems of translation were initially experienced, these were addressed by further evolutionary development of the ruleset.

The review against the criteria of section 2 demonstrated a significant reduction in all eight problem types. For the case study, the notation appears to have eliminated the problems of complexity, omission, duplication, implementation and untestability. However, the claim that omissions have been eliminated needs to be treated with caution. Whilst the trial may have effectively identified some unwanted behaviour, there is no evidence that other missing requirements have been captured.

The problems of ambiguity, vagueness and wordiness were reduced, but not eliminated. The remaining problems are thought to be due to:

- Lexical ambiguity, where a precondition was understood by inference, but not explicitly recorded.

[3] The data contained a combination of large and small data values. It was therefore necessary to scale the numbers using a log function (on values greater than 1) for clarity on the chart.

- General vagueness, which is an innate feature of high-level requirements and difficult to remove until accompanying design decisions are made.
- Wordiness occurred where a particularly long requirement, containing numerous conditional clauses, employed clumsy word constructions.

The increase in the number of requirements was expected. This was because some clauses of CS-E contain numerous compound requirements, which were separated out during the interpretation process. A reduction in the average word-count of the interpretations was also anticipated, and is mainly due to reduced complexity, duplication and wordiness.

Although the results of this case study are very positive, a number of limitations could have influenced the outcomes of the study:

- The sample size was small, consisting of only 36 original requirements.
- The study was restricted to high-level safety-related requirements. It is possible that other types of high-level requirements and lower-level requirements may exist that cannot be adequately represented in the notation.
- Although efforts were made to reduce subjective influences, the classification of vagueness is inevitably a matter of personal opinion and therefore open to inconsistency.

Despite these limitations, the study seems to provide ample evidence to support the hypothesis: that a small set of simple requirement structures would be an efficient and practical way to enhance the writing of high-level stakeholder requirements.

7. Future work

The authors intend to continue with their assessment of the remainder of CS-E. This would involve the translation of all requirements pertaining to the engine control system. Further studies are also planned to assess the notation against an entire suite of high-level engine control system requirements and lower-level elements of the control system design. It is hoped that these studies will establish the effectiveness of the notation by addressing the limitations identified in section 6.

As stated in section 2, a number of known requirements problems were consciously excluded from this study. One of the reasons for this was that the document used in the case study did not include such problems. However, it is expected that use of the EARS templates will address at least some of these issues when applied to other stakeholder requirement documents. For example, applying the syntactic rules will clarify the precise preconditions and triggers, which should highlight conflicting requirements. The intention is to apply the EARS system to a wider range of system documents which contain such problems.

Further thought is necessary to investigate how the *If* and *When* preconditions can be applied to unwanted states. Although the case study identified no such scenarios, unwanted states may require an additional template to define how the system should behave while in an unwanted state.

References

1. Holt J.E., "The Challenges of Civil Aircraft Engine Control System Development at Rolls-Royce", Proceedings of IEEE United Kingdom Automatic Control Council (UKACC) International Control Conference, Glasgow, August, 2006.
2. Warburton, N., "Thinking from A to Z", 2nd edition, Routledge, 2000.
3. Woodcock, J. and Davies, J., "Using Z-Specification, Refinement and Proof", Prentice Hall, 1996.
4. Peterson, J. "Petri Nets", ACM Computing Surveys 9 (1977), 223-252.
5. Object Management Group, UML Resource Page, http://www.uml.org/
6. Holt, J., "UML for Systems Engineering: Watching The Wheels" (2nd edition), IEE, 2004.
7. Object Management Group, Official OMG SysML Site, http://www.omgsysml.org/
8. Alexander, I.F. & Maiden, N.A.M. (eds), "Scenarios, Stories, Use Cases: Through the Systems Development Life-Cycle" Wiley, 2004.
9. Alexander, I.F. & Beus-Dukic, L., "Discovering Requirements", John Wiley, 2009.
10. Vickers, A., Tongue, P.J. and Smith, J.E., "Complexity and its Management in Requirements Engineering", INCOSE UK Annual Symposium – Getting to Grips with Complexity, Coventry, UK, 1996.
11. Vickers, A., Smith, J.E., Tongue, P.J. and Lam, W., "The ConCERT Approach to Requirements Specification (version 2.0)", YUTC/TR/96/01, University of York, November 1996 (enquiries about this report should be addressed to: High-Integrity Systems Engineering Research Group, Department Computer Science, University of York, Heslington, YORK, YO10 5DD, UK).
12. Hooks, I., "Writing Good Requirements", Proceedings of Third International Symposium of INCOSE Volume 2, INCOSE, 1993.
13. Wiegers, K., "Writing Good Requirements", Software Development Magazine, May 1999.
14. "VOLERE Requirements Specification Template", Atlantic Systems Guild, http://www.volere.co.uk/template.htm
15. Lauesen, S., "Guide to Requirements SL-07. Template with Examples", Lauesen Publishing, 2007.
16. "ASD Simplified Technical English: Specification ASD-STE100. International specification for the preparation of maintenance documentation in a controlled language", Simplified Technical English Maintenance Group (STEMG), 2005.
17. Fuchs, N. E., Kaljurand, K. and Schneider, G., "Attempto Controlled English Meets the Challenges of Knowledge Representation, Reasoning, Interoperability and User Interfaces", FLAIRS, 2006.
18. Dittrich, K. R., Gatziu, S. and Geppert, A., "The Active Database Management System Manifesto: A Rulebase of ADBMS Features.", Lecture Notes in Computer Science 985, Springer, 1995, pages 3-20.
19. "Certification Specification for Engines" (CS-E) Amendment 1, 18-20, European Aviation Safety Agency (EASA), 10th December 2007, http://easa.europa.eu/ws_prod/g/rg_certspecs.php
20. "Joint Airworthiness Requirements for Engines" (JAR-E), JAA, http://easa.europa.eu
21. Sindre, G. and Opdahl, A., "Capturing Security Requirements through Misuse Cases", TOOLS 37 (2000), pp. 120-131.
22. "AMC20 Effective 26/12/2007", European Aviation Safety Agency (EASA), http://easa.europa.eu/ws_prod/g/rg_certspecs.php