

# On Systematic Design of Protectors for Employing OTS Items

*Peter Popov<sup>¶</sup>, Steve Riddle<sup>§</sup>, Alexander Romanovsky<sup>§</sup>, and Lorenzo Strigini<sup>¶</sup>*

<sup>¶</sup>Centre for Software Reliability  
City University  
Northampton Square  
London EC1V 0HB  
United Kingdom

<sup>§</sup>Centre for Software Reliability  
University of Newcastle upon Tyne  
Newcastle upon Tyne  
NE1 7RU  
United Kingdom

## Abstract

*Off-the-shelf (OTS) components are increasingly used in application areas with stringent dependability requirements. Component wrapping is a well known structuring technique used in many areas. We propose a general approach to developing protective wrappers that assist in integrating OTS items with a focus on the overall system dependability. The wrappers are viewed as redundant software used to detect errors or suspicious activity and to execute appropriate recovery when possible; wrapper development is considered as a part of system integration activities. Wrappers are to be rigorously specified and executed at run time as a means of protecting OTS items against faults in the rest of the system, and the system against the OTS item's faults. Possible symptoms of erroneous behaviour to be detected by a protective wrapper, and possible actions to be undertaken in response are listed and discussed. The information required for wrapper development is provided by traceability analysis. Possible approaches to implementing "protectors" in the standard current component technologies are briefly outlined.*

*Keywords: system integration, COTS, dependability, error detection and recovery, wrapping, traceability*

## 1 Introduction

In this paper our focus is on developing new techniques for improving the dependability of a system built out of pre-existing items. Use of off-the-shelf (OTS) software promises reduced development cost but raises some difficult problems. There is a tendency to use OTS components in an increasing number of application areas, many of which have stringent dependability requirements. The main problems in employing OTS items in such areas are that they are not reliable enough (or, there is not

enough evidence to support any reasonable claims on their reliability), they do not have complete or correct specification, and very often they are not used in exactly the context they are intended for. New techniques should be developed that allow designers to systematically integrate OTS items into systems without damaging their dependability or, if possible, improving it.

We use the term "OTS" as a generalisation of the more common "COTS" (Commercial Off-The-Shelf) to refer to any code that is not developed specifically for a new system. This includes legacy code, "non development units", pre-existing software from the same company, GOTS (Government Off-The-Shelf), commercial and non-commercial (e.g. freeware or open-source) software.

Wrapping of OTS items is a promising approach to dealing with many problems in employing OTS software. We propose to introduce wrappers as a means for improving the overall system dependability by protecting system components against each other's faults. We use the word "protectors" to designate this category of wrappers. A number of general techniques have been developed to achieve high dependability by employing redundant software: N-version programming, recovery blocks, self-checking pairs, etc. [4]. We view wrappers as redundant custom-built software to be developed during system integration. Within this approach the pair consisting of an OTS item and its wrapper (protector) is treated as a special fault-tolerant architecture bearing similarities with several of the general architectures. We believe that developing protective wrapper is a complex engineering process that needs general solutions.

Assessing the suitability of OTS items against high dependability requirements raises concerns about the availability and suitability of evidence. Techniques for structuring and analysing safety-related arguments have been developed [7] which employ information models and arguments of traceability to deal with the evidence required and generated during a project's life. These

techniques have also been extended to address the issue of “safe use” of OTS items [2]. In our approach, traceability techniques are used to assist in developing protective wrappers.

## 2 The Basic Architecture

Component wrapping is a well known structuring technique that has been used in several areas. A wrapper is a specialised component inserted between a component and its environment to deal with the flows of control and data going to and/or from the wrapped component. The need for wrapping arises from the fact that it is impossible or expensive to change the components to re-use them as parts of a new system, or it is easier to add new features by incorporating them into wrappers. Wrapping is a structured and a cost-effective solution to many problems in component-based system development. Wrappers are usually employed for improving non-functional properties of the components such as adding caching and buffering, dealing with mismatches or simplifying the component interface. With respect to dependability, wrappers are used for ensuring security, transparent component replication, etc.

Our focus is on developing protective wrappers - protectors - that can improve the overall system dependability by protecting both the system against erroneous behaviour of an OTS item and the OTS item against erroneous requests from the system [11]. This development can be a complex process that requires rigour and discipline and should be integrated into the process of the development of the whole system.

Recent research in the area has addressed some important issues but in our opinion there is a need for systematic general solutions. Paper [10] shows how to build wrappers using results of the testing of the OTS item and of fault injection (at its interface). This allows the wrapper to intercept certain inputs and outputs and make their intended recipients ignore them. Paper [6] discusses how OTS items can be used in safety critical systems and proposes to protect the critical services from them by completely isolating these items from the rest of the system using encapsulation mechanisms (this approach cannot be applied in those systems in which OTS items are involved in delivering all types of services). A very interesting approach to developing protective wrappers for an OTS microkernel is discussed in [9]. The idea is to specify the correct behaviour of a microkernel and to make the protective wrapper check calls the microkernel (note that this approach cannot be applied for OTS items that lack a complete, correct specification). In addition, the results of fault injection are used, in that the wrapper is designed to catch those calls that have been found to cause errors of the particular microkernel implementation to be wrapped. A similar approach is suggested [3] for using the Ballista project’s

results from fault injection on different vendors’ POSIX implementations. We do agree that known bug reports should be used in developing wrappers, because it is often unlikely that the vendors of the OTS item will correct it on request. This cannot solve all problems, however: blocking calls which would trigger known faults (bugs) neither tolerates the fault, nor protects against undiscovered faults. Moreover, the proposed solutions do not take into account many important considerations. A typical assumption is that wrappers themselves are “simple” (at least in relation to the wrapped components) and that their development is a trivial task. This is not always the case: indeed, a wrapper is a piece of software like any other and just as prone to defects as any other software engineering artefact of comparable complexity. Wrappers will often be complex artefacts and requiring them to perform protection functions will make them more complex. Moreover, common failures of protectors and the protected item should be taken into account when developing the wrapper and assessing the overall system dependability. We believe that protective wrappers should be used as a general error detection feature and as a systematic means for attempting to deal with errors as early as possible. To avoid uncontrolled complexity, wrapper development needs a systematic, disciplined approach.

The simplest system model of OTS use consists of: the OTS item to be integrated, the computer system in which it is to be incorporated (we call it the *Rest Of the System* - ROS, so that the computer system will consist of the OTS item and the ROS), and the environment controlled by the computer system. We assume that the OTS item can be accessed via its declared interface only, and the system designers may have no information at all about the internal structure or behaviour of such items (in practice, there is a continuum of cases from “pure black box” situations to “clear box” ones, like open source OTS items. Even the latter pose some of the problems we have to deal with when dealing with black boxes.)

## 3 Design of Protector

### 3.1 Acceptable Behaviour Constraints

There are many reasons why improved protection (i.e. error detection and recovery) is important for system integration [11, 10, 12]: the OTS item’s specification can be incomplete; components might not work as promised by their providers or might have bugs; unspecified (non-standard) features of the component might be used by the ROS (this can, for example, affect compatibility with future versions of the OTS item); the environment (controlled system) may suffer failures that cause non-obvious requests on the OTS item; the ROS might misuse the component or the component can output incorrect

commands to the controlled environment. In addition, depending on the application context, developers might decide not to use some part of the component service, or they might know specific restrictions on input/output parameters. In these cases it may be important to check that these restrictions are satisfied.

To design wrappers/protectors, system developers (i.e. integrators) should "develop" their view on what the OTS component and the ROS do and do not do with respect to each other. We call this the *Acceptable Behaviour Constraints (ABCs)* from the viewpoint of the system developers. In particular, the ABCs may specify the boundaries of situations that can cause violations of the environment's safety, and situations that are "suspicious" – those for which the system designer has insufficient evidence for believing that the system is behaving correctly and is in a state in which it is likely to continue behaving correctly. The ABCs should be developed formally and systematically and the protector implemented to enforce (or at least check) their satisfaction. Developing these constraints should be an important part of the process of integrating OTS items: it may be supported by assembling requirements from several viewpoints (from the perspective of the ROS and that of the OTS item) and using traceability arguments to assess the consistency of these viewpoints. Section 4 provides an overview of this assessment.

Having declared the ABCs, the designer takes the usual precautions to tolerate detected errors (or "suspicious" situations – which might develop into errors) or at least to mitigate their effects. The conventional mechanisms for error detection, containment and recovery and/or failure compensation [4] are employed.

Our approach relies on existing research on executable assertions [8] and on design by contract [5]. There are some obstacles to be considered in the way of applying these ideas to developing protectors, mainly because OTS items may be black boxes and integrators do not generally have their complete, correct specification. Only some types of assertions [8] are applicable in our context: e.g., "consistency between arguments", "dependency of return value on arguments", "frame specifications". However, the examples given in [8] and most other papers on assertions refer to low-level information, in a "programming in the small" context, and we think that more complex, application-specific and whole-system-specific assertions should be included in the ABCs of a component. Design by contract has a different purpose: in

our context the system designer develops a protector using his/her view on a "correct" contract between the ROS and the OTS item, and on their correct behaviour with respect to each other, rather than using a contract explicitly accepted by the OTS item's designer.

The various possible ingredients listed in ABCs are alternative, partial descriptions of overlapping views, not subsets in a universe described in one consistent language. General sources of information that can be used are:

- behaviour specification of OTS items as specified by their designers;
- behaviour specification of an OTS item as specified by system designers. This description and the previous one must satisfy certain mutual constraints for the system design to be correct, but they will not be identical. E.g. the system designer's description requires the OTS item to be able to react to a set of stimuli that is a subset of the set specified by the component designer;
- behaviour that the system designer expects from an OTS item (not necessarily approving it): i.e., he/she may know that that it often fails in response to certain legal stimuli;
- component (OTS or part of ROS) behaviour that system designers considers especially unacceptable, without knowing whether it is likely or not;
- behaviour specifications of the ROS.

The specification of the protector, and especially the knowledge and assumptions on which it is based, should be captured in an analysable (traceable) way, in the same way as it is with the OTS item to be protected. This makes it possible to reuse the protector with a new version of the OTS item (or improve the protector when the integrator's understanding of the system improves) and check that the assumptions made are still valid.

The functionalities of protectors that ABCs have to inform are summarised in the table of cues (Table 1) that a protector can be designed to use to determine that some action is in order. The cues are found in the communication ("messages" in the table) between the OTS and ROS components. These categories are not meant to be mutually exclusive, nor exhaustive; separating them this way is useful for use as a checklist for system designers, and because they differ in the complexity of detecting them, how strong an indication they are of something being wrong, etc.

Table 1

Category of cue	of what possible unpleasant events				detection can be programmed using knowledge about
	error of ROS	error of OTS item	threat for ROS	threat for OTS item	
<b>input to OTS item:</b>					
the message is outside the domain (within the OTS item input space, seen as space of possible histories of inputs) with which system designers intended the OTS item to deal (may indicate error by the ROS, or simply a behaviour foreseen by system designers but intended to be considered as an exception)	Y			Y	system design
the message is outside the domain for which the OTS item has been judged "good enough to be trusted"	Y		Y	Y	OTS item
the message is within a domain with which the OTS item is known to have trouble coping	Y		Y	Y	OTS item
the message is an illegal 'input' for the OTS item (incompatible with the item specs as understood by system designer, irrespective of previous history of interactions)	Y		Y	Y	OTS item
the message is erroneous (incompatible - in view of the previous history of interactions - with ROS specs as understood by system designer) for the ROS to issue	Y		Y	Y	ROS/system
the message is an illegal output for the ROS	Y			Y	ROS
<b>output from OTS item:</b>					
illegal output for the OTS item		Y	Y		OTS item
erroneous output for the OTS item		Y	Y		OTS item
risky for the controlled environment in view of its known safety envelope		Y			ROS
indicative of the OTS item having used functions/parts (of the item itself) that we do not trust (either we lack positive evidence of their being good enough, or we have positive evidence that they are not)		Y	Y		OTS item
risky for computer system internally		Y	Y	Y	part of computer system that is threatened

In addition, designers have to specify what recovery actions the protector should perform if the component violates its ABCs. These actions can be directed in either or both directions: to the OTS item and/or to the ROS. Possible reactions are (the list is not exhaustive, nor are these reactions mutually exclusive):

- report exception/error code to the issuer of the message or to a global executive or diagnostic subsystem or a human operator;
- substitute the message with a "safe" one;
- redirect the message to an alternative destination (e.g. a "trusted", though less sophisticated fall-back implementation of the OTS item's functions);
- perform a simpler version of the OTS item's function (as above), provided by the protector itself;
- let the message through, but schedule extra checks for the subsequent behaviour of the recipient;
- re-try the previous interactions that led to the current message;
- undo (compensation) but only using the standard OTS item interface;

- perform damage assessment (using the standard interfaces only);
- put OTS item and/or ROS into a consistent known state (experience shows that OTS items are very often left in unknown inconsistent states when they signal errors [3, 9]);
- switch OTS item off and repair it off-line;
- replace failed OTS item with a new one.

Note that with any of the above, the designer may want to:

- initiate state restoration activities for the suspected erroneous party, e.g. preparing to reset/restart it, or running an audit program on its internal state;
- log the event in a log which is used in interpreting severity of future cues (essentially, reconfiguring the cue detecting functions).

Some of these actions imply additional design precautions; e.g., several require precautions to preserve consistency between the computation histories seen by the

OTS item and the ROS; retry of an action may mean re-sending to the originator of the suspect message the last message that caused it to send the suspect message. Some of these reactions may require complex implementations and designers may decide to exclude them a priori to avoid the risk of getting them wrong.

### 3.2 Example

To demonstrate our approach let us consider a simple example. The application considered is illustrated in Fig. 1. The “controlled environment” is a boiler, with sensors (pressure, P, and temperature, T) and actuators (controlling a heating burner which can be ON/OFF, and inlet/outlet valves) for controlling it. Smart sensors and actuators are used with IEEE 488 interfaces. The OTS

item (“OTS” for brevity) is a PID controller, a card which also implements the IEEE 488 interface. The protector is placed between the smart sensors/actuators and the PID controller. The protector's implementation can be either partially in hardware (something that breaks the physical connections and therefore can insert altered messages) or purely in software (if allowed by either the rest of the control system - ROS - or OTS item). The protocols required by the sensors/actuators (e.g., the precision used to encode the input/output signal, message formats, etc.) are completely known to the system integrator, and so is the environment. This simple example is representative of some real design situations, and at the same time useful for illustrating our general ideas.

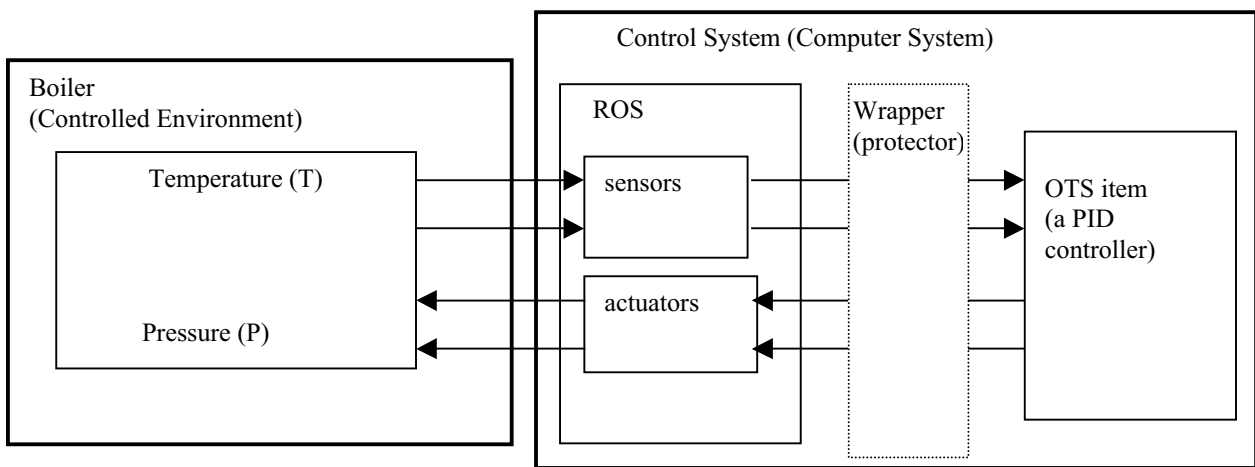


Fig. 1. An example of integrating an OTS item into a system

Table 2

Types of cues	Examples
<i>Output from ROS is illegal per the system designer's specification of system operation</i>	P and T are outside the envelope of values anticipated by system designer
<i>Input to OTS for which OTS is not fully trusted</i>	Measured derivative of T or P is beyond a certain value which is the maximum value for which the OTS has been tested
<i>Input to OTS for which OTS is known to be untrustworthy</i>	T (or) P (or their derivatives) close to boundaries specified for the PID controller, but for which system designer knows (from communication within user groups) that PID has trouble in some cases.
<i>Illegal output from ROS (according to ROS's own specification)</i>	Syntax error in messages exchanged over the IEEE 488 bus
<i>Detectably erroneous output from OTS</i>	'OFF to the burner when the T is low and falling'
<i>Output from ROS is detectably erroneous</i>	ROS sampling rate suddenly increases past specified rate
<i>Output from OTS that is likely to violate implementation constraints in ROS</i>	The PID controller changes its pace of processing and starts sending too frequent messages to ROS

Table 2 lists examples of cues for a wrapper/protector to detect, categorised as in Table 1. In every case shown in Table 2 the protector can take some combination of these actions:

- shut down the boiler to a safe state by sending appropriate commands to the actuators;
- reset the ROS, OTS or both to clear a supposed transient problem;
- take note of a problem but not take any action unless the problem appears to persist.

## 4 Traceability for Protectors

The properties of an OTS item are usually incompletely described; information may be obtained from its interface specification or other associated sources where available. To develop a protector, we need to look at the available information from different viewpoints, each of which can be expressed in terms of relations  $RF$  (requires from) and  $PT$  (provides to) [2]:

- What does the ROS *require from* the OTS item –  $RF(ros,ots)$  ?
- What does the OTS item *require from* the ROS –  $RF(ots,ros)$ ?
- What does the ROS *provide to* the OTS item –  $PT(ros,ots)$ ?
- What does the OTS item *provide to* the ROS –  $PT(ots,ros)$ ?

In each case “What does” refers both to functional properties and non-functional properties such as timeliness and accuracy. The latter are particularly important for OTS items [1] as it is typically the non-functional requirements that differ between (apparently) functionally equivalent OTS items: they can thus be used both to discriminate between OTS candidates and as a cue for developing a protector. Looking at the interactions between these relations we can identify three possibilities:

- $RF(ros,ots) = PT(ots,ros)$  and  $RF(ots,ros) = PT(ros,ots)$ . This is the ideal case where the OTS item is a “perfect fit” for the ROS. This case is unlikely, indeed it is likely that availability of information will limit our knowledge of how close we are to this ideal case.
- $RF(ros,ots) \supset PT(ots,ros)$ . This characterises the situation where there are some properties required by  $ros$  which  $ots$  does not provide (similarly with  $RF(ots,ros) \supset PT(ros,ots)$ ): in this case the OTS item is simply not suitable to be integrated with the ROS.
- $PT(ots,ros) \subseteq RF(ros,ots)$  and  $PT(ros,ots) \subseteq RF(ots,ros)$ . In this case it is the differences between the relations which should be investigated, to identify possible sources of threat which the protector should address.

While we may not be fortunate enough even to have a specification of the ROS, we should be able to fill in the information in viewpoints (1) and (3) above. For (2) and (4) a vendor-supplied product-description will provide some information but this may not be of a trusted quality. In this case the information should be supplemented with external sources such as bug reports, field reports and testing by the system designer. In this context we can view a “bug” as one kind of “unrequired functionality”, i.e. a member of  $PT(ots,ros) - RF(ros,ots)$ .

For each of the properties identified in the viewpoints, a series of supplementary questions can then be systematically asked in order to assess where there would be a threat to either the ROS or the OTS, in a manner similar to a HAZOP (Hazard and Operability) study. Sample questions are:

- What is the effect of this property not being provided?
- What evidence is there that this property can be provided correctly?
- Is this evidence reliable/relevant in this situation?
- For properties provided which are not required, can we predict the effect to ROS or to OTS item?

In each case where threats arise, it should be recorded what protection is implemented (using actions from Section 3), and a level of confidence that the protection is adequate for the threats that have been identified.

## 5 Implementation Issues

A popular way of developing, disseminating and using OTS items is to employ existing component technologies such as, CORBA, DCOM+ and Enterprise Java Beans (EJB). This general approach works well for application-level items (it is worth mentioning that it cannot be applied to software at the levels below the application, e.g. CORBA services, OS, communication protocols, etc.). Such technologies usually offer standard ways of intercepting component calls, and these can be used for implementing protectors. These features are called “interceptors” in CORBA and DCOM+, and “proxies” in CORBA3 and EJB. The CORBA2 specification allows for *interceptor* services. These are services that can be inserted into the normal invocation path for CORBA objects. The interceptor service is registered with the ORB which then ensures that any request sent by a client to an object is passed through the interceptor service, and on return the result also passes through the interceptor service. DCOM+ interceptors are generated automatically by component containers and intercept cross-process calls. EJB and CORBA3 generate *proxies* that stand in place of the target component and allow interception of method invocations sent to the component. The degree to which these interception services and proxies are open

varies. For example, ORBIX has a feature called filtering that is in effect a CORBA interception service.

Although some of these features are either not completely open, as they are used to support particular services, or not flexible enough to allow simple implementation (or tailoring to specific applications, or replacement) of the application-specific protective functions, they can serve as a sound basis for implementing protectors. More experimental work will have to be done in this area to develop better ways of wrapping and to support them with a clear guide to typical patterns for implementing protectors, with libraries supporting efficient implementations of protectors and their typical functionalities. This will assist in systematic incorporation of protector development into the whole system integration process.

## 6 Conclusions

We have proposed a systematic approach to developing protective wrappers which work at the application level to improve the dependability of systems built by integrating OTS software. We treat wrapper development as a specific engineering process including several steps during system integration. Wrapper development starts with specifying *Acceptable Behaviour Constraints (ABCs)* and thus possible violation of the acceptable behaviour of both the OTS item and of the ROS. We discuss initial checklists of possible symptoms of undesired events which the wrappers should try to detect, and defensive actions to be taken. The OTS items may be completely "black" boxes and the information provided by their suppliers is restricted and not reliable. Traceability analysis is used to organise the information for developing and maintaining protective wrappers. Existing component technologies seem to provide sufficient features for implementing the protective wrappers without resorting to non-standard techniques.

The reviewers of this paper helpfully raised the issue of whether this approach would scale well when applied to multiple OTS items in complex systems. This issue really concerns three different effects of multiplying the number of OTS components and of protective wrappers:

- resources: with more OTS items, there will be more places for run-time checks by wrappers and thus more overhead. This is true, but the systematic design approach we advocate allows designers explicitly to state trade-offs between robustness and performance goals. For instance, they can decide the granularity of protective wrapping, e.g. wrapping a set of OTS components together as a single subsystem;
- reliability vs "safety". With the conservative approach we have outlined, of checking for all kinds of symptoms of possible trouble, designers may be drawn to interrupt normal processing too frequently for error recovery. Again, what we advocate is

explicit consideration of the resulting tradeoffs. Without protective wrapping, the default decision is to continue operation no matter what the risk that failures will soon follow, and thus forsaking chances of affordable preventive action;

- unintended side effects and interactions. A mix of protective actions of multiple wrappers and badly understood behaviours of the OTS items looks like a likely source of unintended interactions with possibly serious consequences. A systematic approach to design does not fully protect against them. Designers also need to rely on testing to detect them and on system-level error confinement and recovery provisions for protection against rare but high-consequence failures. Such risks exist in all complex designs with poorly understood components.

In general, we expect a systematic approach for including protection during system integration to help developers in dealing with system complexity, encapsulating errors and developing structured approaches to handling them. The complexity of the protective wrappers themselves should not be an issue as they should be developed systematically (and re-used when possible). Protective wrappers can serve as a solid defence against new problems introduced when upgrading either the OTS items or the rest of the system. Last, our approach promotes the evolution of wrappers as new information is learned about the OTS items' behaviour.

Our future research will concentrate on:

- probabilistic modelling (assessing the protector's error coverage, probability of producing correct action by the wrapped item, and error correlation between the OTS items and the protectors);
- employing diversity with OTS components: investigating architectures with (for example) several diverse items and adjudication of results, and proposing means of pursuing diversity between failure modes of components, wrappers and adjudicators.

**Acknowledgements.** This work is supported in part by the Engineering and Physical Sciences Research Council of the U.K. in the *Diversity with Off-The-Shelf Components* Project (GR/N24056, GR/N23912). Alexander Romanovsky is partially supported by European IST *Dependable Systems of Systems* Project (IST-1999-11585). Our thanks go to Francesca Saglietti and Andrey Povyakalo for discussing with us the example in Section 3.2, and to the anonymous reviewers.

## 7 References

- [1] L. Beus-Dukic. Non-functional requirements for COTS software components. Position Paper. ICSE '2000 Workshop on Continuing Collaborations for Successful COTS Development, June 4-5, 2000, Limerick, Ireland.

- [2] S. Dawkins, S. Riddle. Managing and supporting the use of COTS. In F. Redmill, T. Anderson (eds.), *Lessons in System Safety: In Proc. 8th Safety-Critical Systems Symposium*, 2000.
- [3] P. Koopman, J. De Vale. Comparing the Robustness of POSIX Operating Systems. In Proc. *Fault Tolerant Computing Symposium (FTCS-29)*, Wisconsin, USA, 1999, 30-37.
- [4] P. A. Lee, T. Anderson. *Fault Tolerance: Principles and Practice*. Springer-Verlag, Wien - New York, 1990.
- [5] B. Mayer. Programming by Contract. In D. Mandrioli, B. Meyer (eds.), *Advances in Object-Oriented Software Engineering*. Prentice Hall, 1992.
- [6] C. O'Halloran. Assessing Safety Critical COTS Systems. In F. Redmill, T. Anderson. (eds), *Towards System Safety*. In Proc. 7<sup>th</sup> Safety-Critical Systems Symposium. UK. 1999, 65-74.
- [7] S. Pearson, S. Riddle, A. Saeed. Traceability for the development and assessment of safe avionic systems. In Proc. 8th. Symposium International Council on Systems Engineering (INCOSE '98), Vancouver BC, Canada, July 1998, 445-452.
- [8] D. S. Rosenblum. Towards a Method of Programming with Assertions. In Proc. International Conference on Software Engineering (ICSE-14), 1992, 92-104.
- [9] F. Salles, M. Rodriguez, J.-C. Fabre, J. Arlat. Metakernels and Fault Containment Wrappers. In Proc. *Fault Tolerant Computing Symposium (FTCS-29)*, Wisconsin, USA, 1999, 22-29.
- [10] J. Voas, J. Payne. COTS Software Failures: Can Anything be Done? In Proc. IEEE Workshop on Application Specific Software Engineering and Technology. 1998, 140-144.
- [11] J. Voas. Certifying Off-The-Shelf Software Components. *IEEE Computer*, 31, June, 1998, 53-59.
- [12] I. White. Wrapping the COTS Dilemma. In Proc. *Commercial Off-the-Shelf Products in Defence Applications "The Ruthless Pursuit of COTS"*. IST Symposium, Brussels, Belgium, 3-5 April, NATO. 2000.