On Systematic Design of Protectors for Employing OTS Items

Peter Popov[¶], Steve Riddle[§], Alexander Romanovsky[§], and Lorenzo Strigini[¶]

[¶] Centre for Software Reliability
City University
Northampton Square
London EC1V 0HB
United Kingdom

[§] Centre for Software Reliability
University of Newcastle upon Tyne
Newcastle upon Tyne
NE1 7RU
United Kingdom

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

On Systematic Design of Protectors for Employing OTS Items

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 employed to detect errors or suspicious activity and to execute appropriate recovery when possible; wrapper development is considered as a part of system integration. 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 are classified and a list of possible actions to be undertaken is discussed. The information required for the wrapper development is provided by traceability analysis. Possible approaches to implementing "protectors" in the standard component technologies are briefly outlined.

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 and for assessing the improvements achieved. 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.

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 pair, etc. [LA90]. We view wrappers as redundant bespoke 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 protective wrapper development 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 [PRS98] 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 [DR00]. Within our approach traceability is employed 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 the component and its environment to deal with the flows of control and data going to and/or from the component. The need for wrapping arises from the fact that it is impossible or expensive to change the components, 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 cacheing 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 that can improve the overall system dependability by

protecting both the system from the erroneous behaviour of an OTS item and the item from the erroneous requests from the system [V98]. Our analysis shows that this development can be 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 [VP98] 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 [O99] discusses how OTS items can be used in safety critical systems and proposes to completely isolate them from the rest of the system using encapsulation mechanisms (this approach cannot be applied when relying on OTS items in delivering all types of services). A very interesting approach to developing protective wrappers for an OTS microkernel is discussed in [SR99]. The idea is to specify the correct behaviour of a microkernel and to make the protective wrapper check all functional calls (note, that approach cannot be applied for OTS items that do not have the complete correct specification). In addition, the results of fault injection are used to catch calls that cause errors of the particular microkernel. A similar approach is suggested [KD99] 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 difficult to expect that the vendors of the OTS item will correct it on request. But it is well known that such approaches cannot solve all problems in developing complex dependable systems. 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. Very often wrappers are complex artefacts so that requiring them to perform protection functions may make them more complex or may be costly. 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. This is why wrapper development needs a systematic, disciplined approach.

The simplest system model of OTS use consists of the OTS item to be integrated, the system in which it is to be incorporated (we call it the *Rest Of the System* - ROS, so that the system will consist of the OTS item and the ROS), and the environment controlled by the 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 while 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 [V98, VP98, W00]: 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 e.g. with future versions of the OTS item); the environment (controlled system) may suffer failuresthat 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 system 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 these restrictions.

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, afterwards, implemented in the protector. Development of these constraints is an important part of the methodology which should be used for 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 usual actions to tolerate detected errors (including the "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 [LA90] are employed.

Our approach relies on existing research on executable assertions [R92] and on design by contract [M92]. There are some important issues to be taken into account because these ideas cannot be directly applied to developing protectors: mainly because OTS items are black boxes and integrators do not have their complete, correct specification. Only some types of assertions [R92] are applicable in our context: "consistency between arguments", "dependency of return value on arguments", "frame specifications". But in our opinion these are based on information of a very low level and more complex, application-specific assertions should be included in the ABCs of a component. Design by contracts has a different purpose: in our context the system designer develops a protector using his/her view on the correct contract between the ROS and the OTS item, rather than a contract accepted by the OTS item's designer, and on their correct behaviour with respect to each other.

The various possible ingredients listed in ABCs are alternative, partial descriptions of overlapping views, not subsets in a universe that can be 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 (these two descriptions 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 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 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 four categories are not meant to be mutually exclusive, nor exhaustive. These cues are usefully listed separately as a checklist for system designers and as 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				
	error of ROS	error of OTS item	threat for ROS	threat for OTS item	detection can be programmed using knowledge about
input to OTS item:					
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
output from OTS item:					
illegal output for the OTS item		Y	Y		OTS item
erroneous output for the OTS item		Y	Y		OTS item
risky for the ROS in view of the ROS known 'safety' envelope		Y	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 system internally		Y	Y	Y	part of system that is threatened

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

- report exception/error code, send a message to the operator;
- substitute the message with a "safe" one;
- redirect the message to 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 of 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 [KD99, SR99])
- switch OTS item off end 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 activity for the suspected erroneous party, e.g. prepare to reset/restart party, or run an audit program, on 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, controlled via 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 which use the IEEE 488 interface. The OTS item ("OTS" for brevity) is a PID controller, that is a card which also implements IEEE 488 interface. The wrapper is placed between the smart sensors/actuators and the PID controller. The wrapper implementation can be either partially in hardware (something that breaks the physical connections and therefore can insert altered messages) or purely in software (if this allowed by either ROS or OTS item). The protocols required by the sensors/actuators are completely known (e.g., the precision used to encode the input/output signal, message formats, etc.) to the system integrator, and so is the environment.

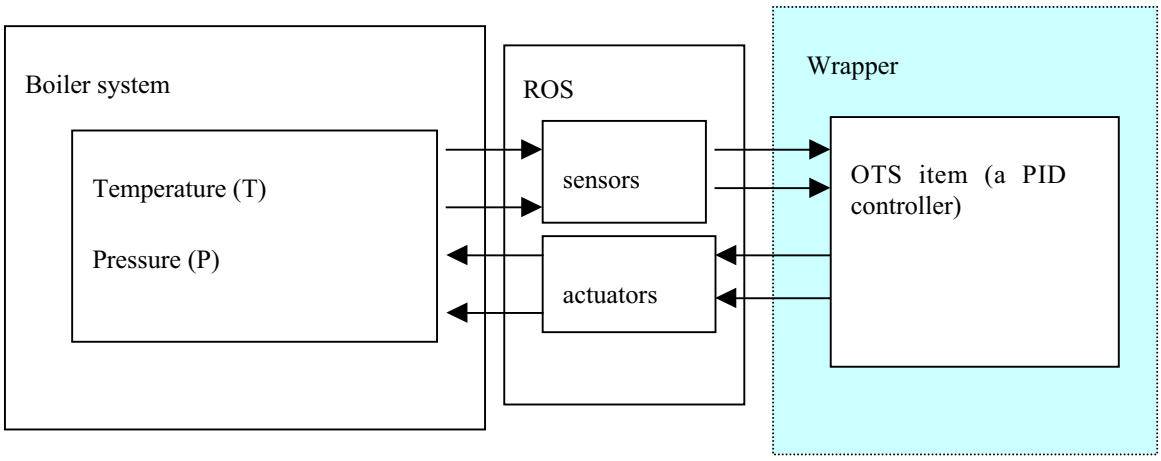

Fig. 1. An example of integrating an OTS item into a system

Table 2

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

In every case shown in Table 2 below the wrapper 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.

We can then identify many possible types of cues that would require an action. In Table 2 we give a list of general types of such cues, with specific examples.

4 Traceability for Protectors

Since we regard the OTS item as a black box any information about its properties must be deduced from its interface specification or from associated sources where available. To develop a protector, we need to look at the available information about this interface from different viewpoints, each of which can be expressed in terms of relations *RF* (requires from) and *PT* (provides to) [DR00]:

1. What does the ROS *require from* the OTS item – *RF(r,c)* ?
2. What does the OTS item *require from* the ROS – *RF(c,r)*?
3. What does the ROS *provide to* the OTS item – *PT(r,c)*?
4. What does the OTS item *provide to* the ROS – *PT(c,r)*?

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 [B00] 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:

- $RF(r,c) = PT(c,r)$ and $RF(c,r) = PT(r,c)$. This 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.
- $RF(r,c) \subset PT(c,r)$. This characterises the situation where there are some properties required by r which c does not provide (similarly with $RF(c,r) \subset PT(r,c)$): in this case the OTS item is simply not suitable to be integrated with the ROS.
- $PT(c,r) \subseteq RF(r,c)$ and $PT(r,c) \subseteq RF(c,r)$. 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 ROS designer. In this context we can view a "bug" as one kind of "unrequired functionality", i.e. a member of *PT(c,r) – RF(r,c)*.

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 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 employing OTS items is to do this within existing component technologies such as, CORBA, DCOM+ and Enterprise Java Beans (EJB) relying only on the standard implementations (without resorting to non-standard techniques). This general approach works well for application-level items (it is worth mentioning that it cannot be applied for employing software at the levels below the application, e.g. CORBA services, OS, communication protocols, etc.). Such technologies 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 when a client sends a request to an object the request 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 adjustment, 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. Our work will focus on providing protector fault tolerance and re-use, and on gaining experience in developing protectors in different application areas.

6 Conclusions

In this paper we propose a systematic approach to developing protective wrappers that work at the application level to improve the dependability of systems built by integrating OTS software. We treat wrapper development as a special engineering process incorporating several activities to be undertaken during system integration. The wrapper development starts with specifying the *Acceptable Behaviour Constraints (ABCs)* that describe possible violation of the acceptable behaviour of both the OTS item and of the ROS. We discuss an initial checklist of possible symptoms of undesired events which the wrappers should try to detect, and defensive actions to be taken. As the OTS items may be completely "black" boxes, in our approach we have to assume that the information provided by their suppliers is restricted and not reliable. Traceability analysis is used to organisethe information for developing and maintaining protective wrappers. Our analysis shows that existing component technologies provide sufficient features for implementing the protective wrappers without resorting to use of non-standard techniques. Protective wrappers can serve as a solid defence against new problems that can be introduced during upgrading both the OTS items and the rest of the system.

Our future research will concentrate on:

- probabilistic modelling (for assessing the protector's 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 by EPSRC/UK *Diversity with Off-The-Shelf Components* Project (GR/N24056). A. Romanovsky is partially supported by European IST *Dependable Systems of Systems* Project (IST-1999-11585). Our thanks go to F. Saglietti and A. Povyakalo for discussing with us the example in Section 3.2.

7 References

[B00] 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.

[DR00] S. Dawkins, S. Riddle. Managing and supporting the use of COTS. In F. Redmill, T. Anderson (eds.), Lessons in System Safety: Proc. 8th Safety-Critical Systems Symposium, 2000.

[KD99] P. Koopman, J. De Vale. Comparing the Robustness of POSIC Operating Systems. In Proc. Fault Tolerant Computing Systems Symposium (FTCS-29), Wisconsin, USA, 1999, 30-37.

[LA90] P. A. Lee, T. Anderson. Fault Tolerance: Principles and Practice. Springer-Verlag, Wien - New York, 1990.

[M92] B. Mayer. Programming by Contract. In D. Mandrioli, B. Meyer (eds.), Advances in Object-Oriented Software Engineering. Prentice Hall, 1992.

[O99] C. O'Halloran. Assessing Safety Critical COTS Systems. In F. Redmill, T.Anderson. (eds), Towards System Safety, Proc. 7th Safety-Critical Systems Symposium. UK. 1999, 65-74.

[PRS98] 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.

[R92] D.S. Rosenblum. Towards a Method of Programming with Assertions. Proc. ICSE-14, 1992, 92-104.

[SR99] F. Salles, M. Rodriguez, J.-C. Fabre, J. Arlat. Metakernals and Fault Containment Wrappers. In Proc. Fault Tolerant Computing Systems Symposium (FTCS-29), Wisconsin, USA, 1999, 22-29.

[VP98] 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.

[V98] J. Voas. Certifying Off-The-Shelf Software Components. IEEE Computer, 31, June, 1998, 53-59.

[W00] 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.