

Protective Wrapping of OTS Components

Peter Popov, Lorenzo Strigini

Centre for Software Reliability

City University

Northampton Square

London EC1V 0HB, United Kingdom

(ptp, l.strigini)@csr.city.ac.uk

Steve Riddle, Alexander Romanovsky

Centre for Software Reliability

University of Newcastle upon Tyne

NE1 7RU

United Kingdom

(steve.riddle, alexander.romanovsky)

@ncl.ac.uk

ABSTRACT

Off-the-shelf (OTS) components are increasingly used in application areas with high dependability requirements. We propose a general approach to developing protective wrappers, in order to integrate OTS items with the rest of the system without reducing the system dependability.

Keywords

Dependability, wrapping, OTS Components, error detection

1 INTRODUCTION

Use of Off-the-Shelf (OTS) components, with their promise of decreased development costs, is occurring in an increasing number of application areas, many of which have high dependability requirements. This raises difficult problems: it may be that the OTS component is not reliable, or that there is insufficient evidence to support any reasonable claims about its reliability. In this paper we focus on a system development approach for wrapping OTS items, to improve system dependability.

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.

Assessing OTS items against high dependability requirements raises concerns about the availability and relevance of evidence. Techniques for structuring and analysing safety-related arguments[5] employ information models and traceability arguments 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 COTS [3].

Several general techniques have been proposed and used to

achieve high dependability by employing redundant software [1]. We consider wrappers as redundant bespoke software used for improving system dependability. Their development is a complex engineering process that cannot be ad hoc, and needs general solutions. We apply traceability information about OTS items to be used in their integration, and develop formal models for analysing the improvement which can be gained in this architecture.

2 THE BASIC ARCHITECTURE

Component wrapping is a well known structuring technique. A wrapper is a specialised component that can monitor, and filter, the flows of control and data going to and/or from a wrapped component. The need for wrapping stems 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. 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 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 [9]. Recent research in the area has addressed some important issues but in our opinion there is a need for systematic general solutions. For example, Voas [8] proposes building wrappers using results of OTS item testing and injecting faults into its interface. This allows certain inputs and outputs to be caught and ignored. A very interesting approach to developing protective wrappers for an OTS microkernel [7] specifies the correct behaviour of a microkernel and makes a protective wrapper to check all functional calls, and the resulting behaviour of the microkernel (note that approach relies on a complete correct specification for the OTS items). In addition, the results of fault injection are used to catch calls that cause errors of the particular microkernel.

We agree that bug reports should be used in developing wrappers, as it is unlikely that the OTS item providers will correct it on request. This cannot solve all problems however: blocking calls which would trigger known faults neither tolerates

the fault, nor protects against undiscovered bugs. Moreover, 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 (potentially complex) piece of software like any other and may be defective, just as any other software engineering artefact. Requiring wrappers to perform protection functions may make them more complex or may be costly. 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. Hence wrapper development needs systematic disciplined approaches that incorporate all steps of the life cycle.

The simplest model of use of an OTS item considers the item to be integrated, the system in which it is to be incorporated (the Rest Of the System - ROS), and the environment controlled by the system. The item may be a black box for the wrapper designers, in that they may know nothing of its internal structure. In practical systems there is a continuum of different levels of greys; some open source OTS items pose some of the same problems as black boxes.

3 DESIGN OF PROTECTOR

There are many reasons why improved protection (i.e. error detection and recovery) is important for system integration [9, 8, 10]: for example, the OTS component specification can be incomplete or incorrect; unspecified (non-standard) features of the component might be used by the ROS; the component might output incorrect commands to the controlled environment.

Acceptable Behaviour Constraints

To design wrappers, system developers (integrators) should form a view on what the OTS component and the ROS do, and do not do, with respect to each other. We call this view the *Acceptable Behaviour Constraints (ABCs)* of the component. The ABCs should be developed formally and systematically and, afterwards, implemented in the protector (for example, a safety envelope). Development of these constraints is an important part of the approach 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 (see Section 4).

Our approach relies on existing research on executable assertions [6] and on design by contracts [4]. These ideas cannot be applied directly for developing protectors, mainly because system designers will not have complete specs of the OTS items. In addition both the ROS and the OTS can have bugs. Only a few types of assertions are applicable in our context: e.g., consistency between arguments, dependency of return value on arguments, and frame specifications (as defined by

[6]. In our opinion these are very low-level considerations: more complex application-specific assertions should be included into ABCs. Design by contracts has a very different purpose and in our context we view protectors as guarantees of execution by contracts.

In addition system designers have to describe what the protector should do if the component does not behave as the ABCs say. These are error recovery actions which can be directed in either or both directions between the OTS item and ROS. Some possible reactions are listed informally below: the list is not exhaustive, as recovery is usually application-specific.

- report exceptions/error code, send a message to the operator, perform additional checks
- let the message through, but prepare for trouble
- re-try
- put OTS item and/or ROS into a consistent known state
- switch OTS item off and repair it off line
- replace failed OTS item with a new one

For each of these actions, it will be advisable to log the event and improve the interpretation of the severity of future cues. Some of these actions imply additional design precautions; e.g., retry of an action may mean re-sending to originator of suspect message the last message(s) that caused it to send a suspect message. Others are very complex actions, and protector designers may decide to exclude them to avoid the risk of getting them wrong.

Case Study

To demonstrate our approach we consider a simple application illustrated in Figure 1. There 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 which implement IEEE 488. The OTS item is a PID controller, a card which also implements IEEE 488. The wrapper is placed between the smart sensors/actuators and the PID controller.

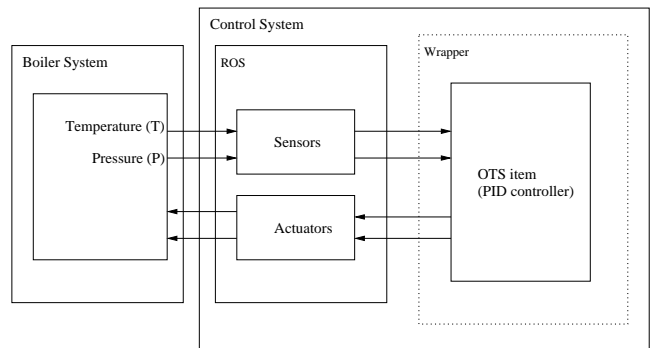


Figure 1: Integrating an OTS item into a system

The wrapper implementation can be either partially in hardware (breaking the physical connections and therefore able

to insert altered messages) or purely in software (if permitted by either ROS or OTS item). We assume the protocols required by the sensors/actuators are completely known to the system integrator, as is the environment.

We can then identify many possible types of cues that would require action: a list general types of these cues is given in Table 1, with specific examples.

Types of cues	Examples
<i>Illegal output from ROS</i>	P and T outside 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 beyond maximum value for which the OTS have 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, known to have trouble in some cases.
<i>Illegal output from ROS (according to ROS's own specification)</i>	Syntax error in messages exchanged over 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

Table 1: Cues for protector

In every case shown in the table, the wrapper can take some combination of these actions:

1. Shut down the boiler to a safe state by sending appropriate commands to the actuators;
2. Reset the ROS, OTS or both to clear a supposed transient problem
3. Take note of a problem but not take any action unless the problem appears to persist.

4 TRACEABILITY FOR PROTECTORS

Since we regard the OTS item as a black box any information about its properties must be deduced from the interface or from associated sources where available. To develop a protector, we need to look at the available information from several viewpoints, each of which can be expressed in terms of relations RF (requires from) and PT (provides to) [3]:

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 [2] 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 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.
- $RF(r, c) \subset (c, r)$. This characterises the situation where there are some properties required by r which c does not provide (similarly with $RF(c, r) \subset (r, c)$): in this case the OTS item is simply not suitable to be integrated with the ROS.
- $PT(c, r) \subseteq (r, c)$ and $PT(r, c) \subseteq (c, r)$. 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?

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 COTS items is to do this within existing component technologies such as, CORBA, DCOM+ and Enterprise

Java Beans (EJB). This general approach works well for application-level items (although it is worth mentioning that it cannot be applied for employing software at the levels below the application, e.g. CORBA services, OS). Such technologies offer standard ways of intercepting component calls that 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 an interceptor service that can be inserted into the invocation path for the component. The service is registered with the ORB that then ensures that when the client sends a request to a component the request is passed through the interceptor service and on return the result also passes through the 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 different providers. For example, ORBIX has a feature called filtering that is in effect a CORBA interception service. In some technologies these features are less open as they are used to support particular services.

These standard ways of wrapping components can serve as a sound basis for implementing protectors. More experimental work will have to be done in the area to develop better ways of wrapping and to support them with a clear guide describing typical patterns of implementing protectors, with libraries supporting their efficient implementations and their typical functionalities. This will provide an assistance in systematic incorporation of protector development into the whole system integration process. Topics of our future research include: supporting protector fault tolerance and reuse; gaining experience in developing protectors in different application areas.

6 DISCUSSION AND CONCLUSIONS

In this paper we have proposed a general approach to developing protective wrappers that work at the application level. Protective wrappers can serve as a defence against new problems that can be introduced during upgrading both the ROS and the OTS item. We do not treat these wrappers as “components” that need to be specified (and wrapped) in the same way as the OTS item.

The designer who assembles a system using OTS items will specify protective wrappers to improve system dependability, and we have suggested some directions for making this part of design more effective. We have proposed a procedure for developing and describing protective wrappers. We use traceability techniques to collect information for developing wrappers, develop rigorous models for analysing the improvement which can be gained in this architecture, and propose implementation techniques which can be used within the existing component technologies for implementing protectors.

Wrapper development is part of system integration, and should be recognised as a complex engineering process. We have described general approaches to specifying constraints (ABCs) that describe correct behaviour of the ROS and the OTS item. Future research will concentrate on probabilistic modelling and employing diversity with OTS components: investigating architectures with (for example) several diverse items and proposing means of achieving diversity between failure modes of components.

ACKNOWLEDGEMENTS

This work is supported by EPSRC/UK *Diversity with Off-The-Shelf Components Project*. A. Romanovsky is partially supported by European IST DSoS Project

REFERENCES

- [1] T. Anderson, P. A. Lee. *Fault Tolerance: Principles and Practice*, Prentice Hall. 1981
- [2] L. Beus-Dukic. Non-functional requirements for COTS software components. Position Paper. Proc. ICSE 2000 Workshop on Continuing Collaborations for Successful COTS Development, June 4-5, 2000, Limerick, Ireland.
- [3] S. Dawkins and S. Riddle. Managing and supporting the use of COTS. In F. Redmill and T. Anderson, editors, *Lessons in System Safety: Proc. 8th Safety-Critical Systems Symposium*, pages 284–300, Southampton, 2000. Safety-Critical Systems Club, Springer
- [4] B. Mayer. *Programming by Contract*. In D. Mandrioli, B. Meyer (eds.) *Advances in Object-Oriented Software Engineering*. Prentice Hall, 1992.
- [5] S. Pearson, S. Riddle and A. Saeed. Traceability for the development and assessment of safe avionic systems. In Proc. 8th Int. Symposium International Council on Systems Engineering (INCOSE '98), pp 445-452, Vancouver BC, Canada, July 1998.
- [6] D.S. Rosenblum. Towards a Method of Programming with Assertions. *Int. Conf. ICSE-14*, pp.92-104. 1992.
- [7] F. Salles, M. Rodriguez, J.-C. Fabre, J. Arlat. Metakernels and Fault Containment Wrappers. In *Fault Tolerant Computing Systems Symposium (FTCS-29)*, 1999, pp. 22-29.
- [8] J. Voas, J. Payne. COTS Software Failures: Can Anything be Done? *IEEE Workshop on Application specific Software Engineering and Technology*. 1998, pp. 140-144.
- [9] J. Voas. Certifying Off-The-Shelf Software Components. *IEEE Computer*, 31, June, 1998, pp.53- 59.
- [10] I. White. Wrapping the COTS Dilemma. In *Commercial Off-the-Shelf Products in Defence Applications “The Ruthless Pursuit of COTS” IST Symposium*, Brussels, Belgium, NATO. 2000.