Prevention of Failures due to Assumptions made by Software Components in Real-Time Systems*

Ajay Tirumala, Tanya Crenshaw, Lui Sha, Girish Baliga,
Sumant Kowshik, Craig Robinson†, Weerasak Witthawaskul
Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL 61801
{tirumala,tcrensha,lrs,gibaliga,kowshik,clrobnsn,witthawa}@uiuc.edu

Abstract

Large scale real-time systems consist of hundreds of commercial off-the-shelf (COTS) and custom software components. Mismatched assumptions between software components are a prime source of failures in these systems. Further, component assumptions are often implicit due to the limitations of current software interfaces. In this work, we introduce a framework to explicitly expose assumptions in software components, and automatically verify these assumptions during system integration. We manage the propagation and composition of these assumptions in the presence of changes and upgrades to individual components.

1 Introduction

Multi-million dollar real-time systems are typically built from COTS and custom components developed by different developer teams. These components usually expose functional interfaces, which we call *software interfaces*, to exchange data beween interacting modules. These interfaces represent the format for data exchange as seen in traditional C libraries and Java interfaces. However, components in a real-time systems make additional assumptions that cannot be represented in such an interface. Examples of such assumptions are units of measurement, value ranges, environmental assumptions and domain-specific implicit attributes.

Several catastrophic failures in large-scale real-time systems can be attributed to the inadequacy of existing interfaces and the inability to track implicit assumptions of components. The Ariane 5 disaster [2] was caused by the reuse of an Ariane 4 component. An implicit assumption made by the component that a variable would never overflow 16 bits was violated in Ariane 5. This assumption, though documented, was not validated in the new system.

A survey we conducted [12] indicates that algorithmic defects in software occur less frequently than the defects that are related to integration issues. In real-time systems, integration defects are caused by assumption mismatches between software components and environmental assumptions which may be invalid. We conjecture that the relatively lower percentage of algorithmic defects is due to mature unit-testing and formal verification of individual components (where the algorithms are embedded). These approaches, however, do not scale to integrated systems due to their complexity and larger state spaces.

In this paper, we present a framework to track assumptions during system integration by specifying *propery interfaces* of components. To motivate this, an example real-time system is described in

*This research is supported in part by MURI grant N00014-01-0576, NSF grants ANI 02-21357, CCR 02-37884, CCR 03-25716 and CCR 02-09202

†Coordinated Science Lab, UIUC

Section 2. Our framework is then presented in Section 3. Section 4 discusses the capabilities of our framework. We then present related work in this field in Section 5, which is followed by conclusions and future work.

2 Example: Real-time Control System

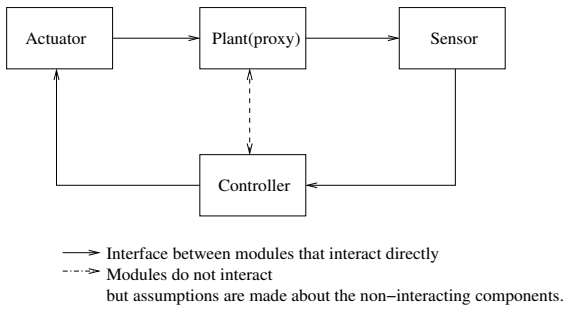

Figure 1. Software interfaces and components in the inverted pendulum control system

To illustrate the significance of this problem, consider a feedback control system for an inverted pendulum, consisting of four major components: controller, actuator, sensor and pendulum proxy[1]. The components communicate through traditional software interfaces (solid arrows in Figure 1). However, many component assumptions are not captured by these interfaces. For instance, the sensor-controller software interface does not specify that the controller assumes that the sensor reports the angle between the pendulum and the track in 'degrees'. Also, the controller assumes a length and mass of the pendulum, which is an example of an implicit assumption of the controller regarding a component with which it does not interact (dotted line in Figure 1). In our study, we found that this simple four-component system had over forty assumptions, that are not a part of the software interfaces. This problem is magnified in real-world software – the ground-based command system for NASA's Hubble Space Telescope consists of 30 COTS components [11].

[1]We use the plant proxy to represent the controlled device, which itself can be changed/upgraded

3 Property Interfaces

We address the problem of representing and validating assumptions made by different components using the notion of a *property interface* between components. This is orthogonal to the software interface between communicating components. *Property interfaces* are used to propagate implicit component assumptions that are not a part of the software interface. They can exist between interacting (e.g. sensor and controller) and non-interacting modules (e.g. controller and plant proxy). Between any two components connected by a property interface, each component publishes its assumptions and its guarantees. The assumptions of each component are verified against the guarantees provided by the other component. The assumptions and guarantees, typically, consist of global component properties, implicit properties of exchanged data elements (if any), and environmental properties (all of these are not a part of the software interface).

3.1 Classification of Properties

The properties of a component in a real-time system can be classified on the basis of their frequency of change. This classification determines the validation methodology of a property in the presence of component changes and upgrades.

System configuration properties: These properties do not change during a mission (series of executions) of the real-time system. E.g. 'Mass of the pendulum' is a system configuration property since it does not change over a mission. When a different pendulum is used in a different run, this property will change for that run.

Static properties: These properties never change during the lifetime of the software. E.g. The controller may assume certain default values, such as the value of the 'Gravitional constant on the surface of the earth', as part of its operating conditions.

Dynamic properties: These properties are those which change during the mission, but at a significantly lower rate than real-time data flow between the modules. E.g. The wear of the motor

gear may require it to be replaced after running for 1000 hours. In this case, the property 'max motor torque' is a dynamic property.

4 Framework

Our framework manages property interfaces of software components as assumptions and guarantees. It enables a component developer to express non-functional properties and validates component assumptions against the guarantees offered by other components. The validation technique is based on the classification of a property from the previous section.

We have developed a tool with an XML back-end to encode the assumptions and efficiently check these assumptions with guarantees. Assumptions are stored as XML schema and guarantees as XML documents which match XML schema documents[2]. Our tool uses an automatically generated GUI as a front-end for increased usability. It performs the following functions:

Check and flag inconsistent assumptions: Given the assumptions and guarantees for a set of components, the tool automatically verifies that all component assumptions are matched by component guarantees. Thus, the assumptions made on the interfaces are machine checkable. E.g. If the units of sensing delay expected by the controller is in *milliseconds* and the sensor specifies it in *seconds*, an assumption mismatch will be flagged.

Verify safety of upgrades: The tool will enforce that a component's assumptions are compatible with the guarantees of other components in the presence of software or hardware upgrade. E.g. If the pendulum is changed and the mass of the new pendulum does not match the controller's assumption, the tool will flag an assumption mismatch due to the upgrade.

[2]XML Schema specify the structure of XML documents and restrictions on the values that the elements in the XML documents can take. They are analogous to the meta-data for a database. For official specification, tools and developments, the reader is referred to [**?**].

Compose properties: The tool has provisions for inserting the functions to compose the properties. We define composition of a set of properties $P_1, P_2, \ldots, P_n$ with the function f as the result of the function $f(P_1, P_2, \ldots, P_n)$. For example, if all the software components in the pendulum example are executed on the same processor, the users can insert a function which calculates the control loop delay using the measured processing and communication times for different components.

Compose modules: The tool provides a mechanism to compose modules. The system integrator can combine several modules and view them as a sub-system with a well-defined software and property interface. A compose and decompose function on a sub-system of components only exposes the component assumptions that are relevant outside the sub-system. All internally matched assumptions are hidden from the external components. This feature increases the manageability of the system without compromising the assumption mismatch detection capability. For example, in the figure 2, if the guarantee G_{31} satisfies requirement/assumption R_{31} and the guarantee G_{32} satisfies requirement/assumption R_{32}, the property interfaces for subsystem enclosed by the dotted box will not include G_{31}, G_{32}, R_{31}, or R_{32}.

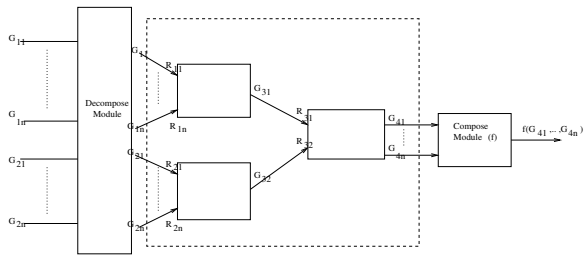

Figure 2. Combining a set of modules with decompose and compose modules

5 Related work

Real-time CORBA by Schmidt *et.al.* [9] provides middleware solutions for real-time systems. Based on this work, there have been effective so-

lutions to manage the QoS properties of the system adaptively and at run-time [5]. Composition of real-time systems has been addressed in [10]. In comparison, our work is more general since it tracks all kinds of assumptions and guarantees made on software interfaces. Also, our notion of property interfaces captures assumptions between modules that do not interact with each other. MetaH [13] provides a framework for designing real-time systems and allows system dependencies to be encoded. The work, however, is strongly tied to fixed priority scheduling. CORBA [3], COM [1] and DCOM [4] have facilities for composition, but their focus is on reusability rather than tracking implicit assumptions with guarantees of the software components. Contracts [7] and Rely/Guarantee [8] can be used to validate the input and output parameters in software interfaces. However, they cannot handle assumptions which are not represented as code (physical representation). The work most closely related to our work is [6]. In comparison, our framework is based on an open standard, making it extensible. Also, we handle composition of assumptions and software modules, which is critical for larger systems.

6 Conclusions and future work

In this paper, we described a framework for capturing implicit assumptions in the environment and software and hardware components of a real-time system. We introduced the notion of property interfaces, which capture implicit assumptions between interacting and non-interacting components. Automatically tracking and verifying these assumptions will reduce testing time and reduce the number of end-product defects. Motivated by a simple inverted pendulum controller, whose components itself contains a large number of implicit assumptions, our framework is implemented using an XML-based engine and also supports composition of software modules and assumptions on the modules elegantly. As future work, we plan to automatically track assumptions in larger real-time systems with tens to hundreds of software modules. We plan to determine the percentage of end-product defects which can be averted, and the reduction in testing effort.

References

[1] *Microsoft Corporation and Digital Equipment Corporation. The Component Object Model Specification.* 1995.

[2] *Ariane 5 Failure - Full Report. http://sunnyday.mit.edu/accidents/ Ariane5accidentreport.html.* July 1996.

[3] *Object Management Group. The Common Object Request Broker: Architecture and Specification, Revision 2.0, formal document 97-02-25. http://www.omg.org.* 1997.

[4] *Microsoft Corporation and Digital Equipment Corporation. The Distributed Component Object Model Specification.* 1998.

[5] Y. Krishnamurthy, I. Pyarali, C. Gill, L. Mgeta, Y. Zhang, Torn, and D. Schmidt. The design and implementation of Real-Time CORBA 2.0: dynamic scheduling in TAO. In *Proceedings of Real-Time and Embedded Technology and Applications Symposium*, May 2004.

[6] J. Li and P. Feiler. Impact analysis in real-time control systems. In *Proceedings of International Conference on Software Maintenance*, 1999.

[7] B. Meyer. Applying "Design by Contract". *IEEE Computer*, 25(10):40–51, 1992.

[8] P.Collette and C. Jones. Enhancing the Tractability of Rely/Guarantee Specifications in the Development of Interfering Operations. *Proof, Language and Interaction*, pages 277–307, 2000.

[9] D. Schmidt. *Real-time CORBA http://www.cs.wustl.edu/schmidt/RT-ORB-std-new.pdf.gz.* may 1999.

[10] J. Stankovic, R. Zhu, R. Poornalingam, C. Lu, Z. Yu, M. Humphrey, and B. Ellis. VEST: An Aspect-Based Composition Tool for Real-Time Systems. In *Proceedings of Real-Time and Embedded Technology and Applications Symposium*, May 2003.

[11] J. R. T. Pfarr. The Integration of COTS/GOTS Within NASA's HST Command and Control System. In *Proceedings of the First International Conference on COTS-Based Software Systems.*, Februrary 2002.

[12] A. Tirumala. *analysis of cause of defects in open source real-time software - a case study of TinyOS. http://www-rtsl.cs.uiuc.edu/defect_analysis.html.* Dec 2004.

[13] S. Vestal. MetaH Support for Real-Time Multi-Processor Avionics. In *Proceedings of Real-Time Systems Symposium*, Dec 1997.