Driving Architectures with UML 2.0

The TAU Generation2 Approach to Model Driven Architecture

A Telelogic White Paper

Authors: **Cris Kobryn and Eric Samuelsson**

Published: August 1, 2003

Abstract

As the world's demand for software inexorably increases, we need to increase the quantity and improve the quality of the software that we produce. There are two major technology trends that aim to address our insatiable appetite for software: automation and outsourcing. This white paper describes a technical approach based on proven engineering principles that primarily addresses the trend towards software automation, but can also be applied towards software outsourcing. This model driven approach to software development described here, which is based on architectural blueprint languages such as UML™ 2.0, and automated by power tools such as TAU® Generation2™, can substantially improve software productivity and quality. The approach is compatible with the Object Management Group's Model Driven Architecture® (MDA®) initiative, and takes advantage of its second-generation MDA standards, such as UML 2.0 and the UML 2.0 Profile for Testing. The paper concludes with speculation about the future of MDA as it evolves from a conceptual to a technical architecture.

Telelogic

CONTENTS

INTRODUCTION

As we transition from the Industrial Age to the Information Age, software is preeminent. We rely on enterprise software applications to run our businesses, embedded software applications to operate our machines, and multimedia software to entertain us. All of these software applications are built on a complex software infrastructure that consists of operating systems, middleware, and networking software.

While this sprawl of software still requires hardware for storage and execution, there is a growing trend for hardware functions to be replaced by software functions. For example, it is now common practice to apply software in the manufacture of cars to improve their fuel economy and safety. Similarly, it is a frequent routine to apply software in the manufacture of stereo and video equipment to improve their fidelity and reduce their footprints.

As we transition from the forty hour work week associated the Industrial Age to the flex hours and increased leisure time associated with the Information Age, we find that software also pervades our entertainment. For example, we commonly listen to music and watch videos that are digitally produced and stored with software.

As the world's appetite for software inexorably increases, we need to improve both the quantity and quality of software that we produce. There are two trends to address this insatiable demand: automation and outsourcing. Software automation refers to the process of transferring the work of developing software from wetware (humans) to other software and hardware. Software outsourcing refers to the general trend to procure human services for developing software from external companies or organizations, especially those in foreign countries with a high education-to-compensation ratio. In the latter case, the outsourcing is commonly referred to as *offshore outsourcing*.

This white paper describes a technical approach based on proven engineering principles that primarily addresses the trend towards software automation, but can also be applied towards software outsourcing. The model driven approach to software development explained here, which is based on architectural blueprint languages such as UML™ 2.0 and power tools such as TAU Generation2™, can substantially improve software productivity and quality.

The first part of the paper introduces the concept of model driven development as it applies to software. It next discusses the Object Management Group's Model Driven Architecture™ initiative, which provides a conceptual architecture and key standards, such as UML 2.0, that enable model driven development.

The second part of the paper shows how the powerful concepts described in the first part can be applied to automate the software lifecycle, starting with business requirements and culminating in testing. In particular, it shows how TAU Generation2 can automate the transformation of a Platform Independent Model of requirements into a Platform Specific Models that can generate production quality code and test scripts. The paper concludes with some speculation about the future of MDA as it evolves from a conceptual to a technical architecture.

THE MODEL-DRIVEN ADVANTAGE

The basic ideas behind model driven development can be traced to the ancient Egyptians, who over 4000 years ago applied both scale models and mathematical models to architect and build their pyramids. The Egyptians architects discovered that by combining scale and mathematical models with incremental prototyping techniques, they could scale their pyramid building technology to the point where they produced the third largest building on the planet: the Great Pyramid of Giza.[1] This insight about the power of modeling, whether passed on or learned independently, has allowed our species to construct progressively larger and more complex buildings, vehicles, machines and electronics.

We are now in the process of applying this insight about modeling to software development, where it is becoming common to refer to *model-based development*. As it pertains to software, model driven development can be defined as follows:

> **model driven development**: An iterative, incremental software development process where the model of a system is iteratively refined into an executable system via a series of systematic mapping transformations. These mapping transformations are typically either partially or fully automated. [Kobryn 2003a]

Model Driven development can be sharply contrasted with conventional software development. Whereas traditional software development tends to be code-centric and human intensive, model driven development is inclined to be model-centric and favors automation.

The differences between *round-trip engineering* and model driven development are more subtle but nevertheless important. Whereas model driven development emphasizes the forward engineering of source code from models via a series of systematic mapping transformations, round-trip engineering is equally inclined to accommodate the reverse engineering of models from source code. While the ability to choose and change either the model or the code is theoretically attractive to most developers, in practice it produces mixed results. The underlying reason for this is that, while models commonly support multiple, progressively refined abstraction levels (e.g., requirements models, analysis models, design models), programming code represents a single, primitive abstraction level – the implementation. Consequently, when a developer changes programming code to address implementation details, such as optimizing execution time or physical storage, the reverse propagation of the changes to the models is frequently problematic. Problems range from poor mapping transformations, where implementation details intrude upon the higher level models, to cases where the mapping transformations are either lacking or incorrect.

The advantages of a model driven development approach are summarized in Table 1 [Kobryn 2003a].

[1] Only the Great Wall of China (c. 215 B.C.E.), which is visible from outer space, and the Grand Coulee Dam (1975), are larger than the Great Pyramid of Giza (c. 2680 B.C.E).

Table 1: Advantages of Model Driven Development

TECHNOLOGY DRIVERS	TECHNOLOGY ADVANTAGES	BUSINESS ADVANTAGES
model = requirements	Ensure requirements are an integral part of model.	Ensure right system is being built.
analysis and design models	Support a wide variety of software methods and processes.	Ensure system is being built the right way.
model simulation	Automate software validation and verification.	Reduce errors and costs early in the lifecycle.
model = code	Automate generation of production quality code.	Accelerate time to market.
model = test	Automate testing.	Ensure system is correct and reliable.

An explanation of the technology drivers in Table 1 follows:

- model = requirements: In a model driven approach, the models must be driven by requirements, which are ideally expressed using a requirements model that can be traced through all related model views (e.g., analysis model, design model, implementation model, test model).
- analysis and design models: Analysis models refine requirements models into high-level, logical constructs that are meaningful to business, software and systems analysts. Design models in turn refine the analysis models into lower-level, physical constructs that can implemented.
- model simulation: One of the most important advantages of a model-based approach is that models can simulate the system they are representing. These simulations provide a cost-effective means to automate system validation and verification (V&V).
- model = code: Another key advantage of a model-based approach is that executable models with full action languages can be used to generate complete production quality code, that contains procedural logic as well as code skeletons.
- model = test: Not only are models useful at the beginning of the development process, they are also helpful at the end to facilitate both black-box and white-box testing of units and systems.

MODEL DRIVEN ARCHITECTURE

The Object Management Group, the world's largest software consortium, is promoting model driven development through its Model Driven Architecture (MDA) initiative. The OMG defines MDA as follows [MDA 2003]:

> **Model Driven Architecture** (MDA): An approach to IT system specification that separates the specification of functionality from the specification of the implementation of that functionality on a specific technology platform.

In order to enforce the separation of concerns between specifications and their implementations, the MDA defines two kinds of models, Platform Independent Models (PIMs) and Platform Specific Models (PSMs), which it defines as follows:

> **Platform Independent Model** (PIM): A model of a subsystem that contains no information specific to the platform, or the technology that is used to realize it.
>
> **Platform Specific Model** (PSM): A model of a subsystem that includes information about the specific technology that is used in the realization of it on a specific platform, and hence possibly contains elements that are specific to the platform.

MDA as a Conceptual Architecture

In the same way that the OMG's Object Management Architecture™ (OMA™) was a conceptual architecture for CORBA™ and its distributed services, the MDA serves as a conceptual architecture for the OMG's primary modeling standards: Unified Modeling Language™ (UML™), Meta Object Facility™ (MOF™) and Common Warehouse Model™ (CWM™)-. Of these three modeling standards, UML is the most essential, since it is the industry standard for software modeling, and all of the other MDA modeling standards, including MOF and CWM, are defined in terms of UML. Stated otherwise, UML is the lingua franca for the MDA initiative.

Figure 1 shows the relationship between a generic Platform Independent Model and Platform Specific Models for several platforms (J2EE, .NET and BREW[2]) using UML notation [Kobryn 2003a]. In this figure, the J2EE, .NET and BREW Platform Specific Models are shown to be derived from a Platform Independent Model. Similarly, the JAR, DLL and BREW file artifacts are shown to be derived from the J2EE, .NET and BREW PSMs.

[2] Binary Runtime Environment for Wireless (BREW) is a platform for developing and deploying applications on wireless devices. See http://www.qualcomm.com/brew/.

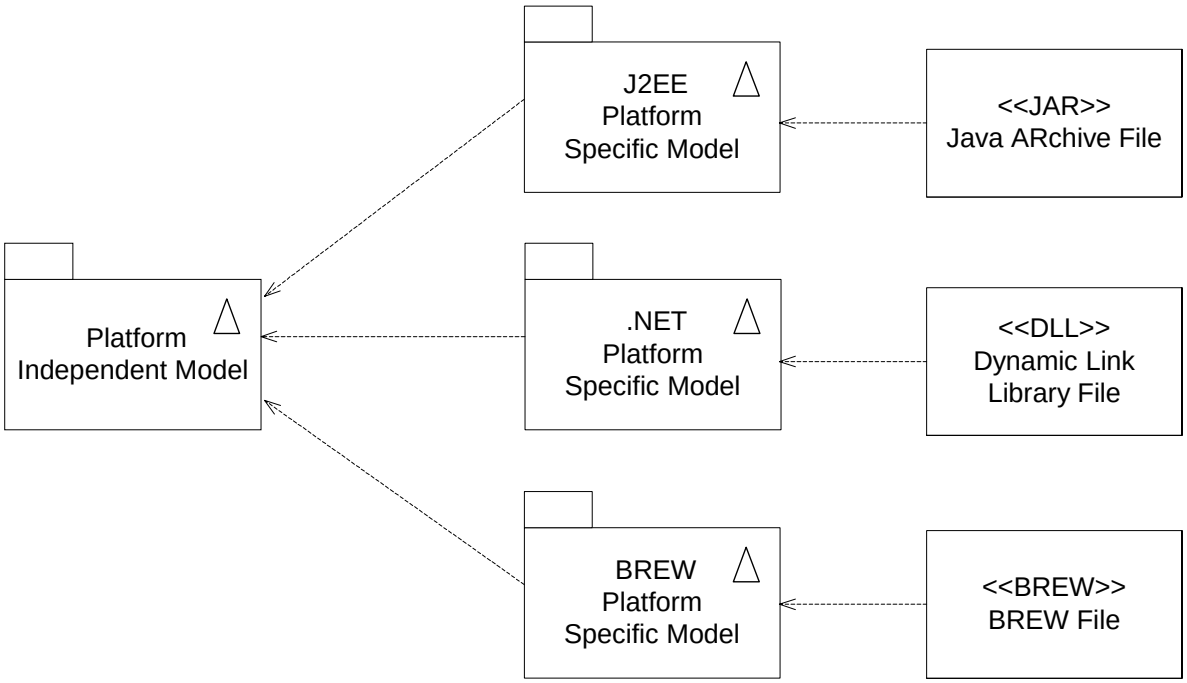


Figure 1: MDA Conceptual Architecture

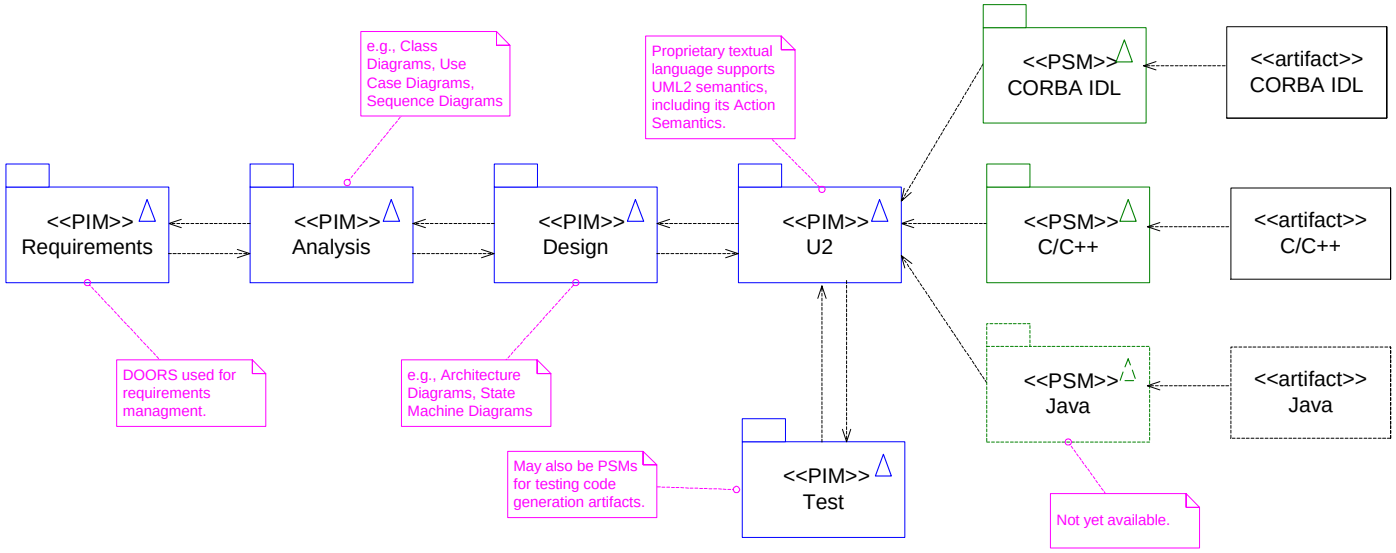
MDA as a Technical Architecture

The MDA conceptual architecture and first generation MDA modeling standards (e.g., UML 1.x, MOF 1.y, CWM 1.z) provide a bold vision and a standards roadmap for realizing the benefits of model driven development. However, in order to realize the full benefits of the MDA approach a robust technical architecture with mature modeling standards is required. Fortunately, the OMG is in the process of finalizing its second generation modeling standards (e.g., UML 2.0, MOF 2.0), and tool vendors are implementing them in their products.

Figure 2 shows the TAU Generation2 technical architecture for implementing MDA using UML 2.0. The figure includes five different PIM views: a Requirements model, an Analysis model, a Design model, a Test model, and a U2 model. The U2 model is

actually a proprietary textual language that maps to the UML 2.0 semantics, including its Action Semantics.

Figure 2: TAU G2 Technical Architecture for MDA



LANGUAGE AND TOOLS: UML 2.0 AND TAU G2

In order to successfully implement model driven solutions, both modeling language standards and tools that implement them are required. In this section we explore the recently adopted UML 2.0 standard, and the first commercial tool that has implemented it, TAU Generation2.

Architectural Blueprint Language: UML 2.0

In order to successfully implement a complete, correct and robust MDA solution, the system architects and designers require an architectural specification language that is precise and concise.

The major improvements to UML 2.0 include, but are not limited to, the following [Kobryn 2003b]:

- Support for component-based development via composite structures. Structured classifiers (both Classes and Components) can be hierarchically decomposed and assembled ("wired") via Parts, Ports, and Connectors.
- Hierarchical decomposition of structure and behavior. In addition to Classes and Components, which are structural constructs, UML2 supports the hierarchical decomposition of the major behavioral constructs, such as Interactions, State Machines, and Activities.
- Cross integration of structure and behavior. The decomposed structures described above can be flexibly integrated with each other. For example, the same Parts that are used in a composite structure diagram of a Class to show its internal structure, can also be used in a sequence diagram to show how the internal structures communicate with each other.
- Integration of action semantics with behavioral constructs. UML actions are now defined in as much detail as a programming languages's actions (or statements), so that you can define executable models for simulations and code generation.
- Layered architecture to facilitate incremental implementation and compliance testing. UML 1.x was a large language, and UML 2.0 is larger still. Taking a lesson from other large languages (e.g., SQL), UML 2.0 packages are organized into three layers (Basic, Intermediate, and Complete) in order to make it easier for vendors to implement and more efficient for standards organizations to test compliance.

Cumulatively these improvements mark a significant evolution of the UML, increasing its precision and expressiveness so that it can be effectively used to model large, complex architectures. Examples that show how UML 2.0 accomplishes this can be found in *Architecting Systems with UML 2.0* [Björkander 2003].

Power Tool: TAU Generation2

Although a precise and concise architectural blueprint language is required for a successful model driven development approach, it alone is insufficient. The language must be accompanied by a power tool that faithfully and efficiently implements the language, so that it can automate the mapping transformations across the various models.

TAU Generation2™ (TAU G2™) is a family of model-centric and role-based tools that are among the first to implement the recently adopted UML 2.0 standard. The tool family consists of TAU®/Developer™ for Software Engineers, TAU®/Architect™ for Systems Engineers, and TAU®/Tester™ for Test Engineers. TAU G2 builds on the model driven compilation technology perfected in TAU SDL Suite™ (a.k.a. TAU G1). TAU G1 proved that real-time software development can be automated using mature specifications languages such as Specification and Description Language (SDL) and Message Sequence Chart (MSC). Given that many of the advanced language features offered by SDL and MSC were adapted and incorporated into UML 2.0, there were compelling technical and market reasons to combine TAU G1's model driven compilation technology with UML 2.0 to produce TAU G2.

TAU G2 provides the following features:

- Precise and unambiguous system specification – Engineers can visually specify systems using the precise, standardized and non-proprietary language of UML 2.0. This results in easy-to-understand, clear and unambiguous specifications.
- Specification of behavior – Whereas most system modeling tools allow only the specification of the system's architecture or structure, TAU G2 also allows engineers to visually specify the dynamic aspects of the system's behavior.
- Automatic application generation - TAU/Developer is the only tool that supports executable UML 2.0 models with behavioral specifications. Developers have access to pre-defined, verifiable code patterns that ensure high quality standards. With these capabilities, developers can automatically generate complete applications.
- Dynamic model verification - With fully controllable model simulation, engineers can verify their work in the analysis, design, and implementation phases. As a result, they can quickly locate and remove errors early when corrections are relatively easy and inexpensive.
- Scalability - Large scale systems can be specified and models can be mapped to how teams want to work, rather than having restrictions imposed by the tool. System architecture and behavior also can be modeled and viewed at the appropriate level of abstraction for the user.

- Integrated requirements management via Telelogic DOORS® - TAU G2 is integrated with Telelogic DOORS, the market leading requirements management solution.
- Automated documentation via Telelogic DocExpress® - TAU G2 is integrated with DocExpress, which provides automatic extraction and formatting of system or software application documentation.
- Change and configuration management via Telelogic SYNERGY™ - SYNERGY provides change and configuration management for TAU G2 and related products.

In this section we show an example of how UML 2.0 and TAU G2 can be used together to drive an architecture for a Satellite Control System (SCS). The Satellite Control System controls the physical behavior of a satellite, such as the physical orientation of its axes relative to a reference line or plane (e.g., the horizon).

Figure 3: SCS requirements expressed as DOORS structured text

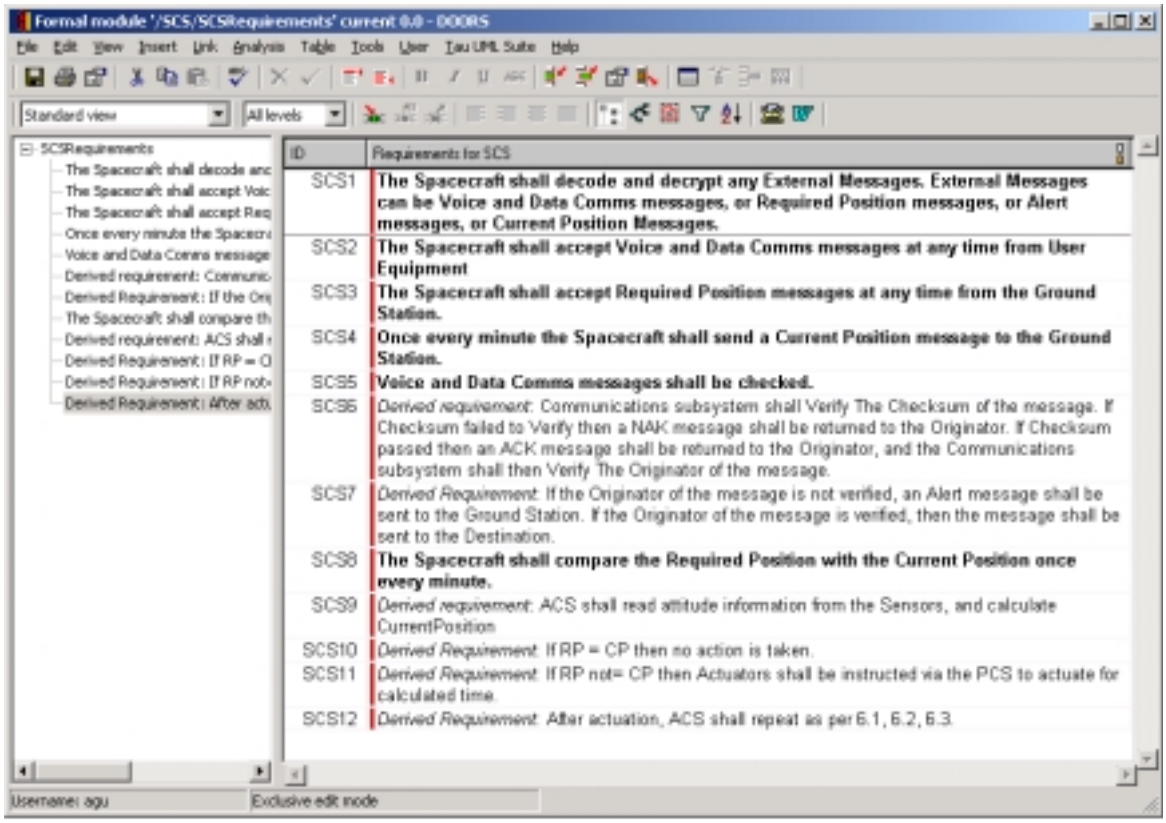


Figure 3 shows a DOORS window that specifies the text-based requirements for the Satellite Control System (SCS). For example, requirement SCS8 specifies that “The Spacecraft shall compare the Required Position with the Current Position once every minute.”

Figure 4: SCS requirements expressed as UML use cases

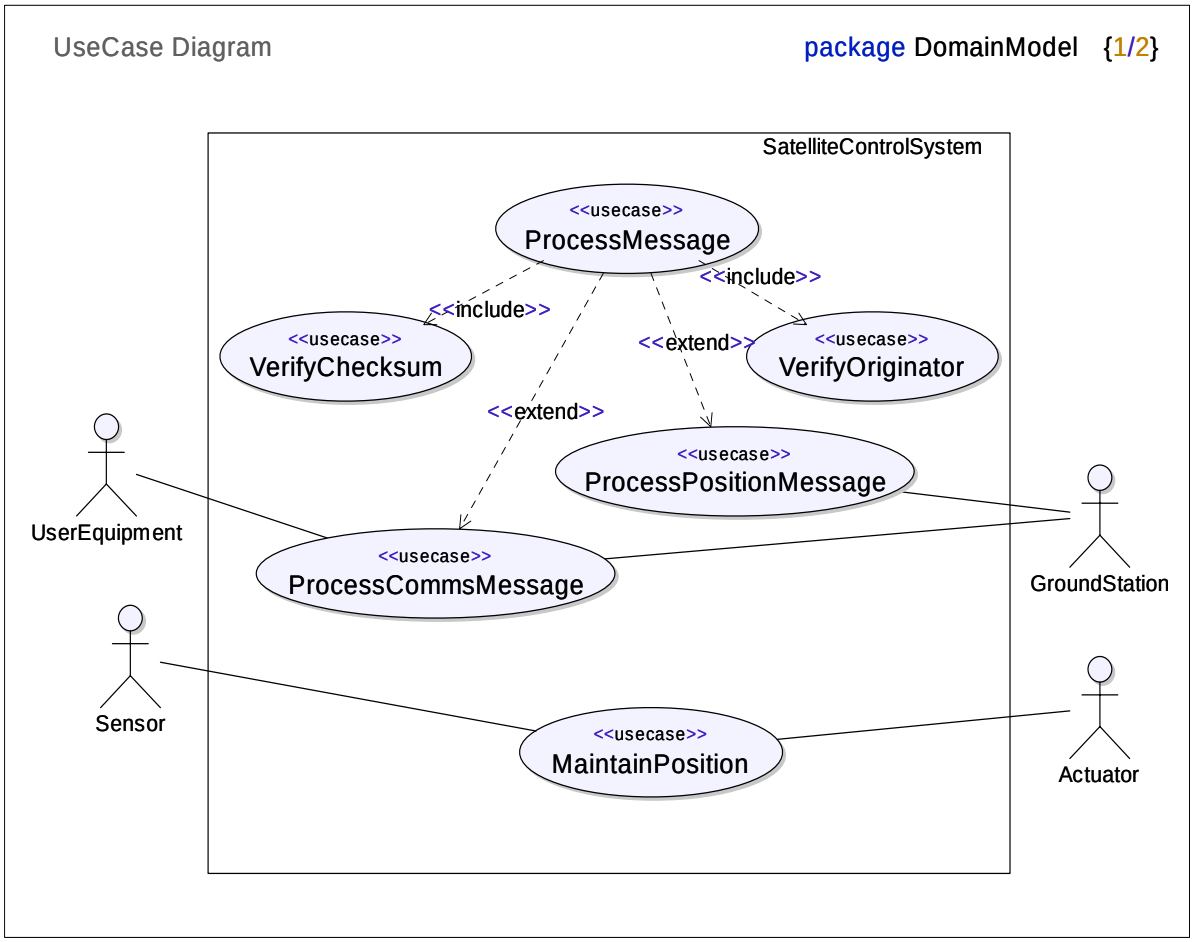


Figure 4 shows how some of the text based requirements previously shown in DOORS can be expressed as UML Use Cases, such as *ProcessMessage* and *MaintainPosition*.

Figure 5: *ProcessPositionMessage* Use Case expressed as Sequence Diagram

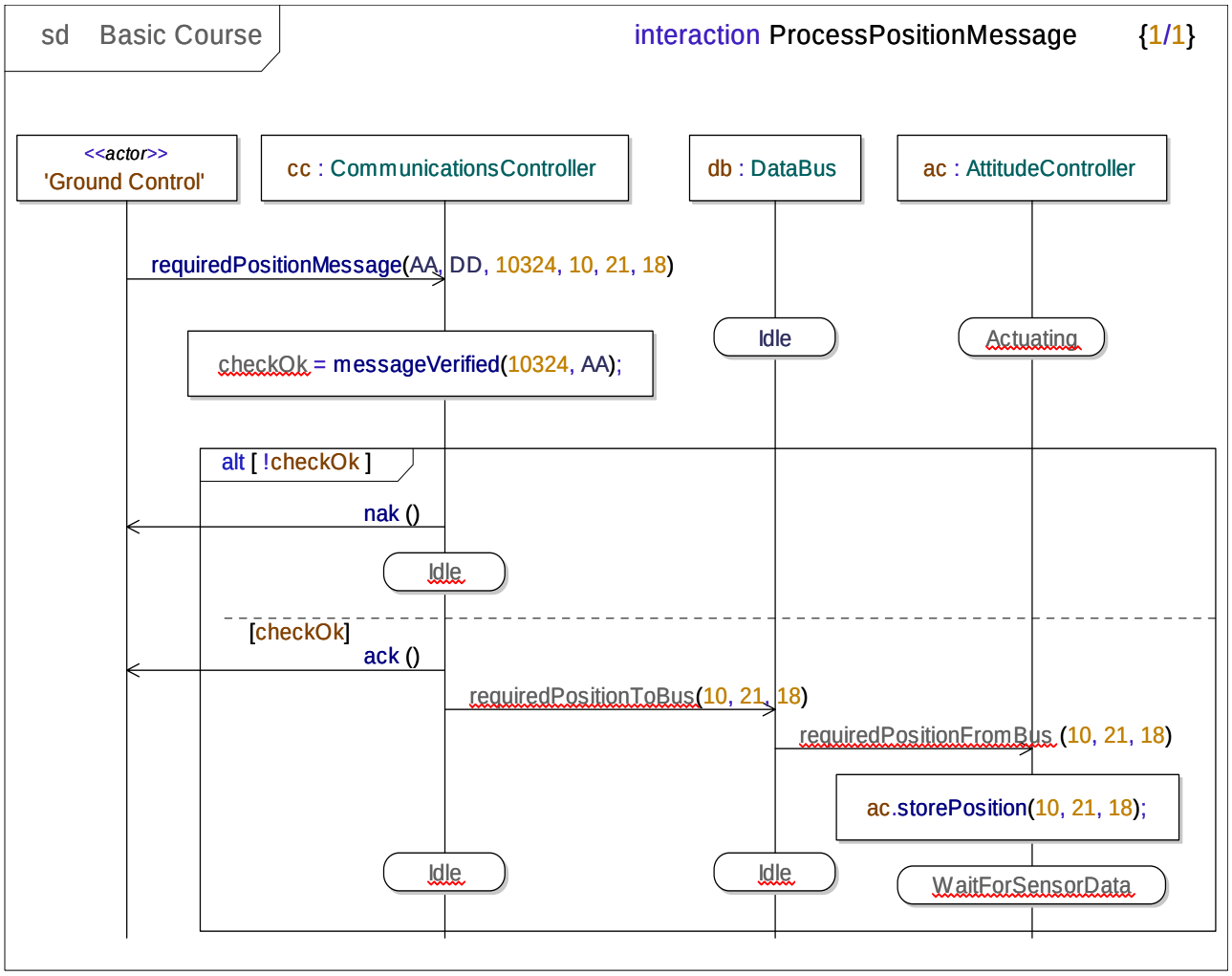


Figure 5 shows the *Process Position Message* Use Case shown in Figure 4 can be specified in detail using a Sequence Diagram. This diagram describes the communications between a *Ground Control* actor and several Satellite Control System parts: *cc:CommunicationsController*, *db:DataBus* and *ac:AttitudeController*.

Figure 6: Composition relationship between *SatelliteControlSystem* Class and its parts

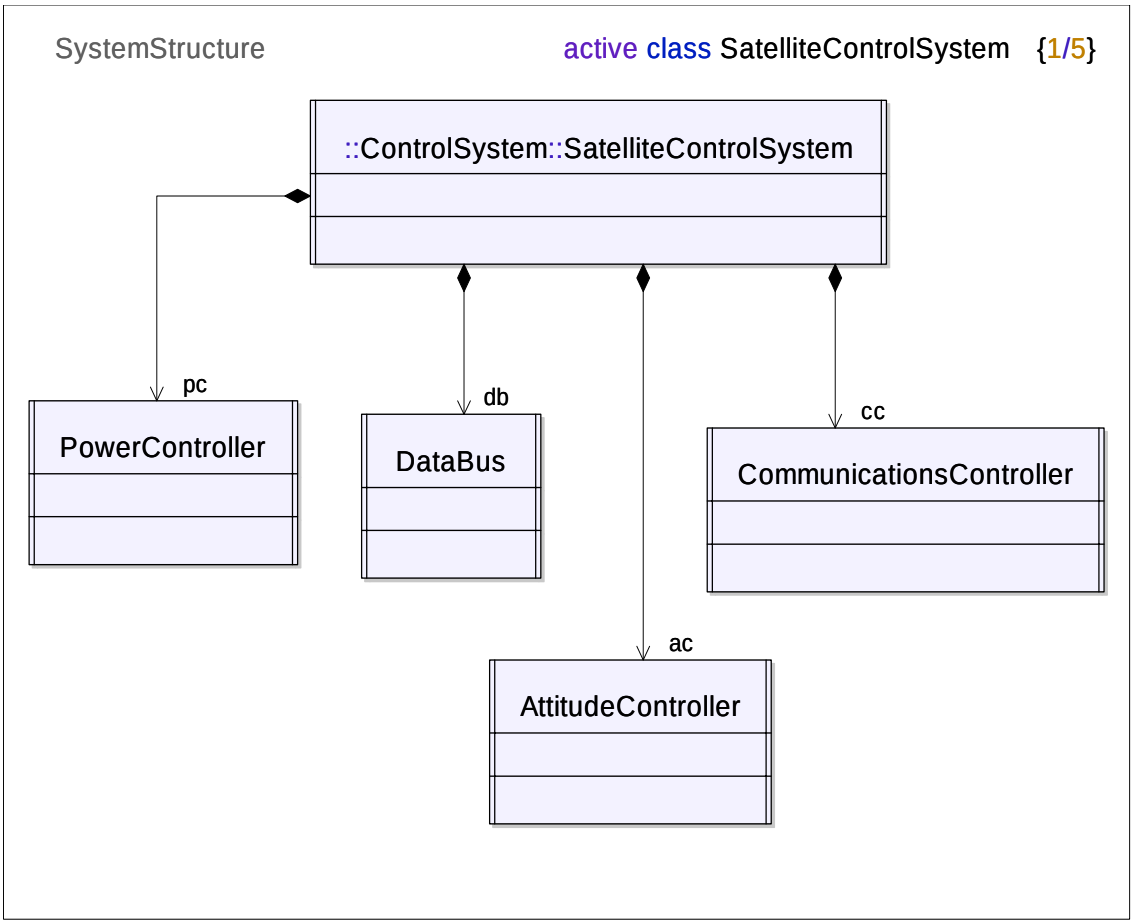


Figure 6 shows a composition (whole-part) relationship between the SatelliteControlSystem class and its constituent parts: *PowerController*, *DataBus*, *AttitudeController* and *CommunicationsController*. This “black diamond” notation for expressing composition has been available since UML 1.x, and is also available in UML 2.0

Figure 7: Internal structure of *SatelliteControlSystem* class

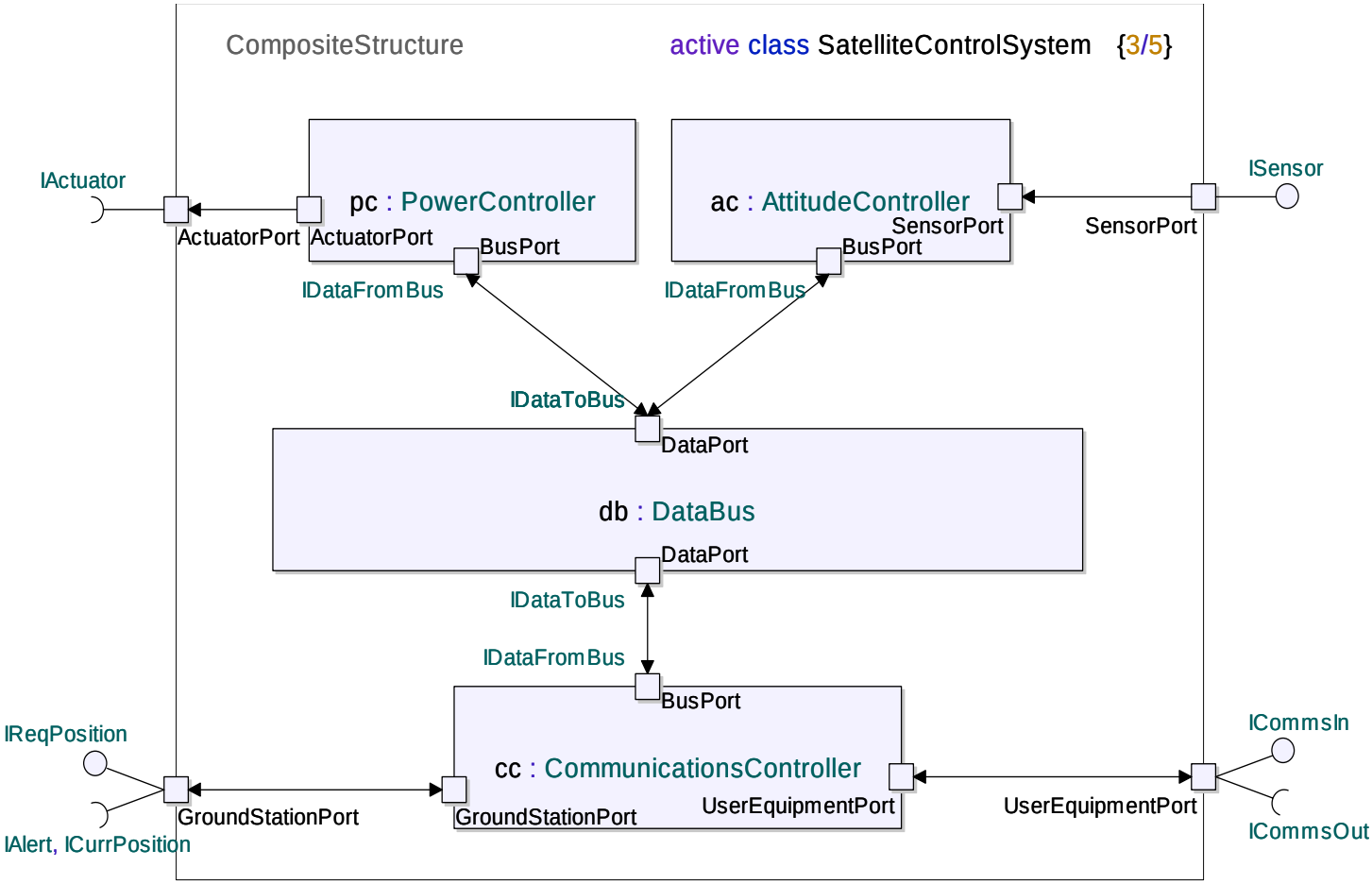


Figure 7 shows a white-box view of the internal structure of a *SatelliteControlSystem* using a Composite Structure Diagram. Composite Structure Diagrams are a new feature of UML 2.0.

Figure 8: Interfaces associated with *SatelliteControlSystem* parts

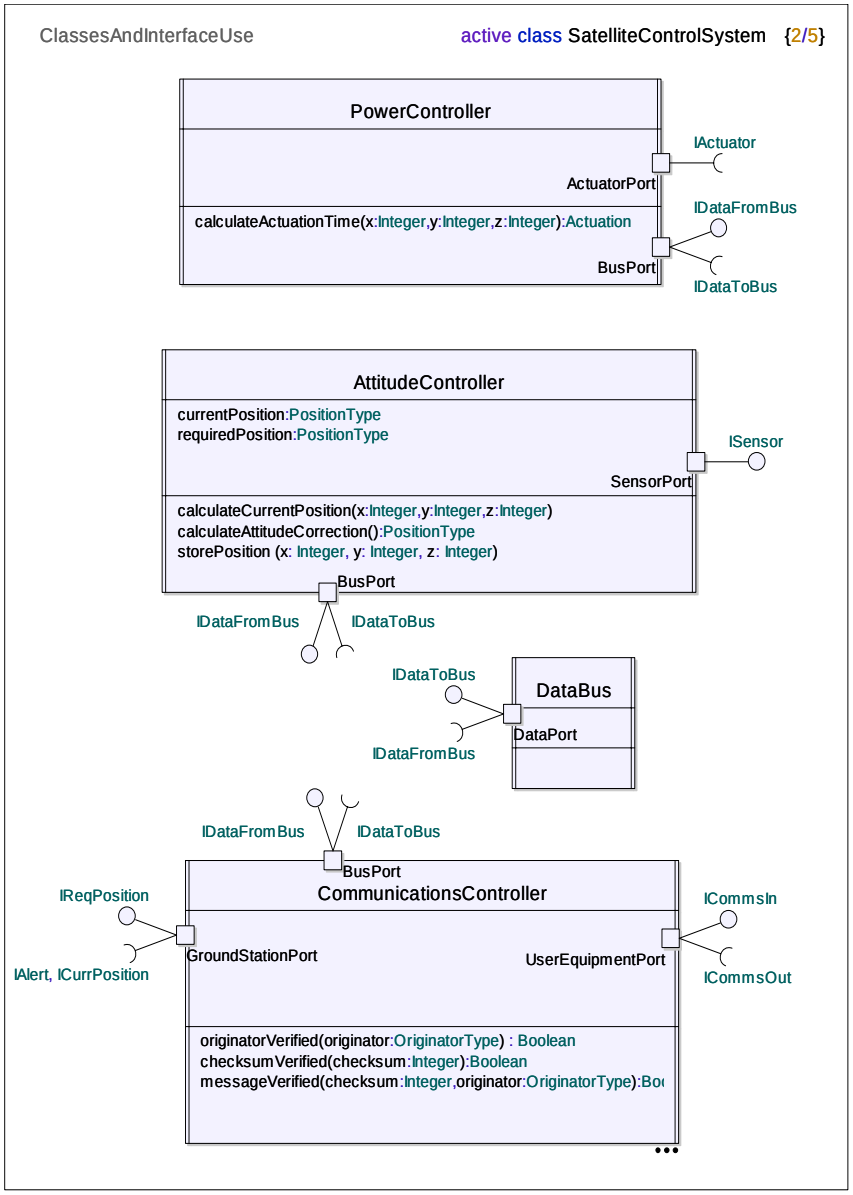


Figure 8 shows the interfaces associated with the part types of the *SatelliteControlSystem* class. Later we will show how these interfaces are used by a CORBA IDL Platform Specific Model to generate IDL (see Figures 11 and 12).

Figure 9: State Machine associated with *AttitudeController* class

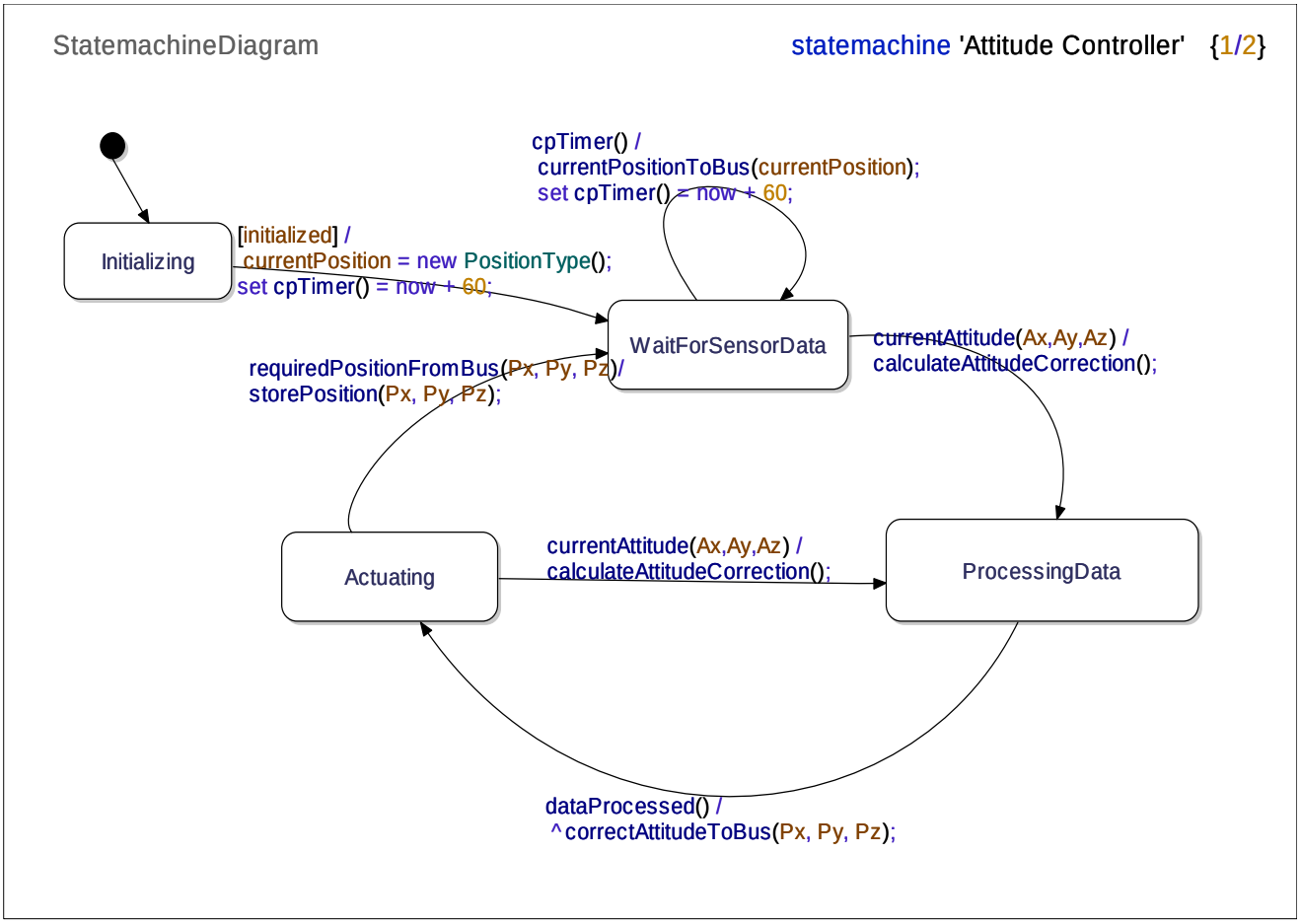


Figure 9 shows a State Machine Diagram for the *AttitudeController* class. This state machine has four states: *Initializing*, *WaitForSensorData*, *ProcessingData*, and *Actuating*.

Figure 10: U2 Action Language Example

```
statemachine initialize {
    start {
        nextstate Initializing;
    }
    state Idle;
    state Initializing;
    state SensorDataCollected;
    state DataProcessed;
    state Actuating;
    for state Idle;
        input currentAltitude() {
            {
            }
            nextstate SensorDataCollected;
        }
    for state SensorDataCollected;
        [dataProcessed] {
            {
            }
            nextstate DataProcessed;
        }
    for state DataProcessed;
        input x() {
            {
                ^ correctAttitudeToBus();
            }
            nextstate Actuating;
        }
    …
```

Figure 10 shows the U2 Action Language for the state machine shown in Figure 9.

Figure 11: Interfaces for CORBA IDL Platform Specific Model

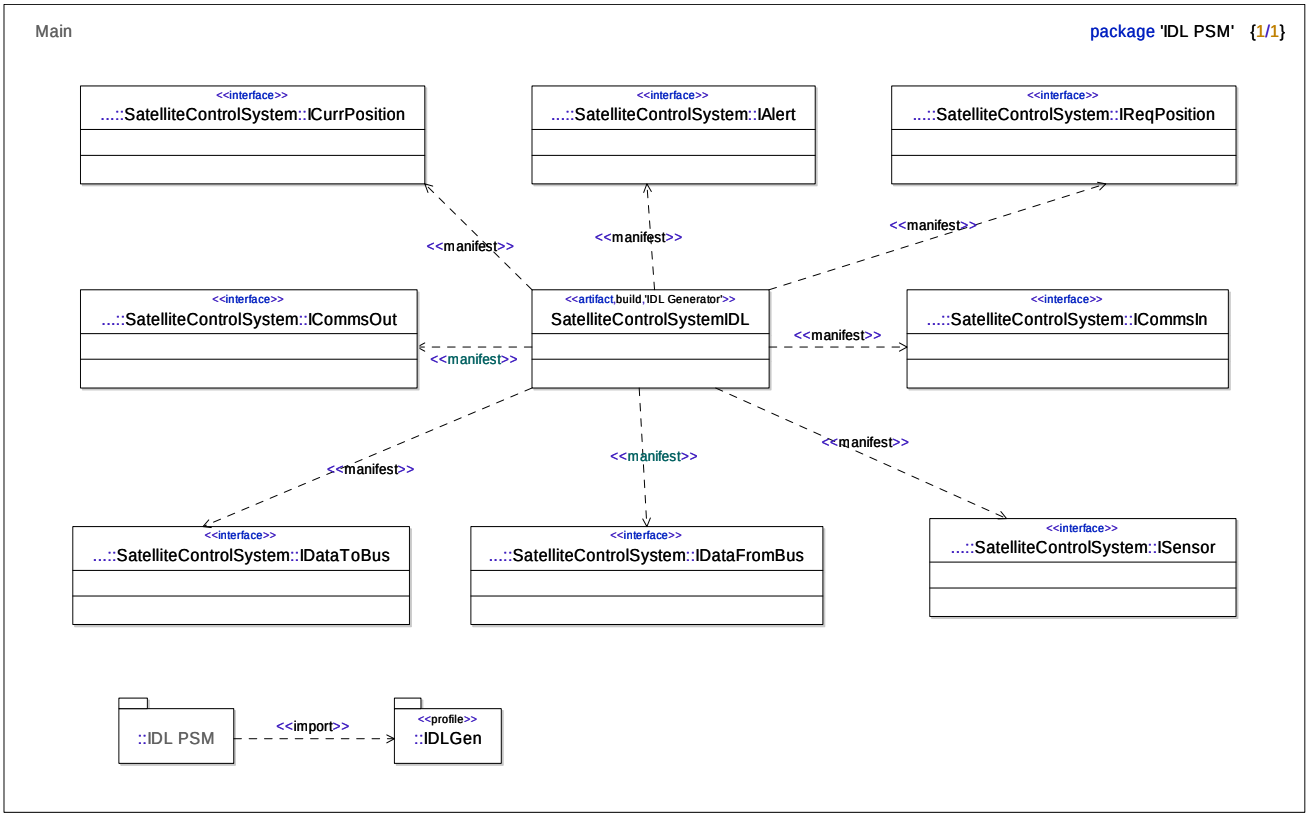


Figure 11 shows the *SatelliteControlSystemIDL* Artifact, which is part of the CORBA IDL Platform Specific Model. The *SatelliteControlSystemIDL* Artifact manifests the SatelliteControlSystem Interfaces by implementing their operations so that IDL code skeletons can be generated (see Figure 12).

Figure 12: Automatically generated IDL interface definitions

```
module SatelliteControlSystemIDL {
	interface ICommsOut {
		oneway void ack();
		oneway void nak();
		oneway void voiceAndDataCommsOut(OriginatorType originator,
			DestinationType destination, Integer checksum, Charstring
			contents);
	}
	interface IDataFromBus {
		oneway void correctAttitudeFromBus(Integer x, Integer y, Integer z);
		oneway void currentPositionFromBus(PositionType position);
		oneway void requiredPositionFromBus(Integer x, Integer y, Integer z);
	}
	interface IDataToBus {
		oneway void currentPositionToBus(PositionType position);
		oneway void requiredPositionToBus(Integer x, Integer y, Integer z);
		oneway void correctAttitudeToBus(Integer x, Integer y, Integer z);
	}
	interface ICommsIn {
		oneway void voiceAndDataCommsIn(OriginatorType originator,
			DestinationType destination, Integer checksum, Charstring
			contents);
	}
	interface ICurrPosition {
		oneway void currentPositionToGS();
	}
	interface IAlert {
		oneway void alert(OriginatorType originator);
	}
	interface IReqPosition {
		oneway void requiredPositionMessage(OriginatorType originator,
			DestinationType destination, Integer checksum, Integer x,
			Integer y, Integer z);
	}
	interface ISensor {
		oneway void currentAttitude(Integer x, Integer y, Integer z);
	}
}
```

Figure 12 shows the CORBA IDL interface definitions automatically generated from the interface model shown in Figure 11.

CONCLUSIONS AND FUTURES

It's inevitable that the software industry will eventually mature, and catch up with other industries based on engineering and automation, such as the computer hardware industry. At some point during this maturation process, it will become common practice for software engineers to specify their products using an architectural blueprint language, such as UML 2.0.

During this evolution it will also become common sensical for engineers to apply a model driven development approach, such as MDA. This approach will need to be supported by power tools, such as TAU G2, that faithfully and efficiently implement the blueprint language, so that it can automate the mapping transformations across the models that represent the various process phases.

What should we expect from Model Driven Architectures during the next decade? We should expect them to evolve from conceptual architectures into technical architectures that solve complex business and technology problems.

What should we expect from MDA tools, such as TAU G2? In general, we should expect progressively tighter integration with traditional Integrated Development Environments (IDEs), and improved integration with requirements management and testing tools. In the case of TAU G2, this means seamless integration with DOORS and TAU/Tester. DOORS requirements can already be visualized as UML elements, and TAU/Tester test scripts are being updated to align it with the recently adopted UML 2.0 Profile for Testing.

These future model driven IDEs will allow developers to efficiently shift and downshift through all the abstraction gears associated with a full application lifecycle. In these high productivity development environments, programming code will likely devolve into a machine readable artifact that is rarely viewed by humans. Released from the drudgery of producing and maintaining low-level implementation code, software developers will be able to pursue more creative activities that return greater business value, such as architecture, analysis and design.

REFERENCES

Publications and Presentations

[Björkander 2003] M. Björkander and C. Kobryn, “Architecting Systems with UML 2.0,” IEEE Software, July/August 2003.

[Kobryn 2003a] C. Kobryn, “Model Driven Engineering with UML 2.0,” presentation, 2003.

[Kobryn 2003b] C. Kobryn, “UML 3.0 and the Future of Modeling,” article to be published, 2003.

[MDA 2003] *MDA Guide, version 1.0, version 2.0*, OMG document omg/2003-05-01. and ad/03-04-01, 2003. [Note: This is a guide, and not a normative specification.]

[UML2 2003] U2 Partners, *UML Infrastructure and Superstructure, version 2.0*, OMG documents ad/03-01-01 and ad/03-04-01, 2003. [Note: OMG Analysis & Design Task Force has recommended both specifications for adoption.]

Web Resources

Information about the U2 Partners’ UML 2.0 submissions for Infrastructure and Superstructure are available at the following Web site:

http://www.U2-Partners.org

Information about Telelogic’s TAU Generation2 product is available at the following Web site:

http://www.TAUG2.com

Information about OMG’s Model Driven Architecture initiative is available at the following Web site:

http://www.OMG.org/mda

ABOUT THE AUTHORS

Cris Kobryn is the Chief Technologist for Telelogic, where he specializes in advanced systems development tools and processes. Cris has applied advanced technologies to solve a wide range of business and scientific problems, and is an expert in distributed software architectures, component-based development methods, and software and systems modeling. He has broad international experience leading high-performance software development teams, and has architected custom applications and commercial products.

Cris is a former Chief Technologist for EDS, and has held senior technical positions at MCI Systemhouse, Harlequin, and SAIC.

As an Object Management Group representative, Cris has been a major contributor to the Unified Modeling Language (UML) specification, which is the industry standard for specifying software architectures. Cris chaired large international standardization teams to specify UML 1.1 and UML 2.0, and serves as the co-chair of the OMG's Analysis and Design Task Force. In recognition of Cris's many contributions to UML and the Analysis & Design Task Force, the OMG presented him with its Distinguished Service Award for the year 2000. Cris is currently chairing a large international standardization team to specify the Systems Modeling Language (SysML) for systems engineering applications (www.sysml.org). He is a member of the IEEE, ACM, AAAI and INCOSE. Contact him at cris.kobryn@telelogic.com.

Eric Samuelsson is a senior software engineer at Telelogic. He is actively participating in the standardization of UML 2.0, and was a major contributor to the UML 2.0 Profile for Testing. Eric specializes in the implementation and customization of model driven development tools. Contact him at eric.samuelsson@telelogic.com.