Domain Driven Simulation Modeling for Software Design

Andrew E. Ferayorni **Hessam S. Sarjoughian**
Andrew.Ferayorni@asu.edu **Hessam.Sarjoughian@asu.edu**

Arizona Center for Integrative Modeling & Simulation
School of Computing & Informatics
Arizona State University
Tempe, AZ 85281-8809

Keywords: Astronomical observatory, design patterns, DEVS, domain specific modeling, software design simulation.

Abstract

System-theoretic modeling and simulation frameworks such as Object-Oriented Discrete-event System Specification (OO-DEVS) are commonly used for simulating complex systems, but they do not account for domain knowledge. In contrast, Model-Driven Design environments like Rhapsody support capturing domain-specific software design, but offer limited support for simulation. In this paper we describe the use of domain knowledge in empowering simulation environments to support domain-specific modeling. We show how software design pattern abstractions extend the domain-neutral simulation modeling. We applied Composite, Façade, and Observer patterns to an astronomical observatory (AO) command and control system and developed domain-specific simulation models for the system using DEVSJAVA, a realization of OO-DEVS. This approach is exemplified with simulation models developed based on an actual AO system.

1 INTRODUCTION

With the rapid growth in complex software-intensive systems, we are faced with the need for capabilities that can reduce their time to market. These systems can benefit appreciably through the use of simulation for analysis during various phases of the system/software engineering lifecycle and especially in the design phase. Simulation provides a powerful approach for dealing with the large number of components and interactions that comprise today's and tomorrow's complex systems since it provides concepts and capabilities that are absent in software analysis and design methods and tools. Software engineering methodologies and tools such as Rhapsody [7] are aimed at developing and validating detailed specifications that are ready for implementation. In contrast, simulation approaches and tools can be used for developing conceptual and architectural design of software-intensive systems. Rather than relying entirely on logical and physical system specifications as in [7] and before entering detailed design followed by implementation, simulation enables evaluation of system architecture behavior and identifying major flaws or shortcomings in a system's architectural specifications in the early stages of the analysis and design phases. Simulation of system architecture produces benefits such as higher quality architectural specifications and reduced costs.

To achieve software architecture simulation we must have the ability to model systems hierarchically and model component behavior. Systems theoretic simulation approaches give us design capabilities such as composition, component connectivity, and time dependent state transitions based on input and output trajectories. Furthermore, Discrete Event System Specification (DEVS), a class of systems theoretic models, provides additional design aspects such as event-based behavior specification and concurrent execution [10]. DEVSJAVA [1], an object-oriented extension of the DEVS formalism, incorporates object-oriented concepts into its simulation modeling capabilities, but by itself does not provide concepts for domain-specific modeling. In order for DEVS to support domain-specific modeling we can extend its modeling capabilities by incorporating design patterns that are appropriate for a given application domain such as AO command and control systems.

In this paper we will describe an approach that accounts for software design patterns in the context of simulation models. We exemplify how general-purpose discrete-event simulation modeling can be extended with the Composite, Façade, and Observer design patters for modeling and simulating an AO command and control system. We present the implementation of the models in an extended DEVSJAVA environment and conclude with a discussion on the benefits of simulation modeling of software design.

2 BACKGROUND

2.1 Software Modeling

The use of object-oriented modeling methods and sound architectural principles (including design patterns) has been well utilized in the software design realm [8]. *Software modeling* emphasizes structural and behavioral specifications of executable software. Models describe conceptual and formal specification of software prior to implementation and testing activities. For example, Statecharts serve as a suitable artifact to describe behavioral blueprint of a system. However, simulating state space of a (hierarchical) Statechart relies on detailed specifications as they were to be implemented.

In recent years software architecture has emerged as a crucial step in the design process of complex software systems. The need for software architecture specifications has brought forth tools and standards for documenting and analyzing them. We have seen the use of simulation in conjunction with architecture specification to produce Executable Architecture Description Languages (EADL) such as Rapide [5]. It is an event-based, concurrent, object-oriented language specifically designed for prototyping architectures of distributed systems. In Rapide, the components of system architecture are defined in terms of their interface connection architecture - defining their ports, constraints on those ports, and connectivity with other components' ports - and their behavior using a partially ordered set of events (POSET). EADLs such as Rapide offer strong support for high level architecture analysis in terms of system components. However, they are limited in their ability to support design patterns.

At the forefront of software modeling techniques is an approach known as Model Driven Engineering (MDE) [6]. A key component of emerging MDE technologies is Domain Specific Modeling Languages (DSML) [2]. The idea behind DSMLs is their ability to define the relationships between concepts in a domain and specify key semantics and constraints associated with those domain concepts. The languages defined by these meta-models account for domain knowledge therefore supporting a declarative approach to modeling design intent. The second key component used in MDE technologies is model transformation. These are transformation engines and generators that analyze aspects of software models in order to support automated mappings to software implementation artifacts. These mappings help to ensure functional and QoS attributes captured in the software models are applied appropriately during implementation [2]. The use of MDE technologies incorporating DSMLs and model transformation in software design is motivated from the standpoint of domain driven software modeling and transition to software implementation. In this regard, simulation is not a primary component and thus the approach is limited to implementation level analysis such as scalability and verification.

2.2 Simulation Modeling

Simulation modeling is concerned with developing model descriptions that can be experimented under artificial settings. Therefore, they need to exhibit dynamics beyond what could eventually be supported by the real system. A central feature of simulation is its support for treating time in logical and/or real-time scales using simulation protocols. Logical time and real-time time are complementary concepts with the former supporting artificially slow or fast passage of time. The importance of manipulating time in simulation is central to simulation models as compared with software models.

Systems theory provides us with the ability to define a system in terms of its structure and behavior. The structure and organization of system components is modeled hierarchically, whereas the behavior of system components can be modeled in continuous or discrete time. Components can be configured with input and output ports, which when connected to the ports of other components, allow interactions between them.

Discrete-Event System Specification (DEVS) is a class of system theoretic models which support the modeling of hierarchical interacting components that can exhibit autonomous and reactive based behaviors. System structure and behavior are captured with atomic and coupled models. Parallel atomic models allow for multiple ports that can accept bags of inputs and produce bags of outputs. Parallel coupled models can consist of any number of atomic and coupled models but must i) consist of atomic models at the lowest level of any coupled model; ii) no coupled model can contain itself; and iii) output to input port coupling resulting in direct feedback is not allowed for atomic and coupled models.

However, the modeling capabilities of systems theory do not support some important software design techniques. For example, UML's classifiers (e.g., Interface) and relationships among classifiers (e.g., inheritance, dependency, and realization) are not supported. Without these kinds of software specification abstractions, it is difficult to specify simulation models of complex software designs that include design patterns and thus explicit support for domain-specific simulation modeling.

2.3 Application Domain

During this research we contacted individuals from different institutions that were responsible for the design of command and control software for astronomical observatories (AO). Braeside Observatory [3], an observatory supported by Arizona State University's Department of Physics and

Astronomy, was one of the designs we reviewed. The approach taken in designing the Braeside system did not involve any simulation based analysis, nor did it utilize current software design techniques such as object orientation or design patterns. The reason these techniques are not used is primarily due to the desire to reuse legacy software that has been proven to work well for control of other observatories. Although simulation did not play a role in the design of the Braeside system, we have seen its use in support of a toolbox for AO systems simulation. The VLTI end to end simulation package [9] provides a logical framework for developing models of an optical telescope. This framework supports modeling the physical components, including control software, that comprise the observatory system. The primary purpose is to evaluate how the components of the observatory will work together in an effort to determine the type of results that can be obtained using the system. Although the toolbox provides a framework for system simulation that includes the control software, it does not attempt to capture architectural details of the control software for the purpose of design analysis.

3 APPROACH

In this work, we introduce design patterns [4] into discrete-event simulation modeling for the AO domain. Instead of solely using *systems theory* and *object-orientation* for specifying simulation models, we also use *design patterns* to capture important traits of common solutions for building command and control systems for AO. This kind of simulation modeling provides principled use of design patterns as applied to this domain. As shown in Figure 1, design patterns specifically suitable for the AO application domain can be added to the DEVSJAVA simulation modeling environment. We have extended the OO-DEVS with a select set of design patterns to enable specifying and simulating *domain-specific simulation model components*. The result of the extended OO-DEVS is a collection of simulation model components where relationships among them include patterns of interaction and dependency beyond *whole-part* and *is-a* relationships which have formal underpinnings in OO-DEVS (see Section 4 for the details of design patterns used in AO domain). For the AO domain we have developed DEVSJAVA-AO which extends domain-neutral DEVSJAVA structural and behavioral modeling constructs with the AO dynamics. The result is a domain-specific simulation environment which can be used to develop specialized simulation models and evaluate alternative AO system architectural and design specifications. Specifics and detailed models of the DEVSJAVA-AO are described in Section 5.

This approach allows developing prominent features of an application domain on top of the general-purpose capabilities of a modeling and simulation environment. There are a number of benefits. First, modelers can take advantage of design patterns to develop domain-specific simulation models. A simulation environment which has built-in design patterns, for example, helps simulating rich dynamics of AO command and control software. Second, since design patterns are incorporated into simulation model components, they can support simulating "software architecture" without first developing UML models which are close to actual detailed designs. Third, basic differences between simulation and software models can be bridged in a logical fashion since high-impact architectural specifications can be evaluated via simulation instead of delaying them until detailed design, implementation, and testing phases.

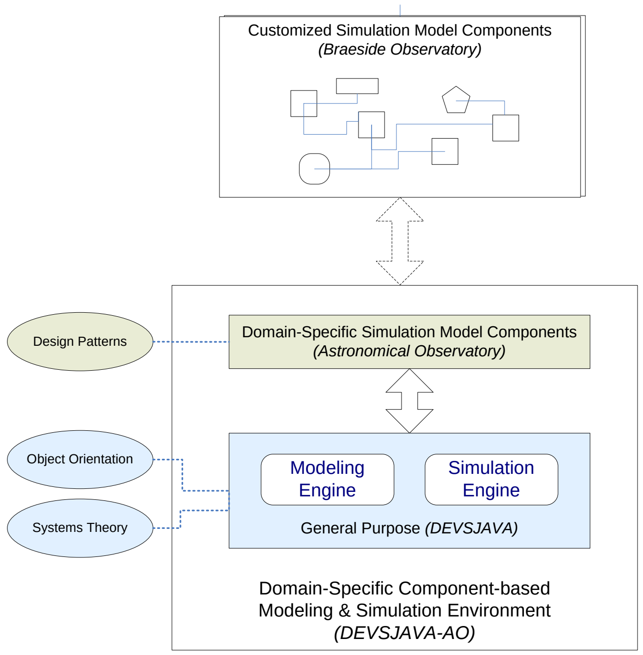

Customized Simulation Model Components
(Braeside Observatory)
Design Patterns
Domain-Specific Simulation Model Components
(Astronomical Observatory)
Object Orientation
Systems Theory
Modeling Engine
Simulation Engine
General Purpose (DEVSJAVA)
Domain-Specific Component-based Modeling & Simulation Environment
(DEVSJAVA-AO)

Figure 1: A conceptual view of simulation model components for an AO command and control system.

Consequently this approach can help with reuse of "solution" simulation models for developing software design models which in turn should lead to improved time to market and increased quality of the end software/system product. The main benefit of design patterns, therefore, is the ability to create *simulatable software architectures* and *designs*.

4 SIMULATION MODELING WITH DESIGN PATTERNS

In the previous sections we discussed how the use of system-theoretic and object-oriented approaches supports modeling of complex distributed software systems. We also looked at the benefits of considering system software design goals when we are developing the simulation model design. In this section we will show in detail how to develop an extension of the simulation environment with design patterns specific to a domain.

4.1 Astronomical Observatory Systems

An Astronomical Observatory (AO) command and control system is complex and well suited for examining the pattern based simulation model design methodology. Typically these systems consist of software components for controlling features of the telescope, detector, and mount. Each of these will perform the functions necessary to interface with the user and the physical system components. The mount control software will be responsible for processing requests to point the telescope to specific coordinates as well as tracking those coordinates over time. The telescope control software will be responsible for handling requests to control telescope accessories, such as adjusting the position of a focuser. Finally, the detector control software will execute requests for taking images and downloading image data. Communication between software components is necessary to coordinate certain system functions.

4.2 Domain Analysis

Identifying domain-specific design patterns for system software requires domain expert knowledge. This knowledge helps us understand what is common among systems in the domain in terms of the structure and interaction of their components. Since there are many different ways to architect a system we cannot attempt to identify patterns that will work for all systems in a domain. Instead we focus on those patterns that can be applied to most systems in the domain. In the next few sections we will discuss different approaches to analyzing the domain and identifying these patterns.

4.3 System Functionality

An excellent approach to collecting requirements for the functionality of a system can be achieved through generation of use case diagrams. Use case diagrams allow us to capture the scenarios (use cases) under which the system will be used from the perspective of its users (actors). Each use case that is generated represents a goal that the user wants to achieve with the system. These goals help model the requirements of the system by specifying what is expected of it. However, these requirements should not specify the details of how the system will achieve these goals.

Once we have developed a strong set of use cases for our system we can easily see the high level functions expected of it. This information naturally maps to another object-oriented concept known as interfaces. With interfaces we can map the expected system functions into collections of operations that can be realized by models in our implementation. For example, an interface for a detector controller would specify the method signatures that when realized can be used for interacting with the detector sub-system. The use of interfaces in design is powerful because it allows us to show what functionality is expected (syntheses, interaction, and collaborations) but not how that functionality will be achieved. The method signatures for each operation of an interface will specify what inputs are to be given and what outputs are expected. The details of how interface operations will be implemented are left to the models that realize them.

The concept of interfaces maps nicely into our first design pattern for the AO domain. The Façade design pattern provides a unified interface to a set of sub-interfaces. AO control systems are comprised of many sub-components that work together to complete a user request. Once we have identified interfaces that capture the services provided to the system users, we can combine all or part of those interfaces to create a Façade. This higher level interface will hide the details of how sub-system components are used to execute the request. In addition, changes to how the sub-system components carry out the request can be made without impacting the user of the Façade. This is an important pattern for the AO domain because instrumentation is frequently upgraded to stay atop research needs and evolving technologies.

In Figure 3 we show the use of an DetectorControllerInterface (DCI) in support of the Façade pattern between an ObservatoryClient (OC) and the AbstractDetectorController (ADC). This component of the system may be implemented with one software component (as shown in this example), or with coordination of many sub-components. Therefore applying the Façade pattern allows us to hide the details of how the interface methods are actually carried out.

4.4 System Structure

When we look to identify design patterns we must consider the *structure* and *interaction* of components in the system. Through the use of domain knowledge we can study these systems and look for patterns in how they can be modeled. Patterns that are more common in the domain will become the focus of the design pattern selection.

When we analyze the structure of the system components we have to look for common patterns in how they are organized horizontally and vertically. Horizontally we look to identify the major nodes of the software architecture that will consist of one or more components working together to produce some behavior. These nodes often separate major functionalities of the system that generally operate independently. Some interaction between nodes may be necessary but is usually kept to a minimum. Beneath these nodes is where we look for vertical organization patterns, where emphasis is placed on how the patterns will support *modification* to adapt to different modeling configurations.

Another driving force in the structure of a system is how its components interact. This involves understanding how control and data will flow in and out of each component as well as what other components it flows to and from. Areas where various configurations are desirable and generic configurations cannot be achieved will be considered when designing for *integrability*.

Analysis of the AO domain in terms of component structure and interaction revealed two significant design patterns. The first is the *composite* design pattern, which allows for the composition of objects into structures that represent part-whole hierarchies. This pattern can be used in the AO domain to support the use of common I/O channels for components communicating within a tree structure. For example, all software components that comprise the mount controller module can provide a common I/O channel structure such as command input, command output, data input, and data output. This allows simplified identification of where command and data information can be obtained and delivered between components of the same tree.

The composite pattern is naturally supported by the component modeling capabilities of systems-theory and DEVS. The pattern can be specialized for the AO domain to support command and data I/O channels by using two classes, CompositeControlElement (CCE) and PrimitiveControlElement (PCE) as shown in Figure 2. These classes support the composite design pattern in its ability to be a parent or leaf node in a hierarchical modeling structure. These classes also extend the DEVSJAVA ViewableDigraph class, which extends DEVS coupled component modeling with visualization abilities.

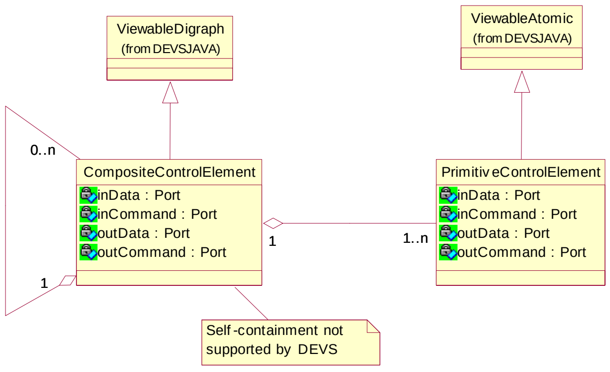


Figure 2: Composite design pattern for AO domain.

The second design pattern identified is the *observer* pattern which allows for components (the observers) to be notified when the state of other components (the subjects) change. This pattern is also referred to as Publish-Subscribe or Subject-Observer. This pattern is important for the AO domain because subject component state changes are often shared with many observing components that may vary from one system configuration to another. For example, when the detector is taking images the mount will need to block any incoming slew requests from the user. Similarly when the detector is finished taking an image the mount will need to unblock. This common coordination between the control software of the mount and detector can be managed via the observer pattern. Furthermore, a new system configuration may introduce a second detector that also needs to be observed by the mount. In this case the pattern supports the client subscribing the new observer to the subject through well defined interfaces. Figure 3 shows how the observer pattern interfaces can be used by two atomic components representing the mount and detector control software.

5 SIMULATION MODEL IMPLEMENTATION

Implementation of our simulation models will require an environment that supports object orientation and provides the ability for extension of the core components with design patterns. Commercial Off The Shelf simulation packages generally do not allow access to core components of the environment, and therefore are not well suited for extension with design patterns. Simulation packages such as DEVSJAVA support modeling using object orientation and also allow for extension of core environment components. For our implementation we extended the DEVSJAVA environment with our AO domain design patterns, and utilized these patterns to implement models of a simple observatory control system. In the following sections we will look at implementation details and discuss challenges faced. We will also present simulation results obtained from an experiments conducted with fully implemented models.

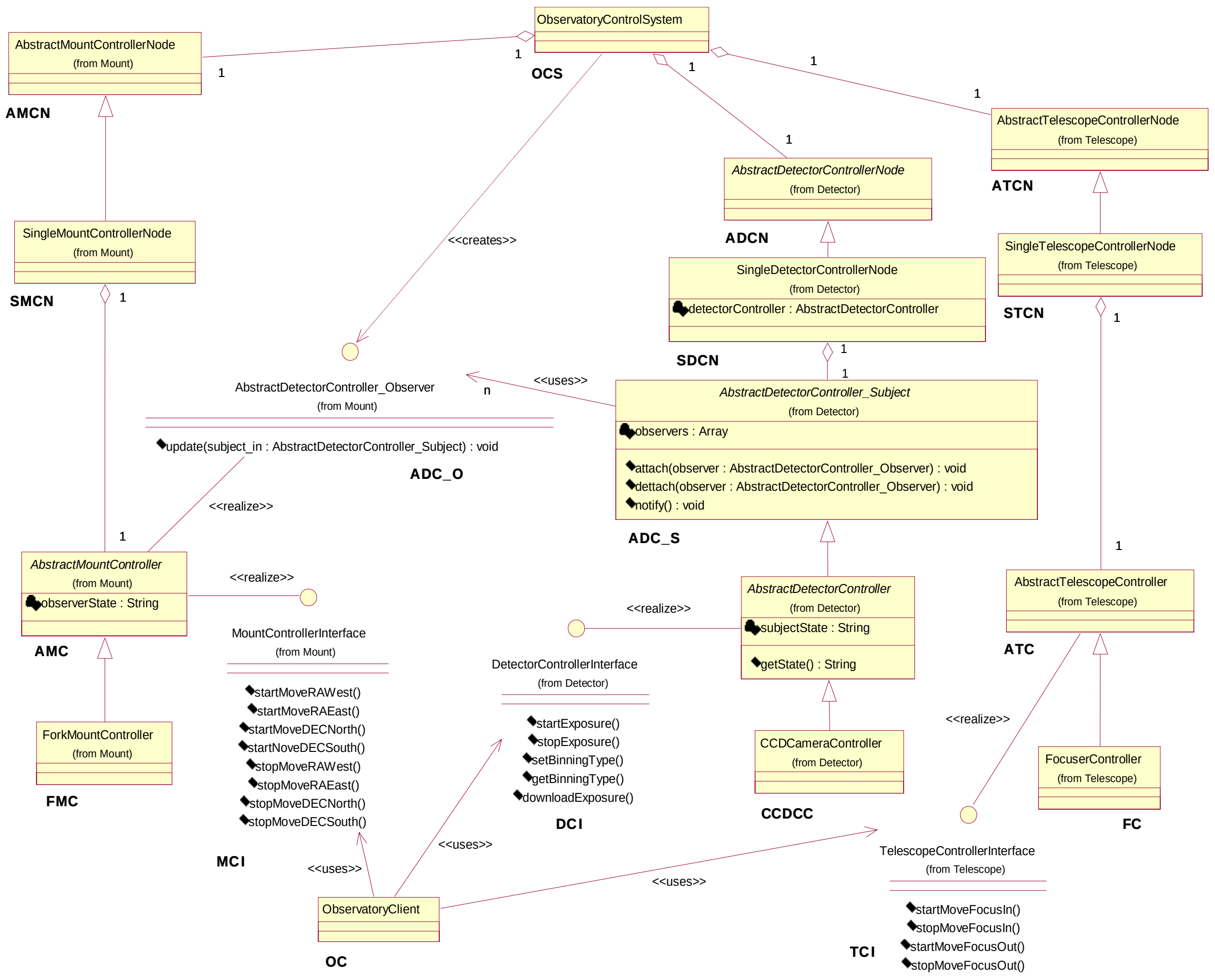
5.1 DEVSJAVA-AO Implementation

The three major software components of an AO control system are those controlling the mount, telescope accessories, and detector instruments. Using the composite design pattern we can define three observatory nodes. They are AbstractMountControllerNode (AMCN), AbstractTelescopeControllerNode (ATCN), and AbstractDectectorControllerNode (ADCN). Each of these is an abstract class extending the CCE abstract base class from Figure 2, thus inheriting its generic I/O port structure. These nodes serve as parent nodes in the composite structure, thus allowing any number of additional parent and child nodes to reside below them. In addition, they can be specialized to capture details of a specific type of configuration that is needed. For example, the control system for Braeside Observatory has a single software component for controlling a CCD Camera detector. We can represent this in DEVSJAVA-AO by specializing the

ADCN with SingleDetectorControllerNode (SDCN). This specialized class will ensure that only one detector controller (coupled or atomic) can exist within the node.

Figure 3 shows our implementation of simulation models for a simple AO control system. In this system the node level abstract classes are specialized using concrete classes that allow each to have a single controller. Each controller node can therefore contain one atomic model (controller) realizing all the associated Façade interface methods. For example, the SMCN consists of one atomic model named AbstractMountController (AMC) that contains methods realizing the MountControllerInterface (MCI). Under a different observatory configuration, we may have modeled the mount controller using multiple components. Because the ObservatoryClient uses the MCI Façade it would not be impacted by the change in the number of mount control components supporting that interface. The ForkMountController (FMC) class shows how we can further specialize the AMC class with details specific to how a controller for a fork type mount would implement the interface methods.

The observer pattern is utilized for state change notification between the mount and detector. In our simple system we have chosen to have a single controller for the detector, represented by the AbstractDetectorController (ADC) atomic model. This class can act as the subject by inheriting from the AbstractDetectorController_Subject (ADC_S), thus providing subscribed observers access to its state. The mount controller is also represented with a single atomic model class named AbstractMountController (AMC), which can be setup as an observer of the detector controller because it realizes the AbstractDetectorController_Observer (ADC_O) interface.



ObservatoryControlSystem
OCS
AbstractMountControllerNode (from Mount)
AMCN
AbstractTelescopeControllerNode (from Telescope)
ATCN
AbstractDetectorControllerNode (from Detector)
ADCN
SingleMountControllerNode (from Mount)
SMCN
SingleDetectorControllerNode (from Detector)
detectorController : AbstractDetectorController
SDCN
SingleTelescopeControllerNode (from Telescope)
STCN
<<creates>>
<<uses>>
n
AbstractDetectorController_Observer (from Mount)
update(subject_in : AbstractDetectorController_Subject) : void
ADC_O
AbstractDetectorController_Subject (from Detector)
observers : Array
attach(observer : AbstractDetectorController_Observer) : void
dettach(observer : AbstractDetectorController_Observer) : void
notify() : void
ADC_S
<<realize>>
AbstractMountController (from Mount)
observerState : String
AMC
MountControllerInterface (from Mount)
startMoveRAWest()
startMoveRAEast()
startMoveDECNorth()
startNoveDECSouth()
stopMoveRAWest()
stopMoveRAEast()
stopMoveDECNorth()
stopMoveDECSouth()
MCI
AbstractDetectorController (from Detector)
subjectState : String
getState() : String
DetectorControllerInterface (from Detector)
startExposure()
stopExposure()
setBinningType()
getBinningType()
downloadExposure()
DCI
AbstractTelescopeController (from Telescope)
ATC
ForkMountController (from Mount)
FMC
CCDCameraController (from Detector)
CCDC
FocuserController (from Telescope)
FC
TelescopeControllerInterface (from Telescope)
startMoveFocusIn()
stopMoveFocusIn()
startMoveFocusOut()
stopMoveFocusOut()
TCI
ObservatoryClient
OC
1

Figure 3: Simulation models for simple AO control system.

Communication between components in the simulation is done via message passing. To simplify this we have defined the ObsMessage class for presenting all the required message data. This class provides member variables and methods for storing and retrieving the specifics of the message. The benefit to having a common message format is reduced complexity in creating and processing messages throughout the system.

5.2 Simulation Using DEVJAVA-AO

The DEVSJAVA-AO environment provides us with the base classes necessary to build atomic and coupled models for a simple AO control system. In addition, it provides classes for basic block diagram visualization of simulation executions. In Figure 4 below we show the DEVSJAVA-AO simulation view for our implementation. The ObservatoryControlSystem (OCS) coupled model serves as the top level node and contains all models in the AO control system implementation. The OC model, which is also specified in the DEVSJAVA-AO environment, stimulates the OCS by sending ObsMessage type messages as input commands over time trajectories. Output data and commands are gathered during many time periods. These simulation results are evaluated to choose suitable command and control designs under a range of operational settings.

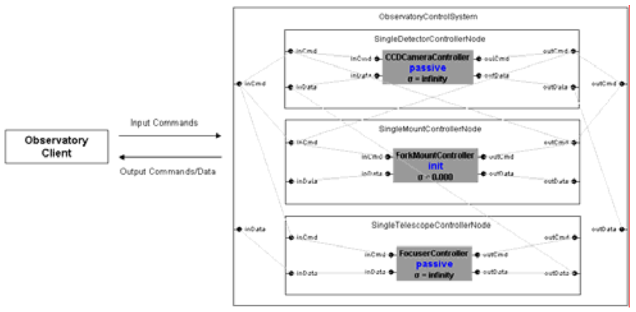


Figure 4: Astronomical observatory simulation models.

5.3 Simulation Results

Simulation provides us with the ability to run experiments on a system and learn about its behavior under certain conditions. Our models of a simple AO control system can be simulated in various experiments to measure many aspects of the system such as correctness, performance, and "what if" scenarios. Extending the DEVSJAVA environment with the DESVJAVA-AO library enforces use of domain specific design patterns in our simulation modeling, but it should not impair our ability to perform DEVS simulations.

One interesting aspect of software architecture that we can test is how addition of a new module will impact the rest of the system. For example, one observatory configuration may require two detectors to be used instead of one. A CCD camera detector will be used to capture data from the visible light spectrum over time while a spectrograph detector will be used to record the individual spectrums of the light. As we discussed before, the mount controller will subscribe to state changes in the detectors via the observer pattern. Therefore, when adding a new detector to the system we immediately see how well the pattern supports a new configuration. In addition, we can run simulations to see how the state changes of the new detector controller impact the behavior of the mount controller.

Another aspect of software that we can test is the behavior of the system when differing algorithms are used to perform certain system functions. For example, when an observation request is received the system must determine if the request is valid. There are two algorithms that can be used to determine whether to accept or reject the request:

1. Check only if the object is currently in view

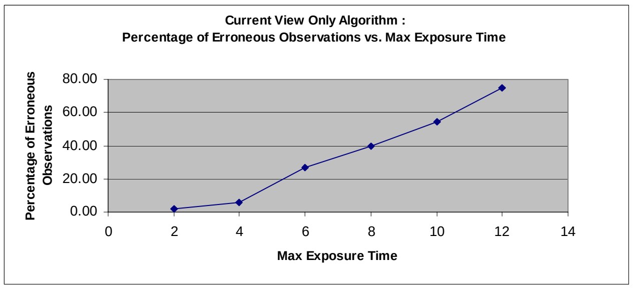
   With this algorithm the mount software will simply check that the coordinates of the observation request are currently in the area of the celestial sphere that the telescope can be pointed to. If they are, the request is accepted and the observation begins. If they are not, then the request is rejected.

2. Check if the object is in view now and that it will be in view for the length of the observation

   This algorithm is more advanced than the first in that it does one additional check. It will not only ensure that the coordinates are obtainable now, but also that they are available for the entire length of the exposure. This means that the object we are tracking will no go below the horizon before the exposure has concluded.

To conduct the experiment we create two versions of the ForkMount atomic model, one that implements the first algorithm and another that implements the second. To see the impact this algorithm change has on the system we conduct simulations in which observation requests generated by the experimental frame are created by selecting random coordinates and a random exposure time. The exposure time is limited by a maximum exposure time (MET) value that we increase by two hours with each simulation run (i.e. 2, 4, 6, 8, and 12). A successful observation request is one in which the coordinates of the object being imaged can be tracked by the telescope from start to finish of the exposure. This scenario is considered to return the desired image. A failed observation request results if the exposure is cut short because the object goes below the horizon. This scenario is considered to return an erroneous image.

The same set of observation requests are fed as input to the models for each MET, once for algorithm 1 and a second time for algorithm 2. This will allow us to see how many successful observation requests occur given the chosen decision algorithm and allowed MET. The first chart in Figure 5 shows the percentage of observations that were erroneous as the MET was increased. Since algorithm 1 only checks that the object is currently in view when the request is received, we start to see more erroneous exposures as the MET increases. The second chart in Figure 5 shows the percentage increase in the number of observations that were successfully carried out using algorithm 2 versus algorithm 1. Here we can see that as the MET increases, we start to see more benefit in using algorithm 2. This is due to an increase in the number of long exposure requests that are received as the MET is increased. Overall the results show us that the second algorithm is a better choice because it eliminates requests that will be erroneous because the object is going out of view before the end of the exposure.



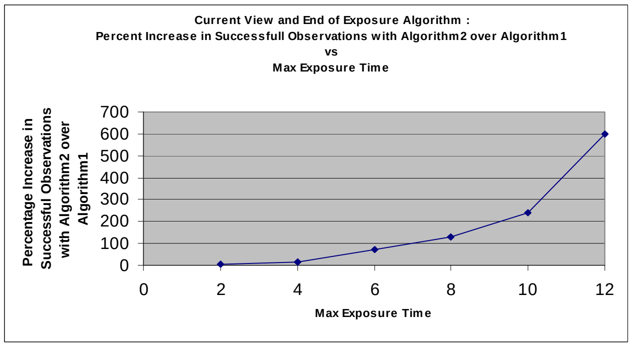


Figure 5: Simulation results using two alternative control algorithms.

6 Conclusion

The motivation behind this work is Simulation Based Acquisition (SBA) which promotes systematic use of simulation across lifecycle of systems from conception to retirement. In this respect, the presented approach focuses on supporting simulation-based software design. This work shows the use of design patterns in support of command and control paradigm for the software development of astronomical observatory control systems using DEVSJAVA-AO. The inclusion of design patterns in a modeling and simulation environment for specific domains plays a significant role in creating software design that can be simulated prior to detailed software design specification, with a key benefit being reduction in the overall software development effort. A future direction for this research is applying the simulation models for developing software controlling an astronomical observatory. Another future research opportunity involves forward engineering from simulation models to software models. A related area of interest is the inclusion of design patterns in real-time simulation modeling.

7 References

[1] ACIMS, *DEVSJAVA*, http://www.acims.arizona.edu, 2006.

[2] Balasubramanian, K., A Gokhale, G. Karsai, J. Sztipanovits, S. Neema, 2006, "Developing Applications Using Model-Driven Design Environments", IEEE Computer, Vol. 39, No. 2, pp. 33-40.

[3] Braeside Observatory, Arizona State University, http://braeside.la.asu.edu, 2005.

[4] Gamma, E., R. Helm, R. Johnson, J. Vlissides, 1995, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley.

[5] PAVG, *Rapide*, http://pavg.stanford.edu/rapide/, 1998.

[6] Schmidt, D., 2006, "Model-Driven Engineering", IEEE Computer, Vol. 39, No. 2, pp. 25-31.

[7] Telelogic, Rhapsody, http://modeling.telelogic.com/ modeling/products/rhapsody, 2007.

[8] UML, http://www.omg.org/cgi-bin/doc?formal/05-07-04/, 2006.

[9] Wilhelm R.C., B. Koehler, 2000, "Modular toolbox for dynamic simulation of astronomical telescopes and its application to the VLTI", SPIE, Vol. 4006, pp. 124-135.

[10] Zeigler, B.P.; H. Praehofer; T.G. Kim. 2000. *Theory of Modeling and Simulation*, 2nd Ed. Academic Press.