System Validation and Verification Using SDL

Ron Henry

Class Project Report

ENSE 623

December 5, 2004

Table of Contents

List of Figures

1 Introduction

1.1 Abstract

This class project involves the application of model-based systems engineering techniques to automation of system validation and verification (V&V) activities. It identifies a conceptual framework for automated V&V that is supported by contemporary commercial tools. The applicability of the framework is then demonstrated through a small but realistic case study. A commercial tool is used to define a formal model for the system described in the case study, generate an executable simulation of that system, and use the simulation to validate the model against high-level use cases and to convert the use cases into formally specified test cases.

1.2 Organization

This report is organized as follows. Section 1 introduces the topic of automated V&V, defining terminology and discussing the potential benefits. Section 2 describes the conceptual framework used for this project. This framework is based on a collection of standards for complementary notations used to represent models and test cases which are supported by commercial tools. Foremost among these notatons is the Specification and Description Language (SDL). Since my interest in these techniques is primarily application oriented, this section includes a proposed methodology intended to guide practitioners in applying these techniques to real systems, and a brief survey of available tools. At the same time, this project began with a study of a Ph.D. thesis by Burton[1] involving the use of logic-based specification languages for automated V&V. An attempt is made to understand what was learned from that research and compare Burton's approach to the SDL approach.

Section 3 introduces the project case study, which involves a system architecture for a remotely controlled telescope, and documents the tool configuration used for this project. Section 4 applies the methodology from section 2 to this application, beginning with a definition of system context and gradually developing a detailed SDL model. At the end of this section, a list of problems encountered during the modeling work is documented for future reference. Section 5 documents how the model from section 4 was used to apply automated V&V techniques to the case study. Screen shots and execution traces are used to demonstrate the results of simulation, model checking and test case generation. Section 6 gives some suggestions for follow-up work.

1.3 Terminology

Since there is widespread confusion on the meaning of the V words and they are not always used to refer to the same thing, it is prudent to begin with the usual definitions. System validation refers to checking a body of requirements or other technical data (such as a design) to ascertain compliance with higher-level requirements or stakeholder needs. System verification refers to checking that a delivered system meets its requirements, usually by testing (although other verification methods are employed where testing is costly or impossible). Note that the validation activity (as defined here) must precede verification, as pointed out in O'Grady.[2] Unfortunately, in standard definitions of V&V the term "verification" is given first, which probably contributes

to the confusion on this topic; for this reason (following O'Grady) I have inverted the order in the title of this paper.

1.4 Motivation

Both validation and verification are currently manual processes in most systems engineering (SE) environments, and therefore slow and prone to human error. Automating these tasks holds the potential to accomplish them more reliably and with less effort. The three major project managment variables — risk, cost and delivery time — are often in conflict, but with regard to V&V, improved technology raises the prospect that all three may be improved at the same time ("better, faster, cheaper").

A system architecture refers to the decomposition of a system into components (which may be systems in their own right) and a specification for how the components communicate. Automated validation of system architectures is feasible if the architectures can be formalized to the point where the external behavior of the system can be predicted and compared to a model of expected behavior from the original requirements. Large systems are often developed top-down in layers, with the architecture of one layer giving rise to requirements at the layer below. Therefore, validating an architecture against higher-level requirements is a means of validating lower-level requirements. This process can be partially automated for all layers except the top, where stakeholders must be involved in writing or reviewing the initial requirements to ensure that the system will meet their needs. Automated validation can substantially reduce the risk of requirements errors leading to product defects or a system that fails to meet stakeholder needs. Automated validation makes it easier to catch such errors before detailed design begins, when the cost of fixing them is much lower.

On the verification side, formal modeling makes it feasible to generate test cases automatically, even for a system that does not yet exist (specification-based testing). There are two main advantages to automating the verification process: reliability and cost. Automated test case generation has the potential to be more reliable than manual techniques because it is more precise and can find faults that a human engineer would miss. It should also improve productivity by automating what is normally a labor-intensive process. Testing typically consumes more than 50% of software development costs in safety-critical systems, with the bulk of those costs in the construction and review of the test cases rather than their execution[3].

2 Conceptual Framework

2.1 Formal Methods

As mentioned above, the key to automated V&V is developing a formal model of specified system behavior. This means a model with precise and unambiguous semantics. Once a formal model has been developed, it can be used to check the consistency of one specification with another, and also as a basis to determine if an implementation ("application under test") conforms to the specification.

Finite state machines (FSMs) are a popular representation for formal modeling. The mathematical definition of FSMs satisfies the requirement for precise, unambiguous semantics. FSMs are not Turing-complete[4] and thus fall short of the power of general computation, but this can be an advantage as it makes their behavior easier to analyze. FSMs are well suited for describing reactive and embedded systems, which must continually monitor their environment and respond to any changes. Because they are represented graphically, FSMs allow specification of behavior through a graphical editor rather than traditional programming, which is often easier for the domain expert.

Taken individually, FSMs are inadequate to describe a complex system, because there is no mechanism for abstraction and the number of states and transitions quickly becomes unmanageable. However, this limitation has been addressed with extended finite state machines (ESFMs), which allow for communicating networks of concurrent processes each represented by an FSM. A further extension is Statecharts, which allows states to be partitioned into sub-states and FSMs to be grouped into a hierarchy. The Unified Modeling Language (UML) provides several types of diagrams for modeling behavior, but Statecharts are the only one with sufficient formality for automated V&V. (To make things more confusing, UML 2.0 no longer uses the term Statechart and goes back to "state machine"[5]).

2.2 Logic-Based V&V Approaches

This project began with a study of the Burton dissertation[1], which involved generating a Statechart model for a demonstration project in the safety-critical domain of aircraft engine control. This model was translated into the logic-based specification language Z. (To demonstrate the generality of the method, a second semi-formal method called PFS was also used as an input representation and translated into Z.) Automated model checking and theorem-proving techniques were then applied to check the requirements and generate the test cases. The test cases themselves were represented formally, which enabled the use of conventional constraint solving techniques (such as linear/integer programming) to generate test data.

Burton notes that formal methods based on logic-based specification languages such as Z and VDM have yet to gain much acceptatnce in industry. The main reason is a mismatch between the representation required by the formal methods (predicate calculus or some variant) and the representations used in industry for engineering the safety-critical systems that would benefit most from automated V&V. Logic-based formal languages remain difficult for most engineers to use, and have the drawback of being very sensitive to change (even a small change in the

underlying requirements may force a major rewrite of the formal specification). These problems make it very costly and often infeasible to write specifications for complex applications directly in a formal language like Z and then validate them against higher-level requirements. Burton states that "formal specifications can be extremely time consuming and costly to produce. Small changes in the requirements can result in much rework of the specification and accompanying analysis."[6] This was the motivation for beginning with a more "friendly" language for knowledge capture and developing a translator. However, the proof heuristics used for test case generation still had to be written in Z, so this architecture required expertise in both representations.

My original intention to follow this approach in my project had to be abandoned for lack of suitable tools. Research into Z translation turned up a few interesting academic projects such as VISTA[7] (VIsualization of STAtecharts, from the University of Texas), but little that was freely available for download. One of the best Web resources in this field is "Formal Methods Links", by Mark Utting[8]. After exploring what seemed like dozens of blind alleys, I did find a freeware tool called Nitpick (now Ladybug) that is based on a subset of Z[9] and got it running on my Macintosh at home, but its functionality was limited to model checking and did not include test case generation. Many of the links on the Z User's Group website[10] were broken and many others dated from the mid-90s, which seemed a bad sign.

One question about the Burton work that seems more important with hindsight is why it was necessary to translate the model into Z in the first place. Burton makes it clear why Z is unsuitable as a modeling language and then arrives at the solution of capturing a model from a graphical language like Statecharts and translating it into Z. But why not use the Statechart model directly, which has the advantage of already being supported by commercial tools? Burton also mentions SDL and UML as "other popular graphical notations that are also supported by commercial tools", but goes on to say:

> Statecharts go some way to satisfying Leveson's requirements on specification languages by hiding the formalism of the model but retaining a well defined set of semantics (although there are currently many different versions of the semantics in circulation). Statecharts provide a means of visualizing changes to the system state and can be a more intuitive way of specifying functional behaviour than model-based notations such as Z or VDM-SL. However, the languages and supporting tools are often poorly integrated into the verification and validation parts of the process. The semantics of the notations tends to be complex and are often not obvious from inspection of the diagrams alone. This complicates the task of providing automated V&V support for these notations.[11]

This seems a rather weak argument: even if some tools are "poorly integrated" into the V&V process, all of them don't have to be. Tools will evolve in the direction demanded by the users, so if users demand well integrated support for automated V&V, it will be provided. Logic-based languages clearly have many **potential** advantages for V&V. A predicate-calculus representation facilitates the use of theorem-proving techniques to prove properties about the specification and the application under test. The idea of applying these same techniques to the test cases

themselves for optimization is also powerful. However, the lack of commercial support for these languages (which have been around a long time) indicates that these advantages are not yet compelling enough to overcome the usability difficulties and persuade a significant number of users to switch. Because of the lack of standards and tools, each researcher in this area has to develop their own translator. Do we want to spend our time doing automated V&V, or writing translators? It may be an interesting area of research, but from an engineering perspective it definitely seems like a backwater.

I was more interested in finding a tool that was powerful enough to cover a range of automated V&V activities. Commercial tools tend to be of higher quality and better supported than the one-of-a-kind academic packages available on the Web. I was fortunate to find one that was available under an existing UMD academic license. Along the way, I found there were some interesting concepts behind the tools.

2.3 Standard Notations for Formal Modeling

Before plunging into modeling a system, it is helpful to have at least a rudimentary knowledge of three formal notations that have been developed to support model-based systems engineering: SDL, MSC, and TTCN. SDL is used to specify system architectures and state-machine models. MSC is used to represent requirements in the form of use cases and to trace execution. TTCN provides an abstract yet formal representation for test cases. All of these are nonproprietary, backed by standards, and can be represented in either a graphical form (for model capture and display) or a textual form (for storage and interchange with other tools). Taken together, they represent a powerful foundation for automated V&V.

2.3.1 Specification and Description Language (SDL)

SDL had its origins in the telecommunications industry. Mitschele-Thiel introduces SDL as follows:

> SDL is the main specification and description technique in the telecommunications area. SDL has been standardized by the International Telecommunications Union (ITU). In conjunction with tools, SDL is used by the majority of companies in the telecommunications industry, mainly to design communication protocols and distributed applications. In addition, it is employed during the standardization process of new protocol specifications by international standardization organizations such as ITU and ETSI. Besides its use in telecommunication, there is growing interest in using SDL for the design of real-time and safety-critical systems.[12]

SDL combines ESFMs with hierarchical data flow diagrams for structuring a system into a hierarchy and passing signals between different nodes in the hierarchy. The top layer of an SDL model is the system, which consists of one or more blocks. Blocks can contain either processes (modeled as FSMs) or other blocks (thus blocks may be nested as many times as necessary), but not a mixture of blocks and processes. Actual behavior is specified in the processes, which are defined as FSMs. Within a process, transitions are specified through additional symbols, not by

drawing arrows between states. A transition is defined by specifying the input to be read and the next state. As part of the transition, the process may also send an output to another process, set a timer, or assign a local variable. More detailed logic may be specified by calling a procedure.

Processes communicate by exchanging signals, which are propagated from one diagram to another through use of named input/output ports. Delays may be modeled on communication channels[13]; this could be used, for example, to simulate network performance or model light-time delay when communicating with a distant spacecraft. Processes may be dynamically created or destroyed. Processes are concurrent and asynchronous, but may be synchronized by exchanging signals. All signals and local variables must be explicitly declared in the highest-level agent (system, block or process) that uses them. Processes do not have access to the local data of other processes. Signals can activate transitions in other processes, generating other signals that propagate through the system. When there are no more transitions to execute in the model, time is allowed to advance. When a timer expires, it generates an event within the process that set it, which is treated like an internal signal. There are a lot more subtleties to the language, of course, but the above paragraph is enough to get started. The full ITU Z.100 specification[14] is freely available on the Web. While this is hardly a learning guide, it proved indispensable on several occasions when I needed to know some detail of the language.

The original Z.100 recommendation for SDL was in 1980[15], and the standard has been updated roughly every four years. A major extension occurred in 1992 (SDL-92) with support for object orientation through the definition of classes for agents (referred to as types) with single inheritance. The latest update, SDL-2000, extends the ESFM model to allow for composite states, making the SDL model equivalent to Statecharts.[16]

While Mitschele-Thiel dismisses the abstract structuring constructs of SDL (system and block diagrams) as merely a static structuring construct that "does not have a major influence on the implementation"[17], I see this as the most important aspect of SDL for large-scale systems engineering. Data flow diagrams are an indispensable technique for defining system architecture. In most SE environments these diagrams are widely used, but they are drawn with a general-purpose graphics tool (e.g. Visio or Powerpoint), with no notion of legal syntax or consistency checking, and even complex systems are often depicted with a flat diagram. UML provides some capabilities for modeling system architecture (component diagrams, deployment diagrams, and in UML 2, component structure diagrams), but there is no mechanism for formally connecting diagrams at different levels in a hierarchy. SDL makes it easy to draw hierarchical diagrams with formalized interfaces, so that each level can contain the requisite amount of detail without becoming overly cluttered. This model can be checked for consistency across diagrams (such as a signal at one level not connecting to a signal at the next higher level).

When a system is partitioned into blocks (or a block into lower-level blocks), a decision must be made on which lower-level entity handles each signal coming into the system from outside, and how the lower-level entities collaborate to support the interfaces at the higher level. This amounts to allocating interface requirements to subsystems, an important aspect of requirements engineering. In complex systems, many requirements problems arise not from misunderstanding of the requirement itself, but from confusion over which subsystems are responsible for handling the requirement and how they communicate (a problem familiar to anyone who has worked in a

large organization). Keeping the different levels of requirement consistent is a hard problem in practice, and formalizing the architecture permits the detection of inconsistencies that might otherwise go unnoticed until much later.

As its name implies, SDL is intended as both a specification language (through the system arcthitecture expressed as a hierarchy of system and block diagrams) **and** a design language (through the process behavior, which can be as detailed as needed). One problem with FSMs from an SE perspective is that the definition can be too detailed. Systems engineers need to enforce a clear boundary between their model of the system and the system itself. For systems containing hardware elements, this separation of perspective is enforced because the hardware elements must be simulated. For software-only systems, it is harder to know where to stop. Models **can** be used to generate an application, and tools like Rational Rose[18]are designed to do just that, but this is not an SE activity. Therefore, defining state-machine behavior in detail may be neither necessary nor desirable from an SE perspective. On the other hand, defining complex system architectures **is** an SE activity. SDL's abstraction features enable detailed exploration of the system architecture while the processes underlying that architecture are kept very simple, exposing just enough behavior to support the interfaces. This project was intended in part to illustrate how a complex system can be modeled with little or no knowledge of the detailed behavior of its components.

2.3.2 Message Sequence Charts (MSCs)

The MSC language , another ITU standard, is both a requirements language and a trace language. As a requirements language, MSCs formalize a use case by defining sequences of messages that are to be exchanged between objects and between objects and external actors. Each object or actor is drawn with a vertical bar, which is called its lifeline. Time is implicitly on the vertical axis while communication between processes is horizontal. This concept is familiar from UML sequence diagrams, but MSCs are a bit more formal.

MSCs come in two flavors, "system level" and "process level." A system-level MSC represents the system as a single column, showing only the messages exchanged between the system and its environment. (In both SDL and MSC, all external interfaces to an agent are shown as going to "the environment"; there is no way to make distinctions among external actors.) A process-level MSCs depicts the internal communications of the system, showing messages exchanged between individual processes and between processes and the environment.

As a trace language, an MSC records the dynamic behavior of an execution trace of the system, showing the actual messages exchanged. The use of the same notation for requirements and execution traces permits easy comparison for V&V purposes.

There is also something called a high-level message sequence chart (hMSC), which appears to be a way to structure and organize MSCs, but I had no time to investigate that in this project.

2.3.3 Test and Test Control Notation (TTCN)

TTCN is also referred to as Tree and Tabular Combined Notation. It is an ISO standard for the reprsentation of test cases as event trees. A TTCN test suite is "a collection of test cases together with all the declarations and components needed."[19]TTCN provides an abstract representation for a test suite, suitable for user inspection and editing. Later, the TTCN suite can be compiled into the target language to generate test driver software. TTCN is designed for black-box testing: the application under test can be perturbed and observed only at predefined interfaces, called points of control and observation (PCOs).

2.4 SDL Tools

A good tool will make just about any job easier, but in the case of model-based SE, the use of automated tools is not merely a convenience: it is the whole point of building the model. SDL tools can perform the following functions:

- Graphical editing and syntax checking
- Model checking (also referred to as static analysis; this checks consistency between different model elements)
- Animation of a specification via executable simulation
- State-space exploration (a more advanced form of model checking which searches the execution space of the system to look for paths that might lead to an error)
- Validation of an SDL architecture against an MSC use case
- Test case generation, guided either by manually driven simulation or by MSCs

Telelogic appears to be the market leader in this area through its Tau/SDL[20] product. According to Mitschele-Thiel, the best-known SDL tools as of 2001 were Tau/SDL and ObjectGEODE[21]. ObjectGEODE, originally developed by CS Verilog, has since been acquired by Telelogic. A quick Web survey did turn up some competitors, such as Solinet's SAFIRE-SDL product[22]and Cinderella[23]. A new tool called ezSDL[24]from United Computer Scientists, Ltd. has just been released.

All these appear to be high-end tools marketed to large organizations. Cinderella is the only one whose website gave pricing information, and it showed prices of $2500 for the SDL portion and another $1600 for the MSC tools (TTCN was not mentioned). I chose Tau/SDL for this project. Its advertising literature was impressive, and UMD happened to have a license for the product through the CS department. This license had been allowed to expire, but with the help of Prof. Victor Basili, it was renewed in time to use it on the project.

2.5 Proposed Methodology

The activity diagram in Figure 2-1 depicts the proposed process for automated V&V (taken in part from the Tau/SDL methodology guidelines).

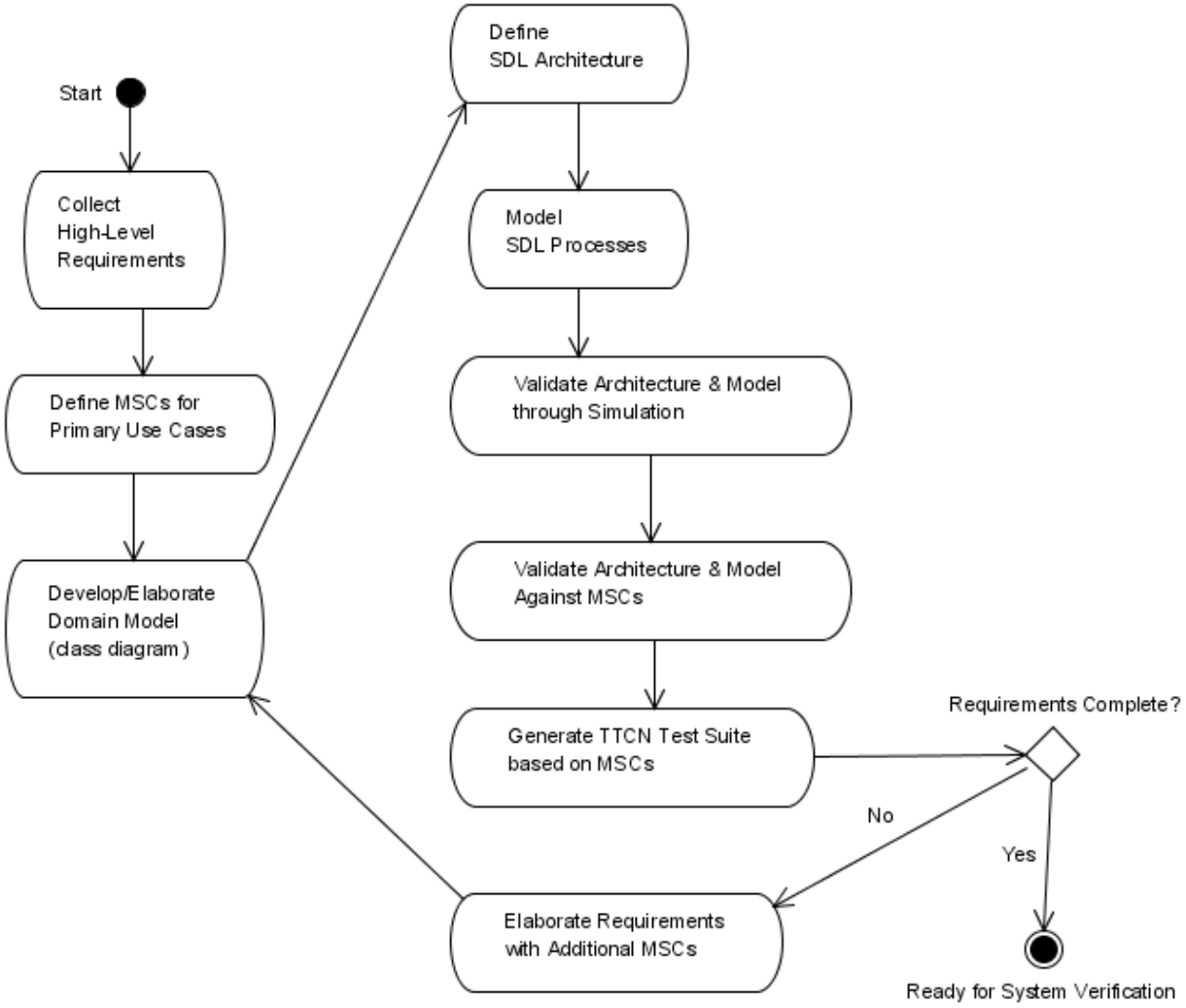

Figure 2-1: Automated V&V Methodology

The first three steps are familiar. First a set of high-level, informal text requirements is collected. Second, an initial set of use cases is defined. Here these are represented in the form of system-level MSCs. Third, a domain model (class diagram) for the key objects in the domain is developed. This class diagram is not used in SDL, which has relatively poor features for object-oriented data structures, but it is recommended anyway in order to understand the domain better in preparation for system architecture definition.

The next step is to define the architecture of the system as SDL system and block diagrams. The SDL model is normally developed top-down, although it is also possible to assemble it bottom-

up from reusable components. In this process it is assumed that no suitable components exist. After checking and reviewing the system architecture, the process-level behavior is defined. This opens the way to generating an executable simulation of the system.

Once the executable is generated, the system is simulated to validate the system architecture and process-level design. The simulation is run and inspected by hand, so validation is still a manual process at this point. The next step, however, is to validate that the system-level use cases (MSCs) developed at the beginning can be executed as expected using the SDL model. If this validation is successful, the process proceeds to verification, which is accomplished by using the MSCs to generate a TTCN test suite. This test suite can be run against the actual application once it is developed. In an incremental development model, the system is elaborated with additional MSCs and the process is repeated as often as necessary.

3 Project Formulation

3.1 Remote Astronomy Case Study

The case study chosen for this project was control of a remote astronomical observatory, a continuation of the project from ENSE 621. This application is modeled after a space observatory such as Hubble or James Webb, but because there is nothing in the model that requires the telescope to be in space, it was abstracted as a remote observing platform which could be anywhere. In this extremely simplified model, the observatory supports only a single instrument (a camera). The observer can turn the instrument on, turn it off, or use it to take an exposure with a specified duration and bandpass filter. Before an exposure is taken, the telescope must be slewed to the target and guide stars must be acquired to stabilize the pointing. During the exposure, photons arrive from the environment, are captured and focused by the telescope, and are stored in a detector within the camera. After the exposure is complete, the instrument is commanded to "read out" its data into an internal data buffer. Subsequent exposures can also add data to the buffer. When a series of exposures is complete, the observer commands the instrument to dump the data buffer to secondary storage (a data recorder). A subsequent command will download the data from the recorder.

While this is a greatly oversimplified description of how a real observatory is operated, it still gives rise to a complex architecture to model the observatory, providing an adequate basis for evaluating the suitability of SDL for this type of problem. The point of the project was to demonstrate a process for automated V&V, not to model the application in detail. I had originally planned to add some contingency scenarios to work in more reactive behavior. This extension would not have been difficult, but had to be dropped for lack of time.

3.2 Tau/SDL Configuration

The Tau/SDL software (version 4.6) was installed on a Windows platform at home. Tau/SDL requires installation of a third-party C compiler in order to produce executable applications. Borland and Visual C++ were recommended. I downloaded the recommended freeware Borland C++, installed it, and verified that it could be used to compile a trivial "hello, world" program.

Tau/SDL apparently offers no way for the user to configure the system with the C compiler; it just uses the operating system to find the compiler and then runs it. This worked the first time (I'm not sure what would happen if more than one compiler were installed). The only hitch is that the C compiler version available from Borland (version 5.5) was newer than the one listed as supported in the Telelogic documentation (version 5.02). It is possible this configuration change may have introduced incompatibilities, which could explain some of the strange behavior documented in section 4.6 below.

4 System Definition

4.1 System Context

The use-case diagram in Figure 4-1 defines the system context for the Observatory case study. There are three external actors, an Observer (who operates the Observatory and receives the data) and two from the sky, a Target and GuidingSource. In this simple model, there are three use cases: TurnOnInstrument, TurnOffInstrument, and Observe. The Observer participates in all three. The Target and GuidingSource participate only in the Observe use case.

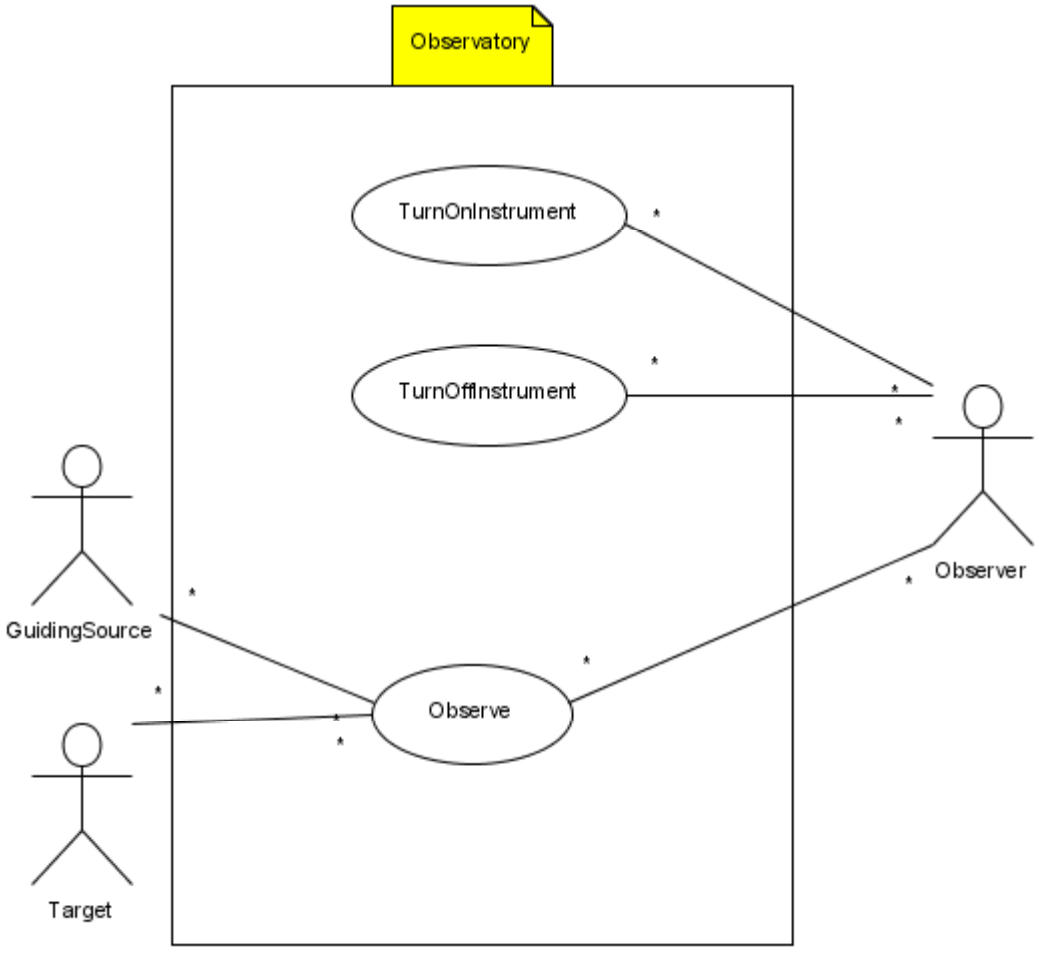

Figure 4-1: Observatory System Context

4.2 Use Case Definitions

The use cases in this project were represented as system-level MSCs for future use in validating the SDL model. Figure 4-2 and Figure 4-3 show the externally visible behavior for TurnOnInstrument and TurnOffInstrument, which is so simple as to be trivial. The Observer (acting through the Environment) sends an InstrumentOn command to the Observatory, which

responds after some time with an InstrumentReady signal. TurnOffInstrument is similar: the environment sends an InstrumentOff command, and the Observatory responds with confirmation via InstrumentPowerOff.

Note that the InstrumentOff use case includes the InstrumentOn use case as a preamble. Since these use cases are intended to be used formally in validating the model, we have to be careful about their definition. When the model is started, the instrument state is initialized to PoweredOff, but in order to test the InstrumentOff use case, it should be in the Ready state. I wasn't sure how to define preconditions for an MSC or refer to one MSC in another (perhaps this is what the hMSC construct is for), so to avoid any ambiguity, I simply duplicated the TurnOnInstrument behavior as the first step in both TurnOffInstrument and Observe. For a more complex set of use cases, it would be necessary to find a more modular approach.

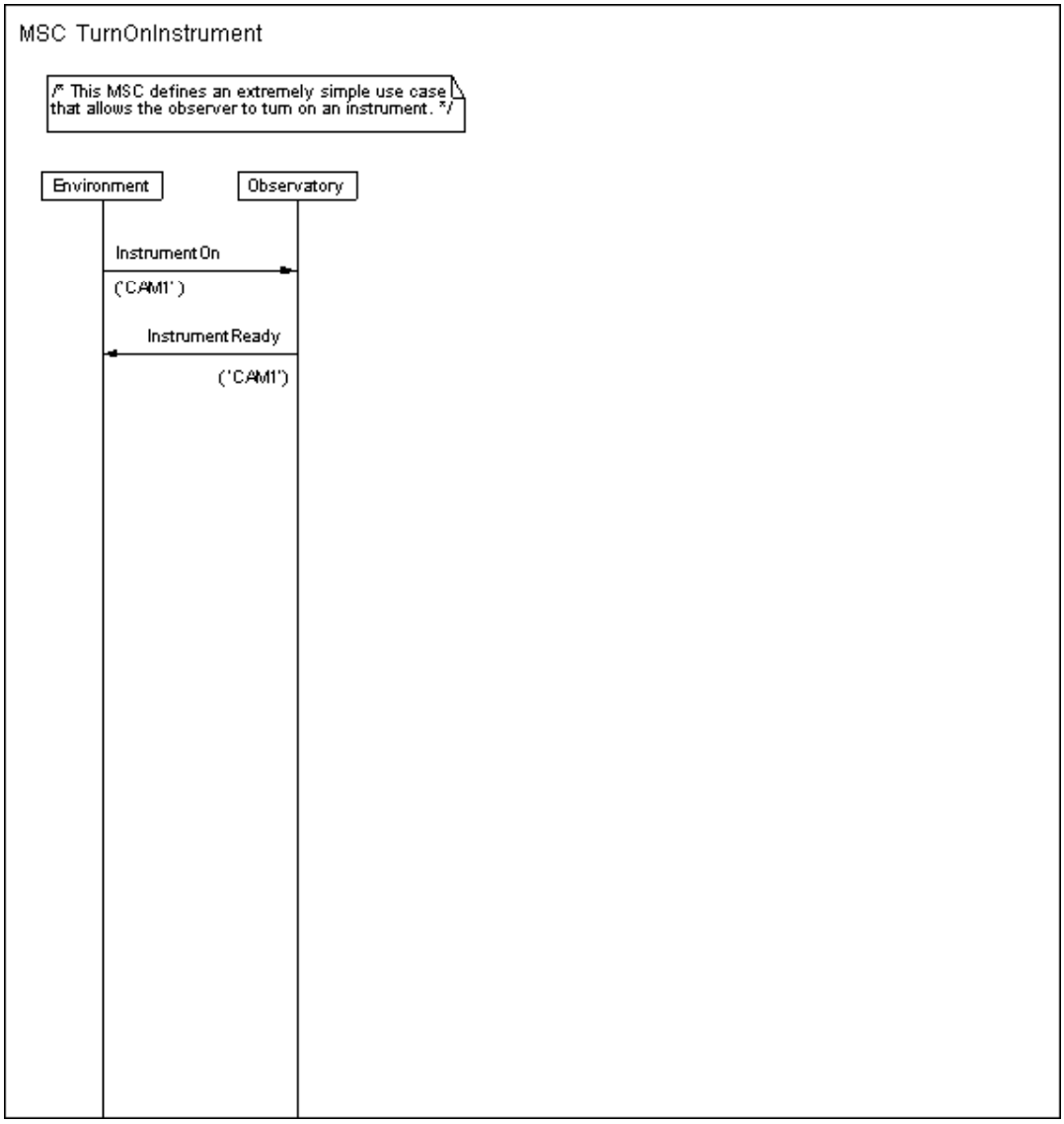

Figure 4-2: TurnOnInstrument

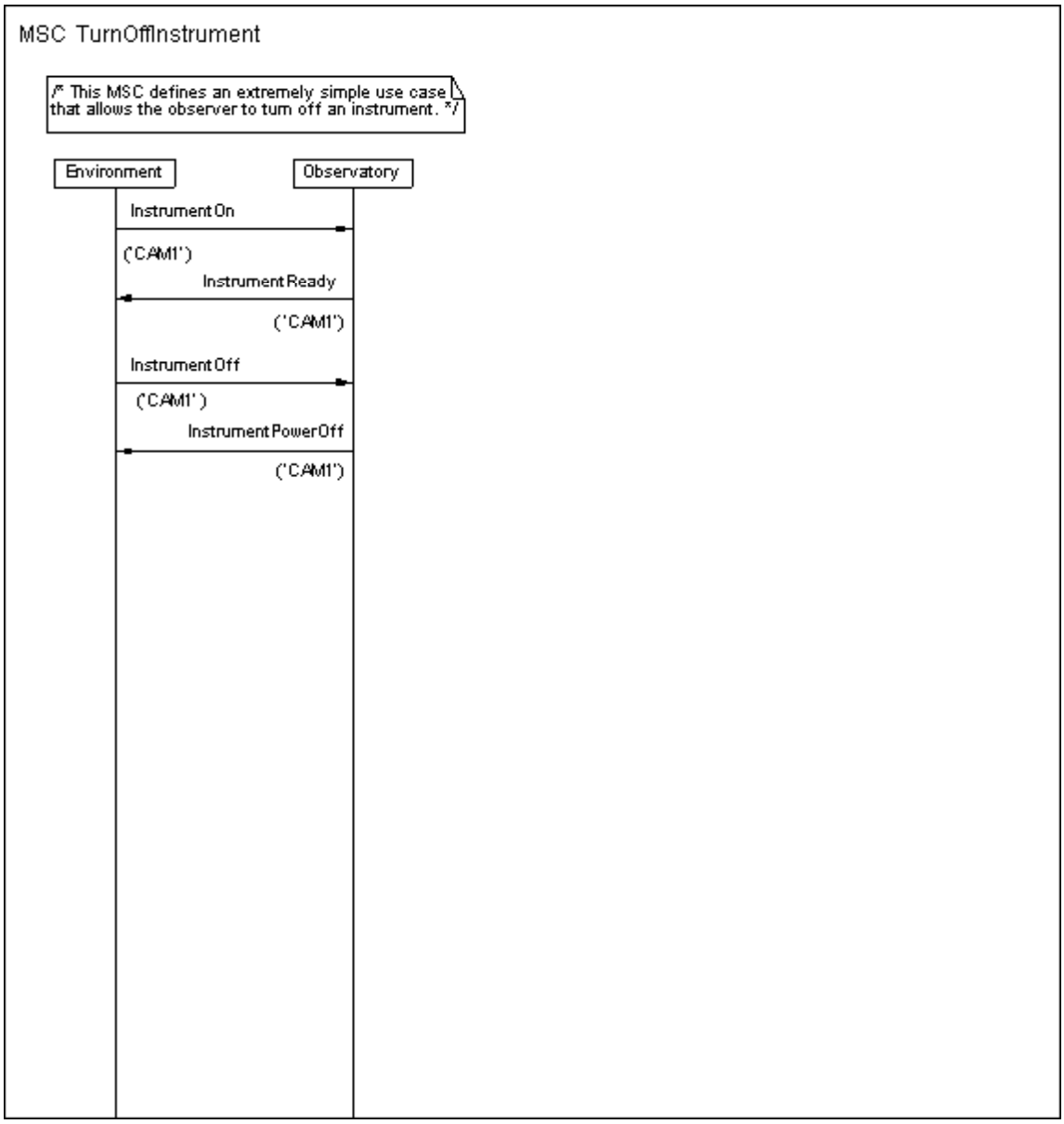

Figure 4-3: TurnOffInstrument

Figure 4-4 shows the Observe use case, a specific test scenario that is used as a basis for the automated V&V work in this project. After turning the instrument on, the Observer sends an Expose command. This command specifies the instrument to be used (CAM1), the celestial coordinates (right ascension and declination) of the target to be observed, the celestial coordinates of the guiding source, the exposure duration (5 time units), and the filter name (VIS_FILTER). Note that the exposure does not begin immediately; the system must wait for a GSLock signal from the environment, confirming that the telescope pointing has been stabilized enough to provide meaningful data. (The planned contingency scenarios involved failure to establish lock or loss of lock after it had been achieved.)

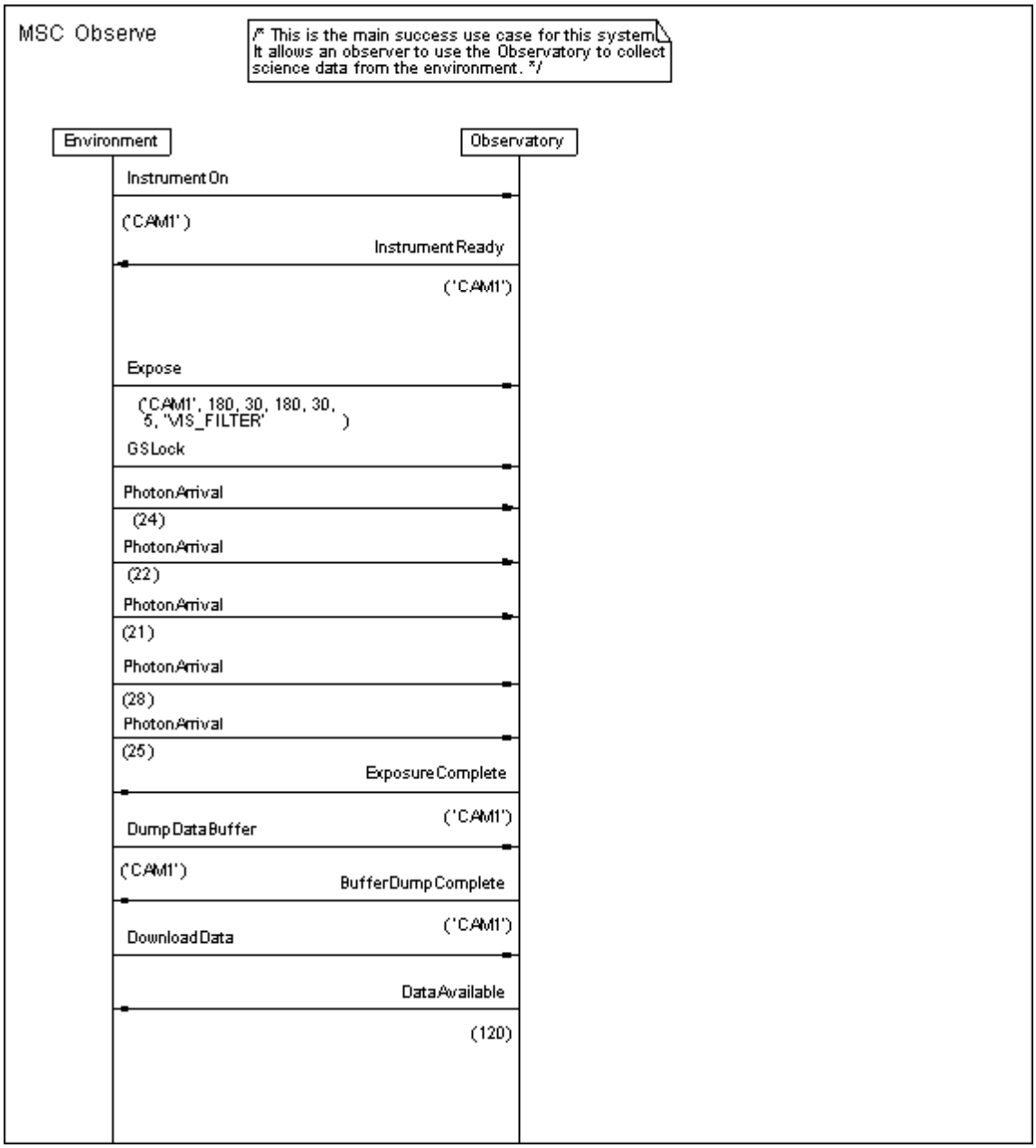

Figure 4-4: Observe

After sending the GSLock signal, the exposure begins. PhotonArrival signals begin arriving from the target (also part of the environment). This model assumes that photons arrive in packets, with a specified number of counts. While this is not physically realistic, it is convenient for testing: a large number of counts can be simulated with a small number of signals. In this scenario, five distinct PhotonArrival signals with slightly different count values are sent during the exposure.

When the specified exposure time has elapsed, the Observatory sends an ExposureComplete signal to the Observer. Additional exposures could then be taken, but in this scenario, the Observer immediately sends a DumpDataBuffer command to get the data off the instrument and

onto the recorder. When this has been accomplished, the Observatory responds with a BufferDumpComplete signal. The Observer then sends a DownloadData command and the Observatory responds with a DataAvailable signal. In this simplified model, the only “data” that is returned is a count of photons received during the set of exposures that produced the data on the recorder. This is sufficient to test that there is a meaningful relationship between input and output. For this use case to be verified, the model must be capable of processing five independent PhotonArrival signals, adding them, and returning the sum when the DownloadData command is received.

4.3 Domain Model

Figure 4-5 gives the class diagram for the Observatory system. This information is not used in the model itself, but following the proposed methodology (section 2.5), it is performed as an exercise in order to improve understanding of the domain prior to developing the model. The model shows the Observatory as an aggregation of three subsystems, a Telescope, InstrumentModule, and SupportModule. (In the relatively crude SDL object model used to develop this class diagram, there is no symbol for generalization and no distinction between aggregation and composition.) The SupportModule contains the DataRecorder. The Observatory receives data from an Astronomical_Object, which may be either a Target or a Guiding_Source.

The InstrumentModule contains a Guider and a ScienceInstrument, which may be either a Camera or a Spectrograph. A ScienceInstrument contains an InstElectronics subsystem, a FilterSubsystem, a Detector, and a DataBuffer. The FilterSubsystem contains a FilterWheel, which consists of one or more Filters. If the ScienceInstrument is a Camera, it additionally contains a Shutter to block external light to permit certain types of calibrations.

This model serves as a basis for the system architecture developed in section 4.4, but the entities defiend here do not match the SDL agents exactly. That is because the architecture definition inevitably led to a more precise understanding of the system and resulted in the definition of additional entities that may be logical (representing software) ather than physical. The class diagram is intended to be more high-level and conceptual, and was not updated to reflect the particular architecture defined for this system.

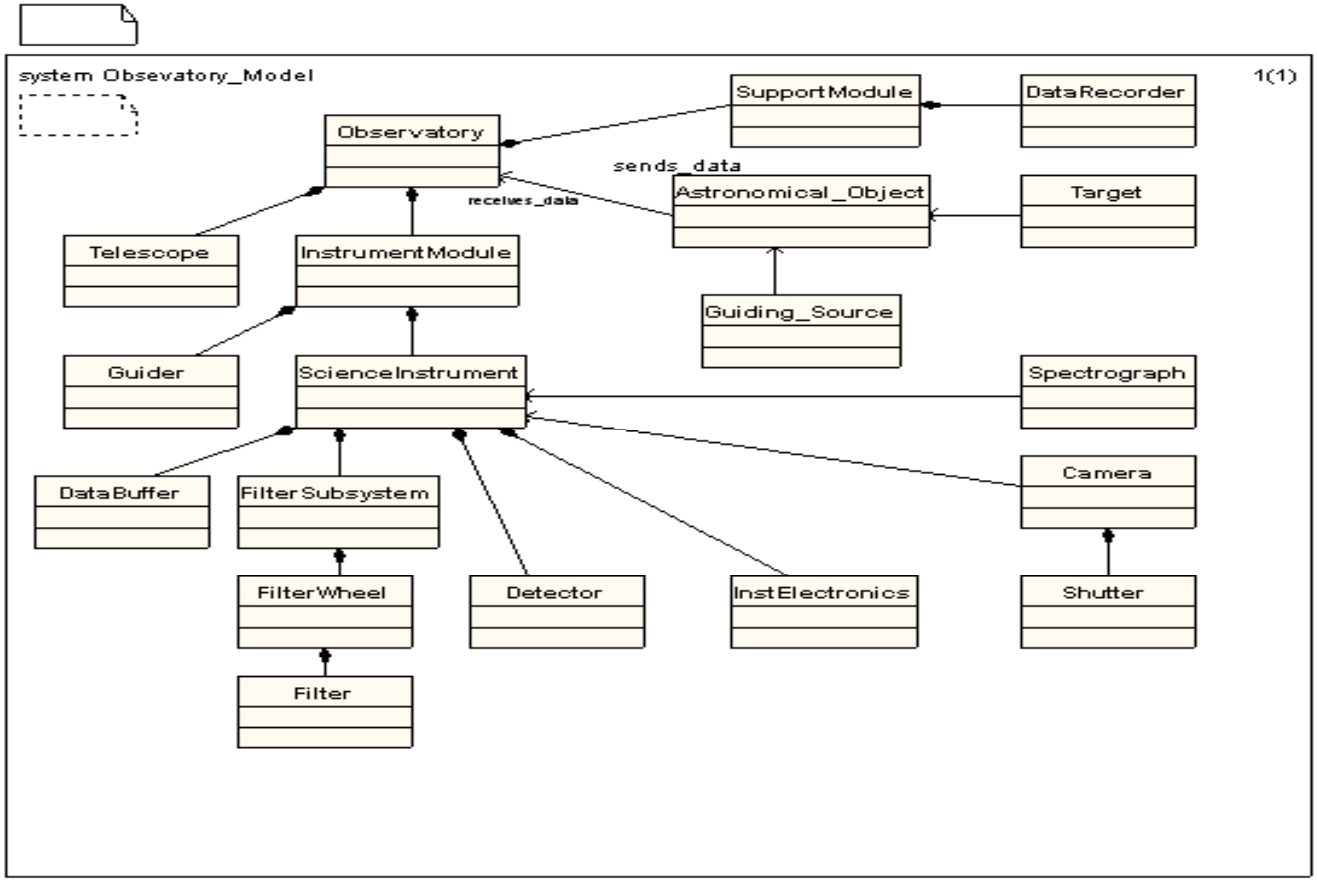

Figure 4-5: Observatory Class Diagram

4.4 SDL Architecture

The architecture of the Observatory system is defined in seven SDL agents: one system diagram and six block diagrams. As many as three levels of abstraction were needed before the process level was reached.

Figure 4-6 shows the top-level system diagram. All the responsibilities of the system, in terms of supporting the external interfaces from the use cases, must be defined at this level. The use cases defined five commands from the Observer: InstrumentOn, InstrumentOff, Expose, DumpDataBuffer, and DownloadData. The signals returned to the Observer are InstrumentReady, InstrumentPowerOff, ExposureComplete, BufferDumpComplete, and DataAvailable. Further, there are two physical signals from the environment that must be processed: GSLock and PhotonArrival. These represent interface requirements that the system must support. Each "channel" to the environment is given a global name so it is visible to agents at lower levels; this serves to identify the external interfaces on those diagrams. By convention, separate channels are defined for input and output.

Note that the above interface requirements must not only be defined, but also allocated to subsystems (blocks). The Observatory comprises three subsystems: InstrumentModule, SupportModule, and TelescopeBlock. InstrumentModule handles all the commands from the observer that go to an instrument. SupportModule, which will contain the data recorder, handles the DownloadData command. The PhotonArrival physical signal goes to the TelescopeBlock. The GSLock physical signal goes to the InstrumentModule, which will contain the guiding sensor.

Further, internal interfaces may be introduced to define signals that are not visible to the environment but are needed to communicate among subsystems. In this architecture there are two internal interfaces. The interface between InstrumentModule and SupportModule allows the InstrumentData to be transferred when the data buffer is dumped to the recorder. The interface between InstrumentModule and TelescopeBlock allows the InstrumentModule, which receives the Expose command, to tell the TelescopeBlock to slew to the target. The TelescopeBlock sends a SlewComplete signal when the slew is finished. The TelescopeBlock also passes the PhotonArrival physical signal on to the InstrumentModule for processing.

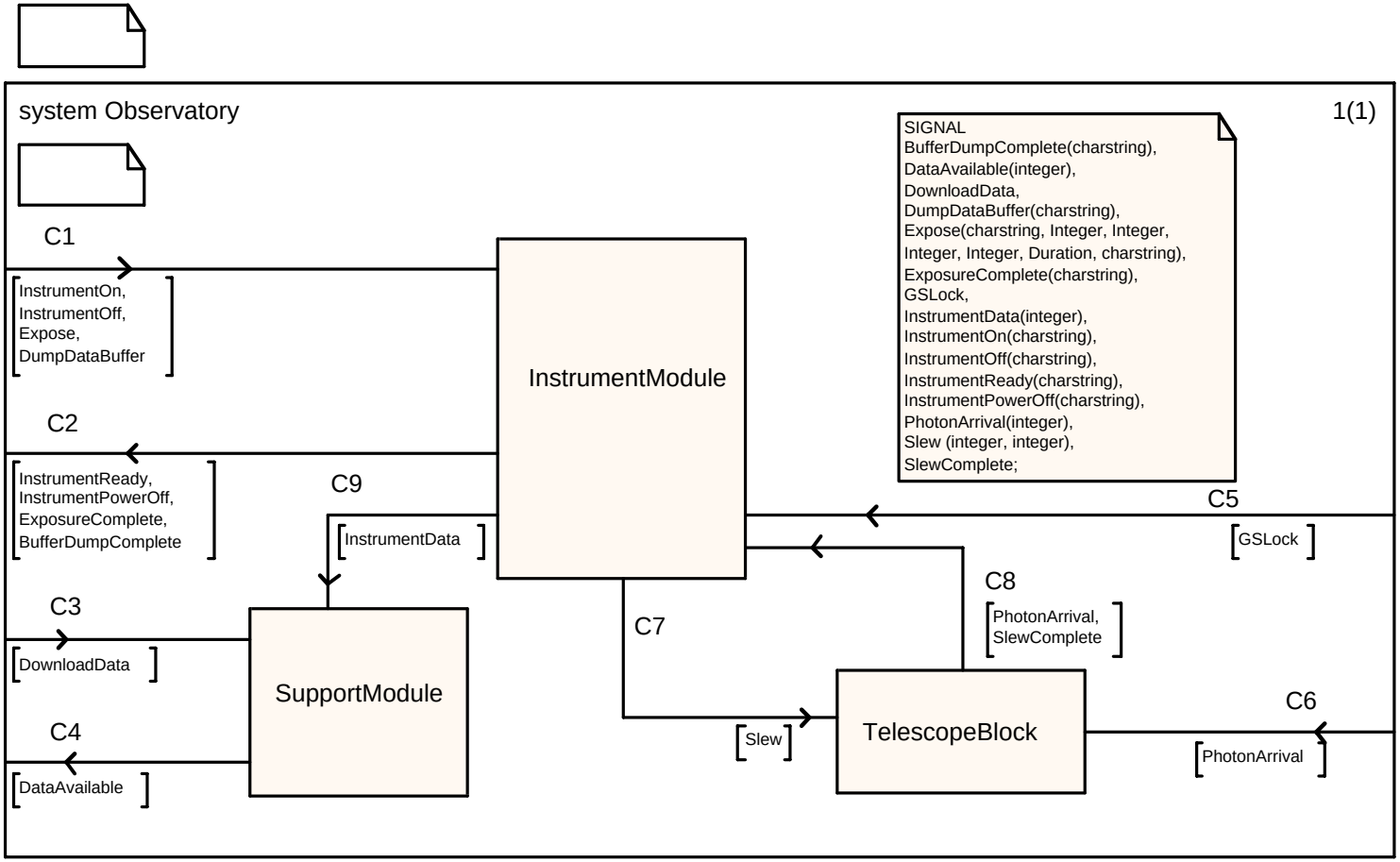
system Observatory
1(1)
SIGNAL
BufferDumpComplete(charstring),
DataAvailable(integer),
DownloadData,
DumpDataBuffer(charstring),
Expose(charstring, Integer, Integer,
Integer, Integer, Duration, charstring),
ExposureComplete(charstring),
GSLock,
InstrumentData(integer),
InstrumentOn(charstring),
InstrumentOff(charstring),
InstrumentReady(charstring),
InstrumentPowerOff(charstring),
PhotonArrival(integer),
Slew (integer, integer),
SlewComplete;
C1
InstrumentOn,
InstrumentOff,
Expose,
DumpDataBuffer
InstrumentModule
C2
InstrumentReady,
InstrumentPowerOff,
ExposureComplete,
BufferDumpComplete
C9
InstrumentData
C5
GSLock
C8
PhotonArrival,
SlewComplete
C3
DownloadData
SupportModule
C7
C6
Slew
TelescopeBlock
PhotonArrival
C4
DataAvailable

Figure 4-6: Observatory (Level 1)

Figure 4-7 shows the diagram for a much simpler agent, SupportModule. This one has only one process, DataRecorder, which handles all the interfaces. The channel names (C3, C9, and C4) match the interface responsibilities from the Observatory diagram that were allocated to SupportModule: DownloadData, InstrumentData, and DataAvailable. Additional signal routes, SM1-SM3, are defined to pass these signals into the DataRecorder process.

The DataRecorder **process** (the large hexagon in the figure) is defined as an instance of the DataRecorder **process type**: a class that encapsulates behavior (shown as a smaller hexagon with double lines). All the processes in this model but one are defined as process types to illustrate this object-oriented capability. Process types can be reused in any application that supports the same interfaces. The declaration (1,1) states that the block is initialized with one instance of this process and that the maximum number of instances is also one. The SDL capability for dynamic creation of process instances was not used in this project, since most processes correspond to physical components that always exist.

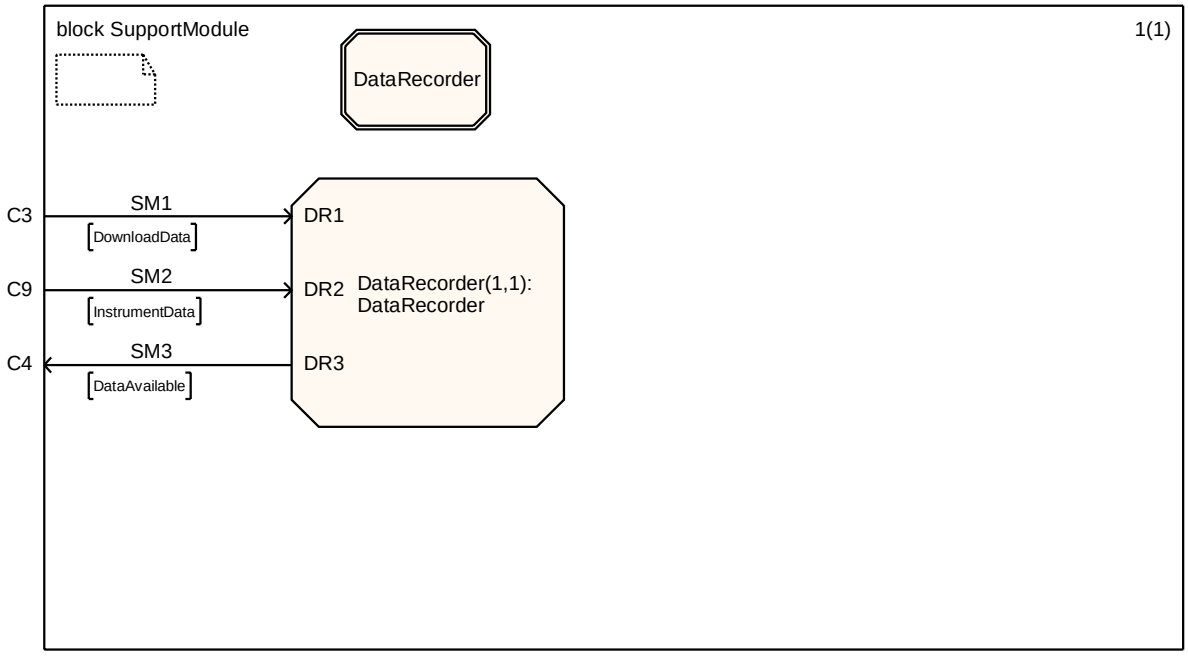

Figure 4-7: SupportModule (Level 2)

Figure 4-8 shows the TelescopeBlock, which is a bit more interesting. This block comprises two processes, OpticalAssembly and AttitudeControlSystem. Recall from the Observatory diagram that TelescopeBlock had two input channels, C6 and C7, and one output channel, C8. The OpticalAssembly block, which contains the physical telescope, simply passes the PhotonArrival signal through to its environment (note that the block has no visibility to anything that happens outside its scope). This is the behavior required from the Observatory diagram, which showed PhotonArrival as coming from the environment and being passed through from TelescopeBlock to InstrumentModule. However, since the component is still being modeled by an active process, the InstrumentModule must define the behavior to process the input and generate the output. If this does not happen, the signal will be blocked.

The AttitudeControlSystem block is equally simple: it handles the Slew command and sends a SlewComplete signal to its environment when done. Both blocks are defined as process types, as before.

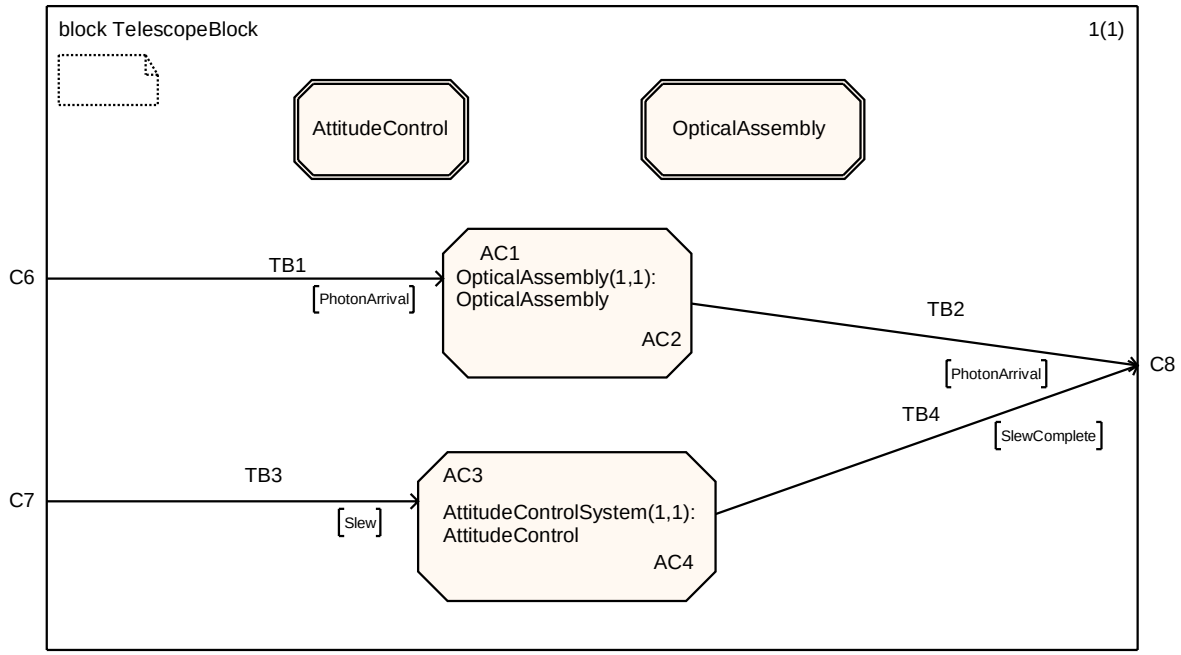

Figure 4-8: TelescopeBlock (Level 2)

Figure 4-9 shows the InstrumentModule block, which contains most of the complexity in this model. In order to manage this complexity, three additional blocks are defined. The InstManagerBlock manages all the interfaces with the observer. The guiding function is assigned to the GuiderBlock, while the single instrument in our model (CAM1) handles all instrument commanding and data processing.

This block, unlike the previous two, has internal interfaces between subsystems, and so additional signals are declared. The InstManagerBlock sends a DropLock command to the

GuiderBlock at the start of an exposure, in preparation for a slew to a new target. After a slew is complete, it sends a LocateGS command to the GuiderBlock, which is answered by a GSLocated signal. If the guide star is successfully located, the InstManagerBlock can send an AcquireGS command to the GuiderBlock. The GuiderBlock must wait for a GSLock signal from its enviroment before sending the LockEstablished signal to the InstManagerBlock.

The InstManagerBlock passes the four instrument commands from the environment (InstrumentOn, InstrumentOff, Expose, and DumpDataBuffer) on to CAM1 and relays the result signals back to the environment. The Expose Command is changed to DoExposure at this point to simplify the interface. In addition, the physical PhotonArrival signal is passed into the instrument.

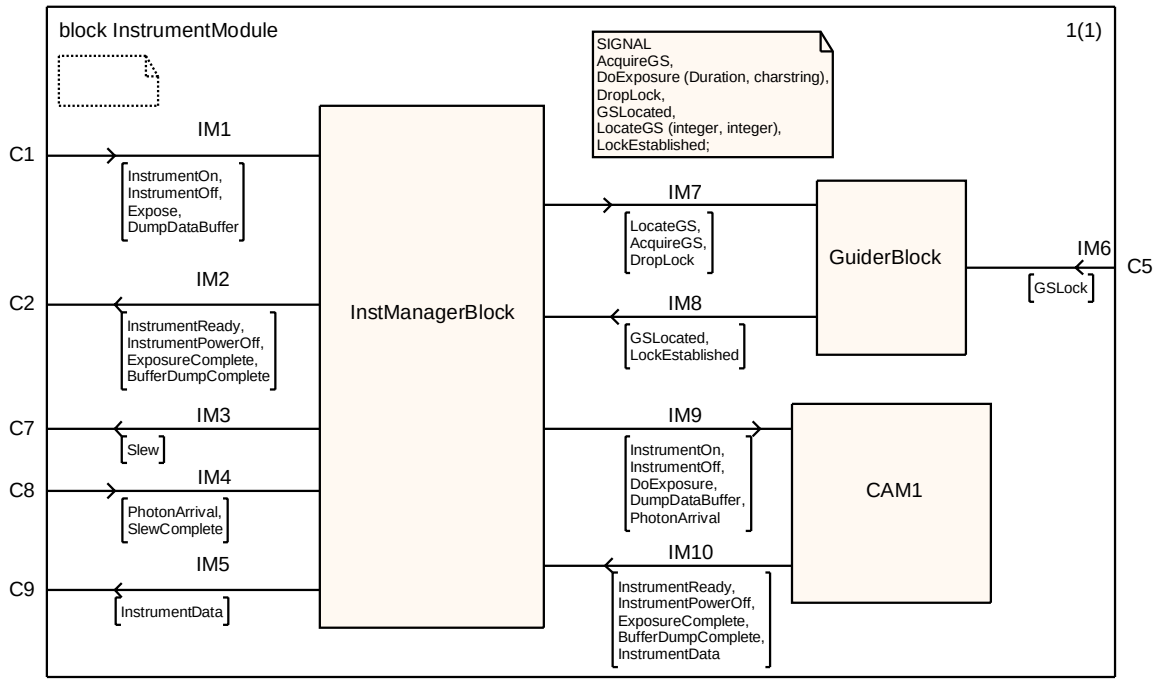

Figure 4-9: InstrumentModule (Level 2)

Figure 4-10 shows the InstrumentManager block, which consists of a single process called InstrumentManager. This process receives all the inputs from the environment and generates all the outputs. The same goes for GuiderBlock (Figure 4-11). Blocks like these, containing a single process, would be unnecessary if not for the SDL restriction that blocks and processes may not be mixed in the same diagram.

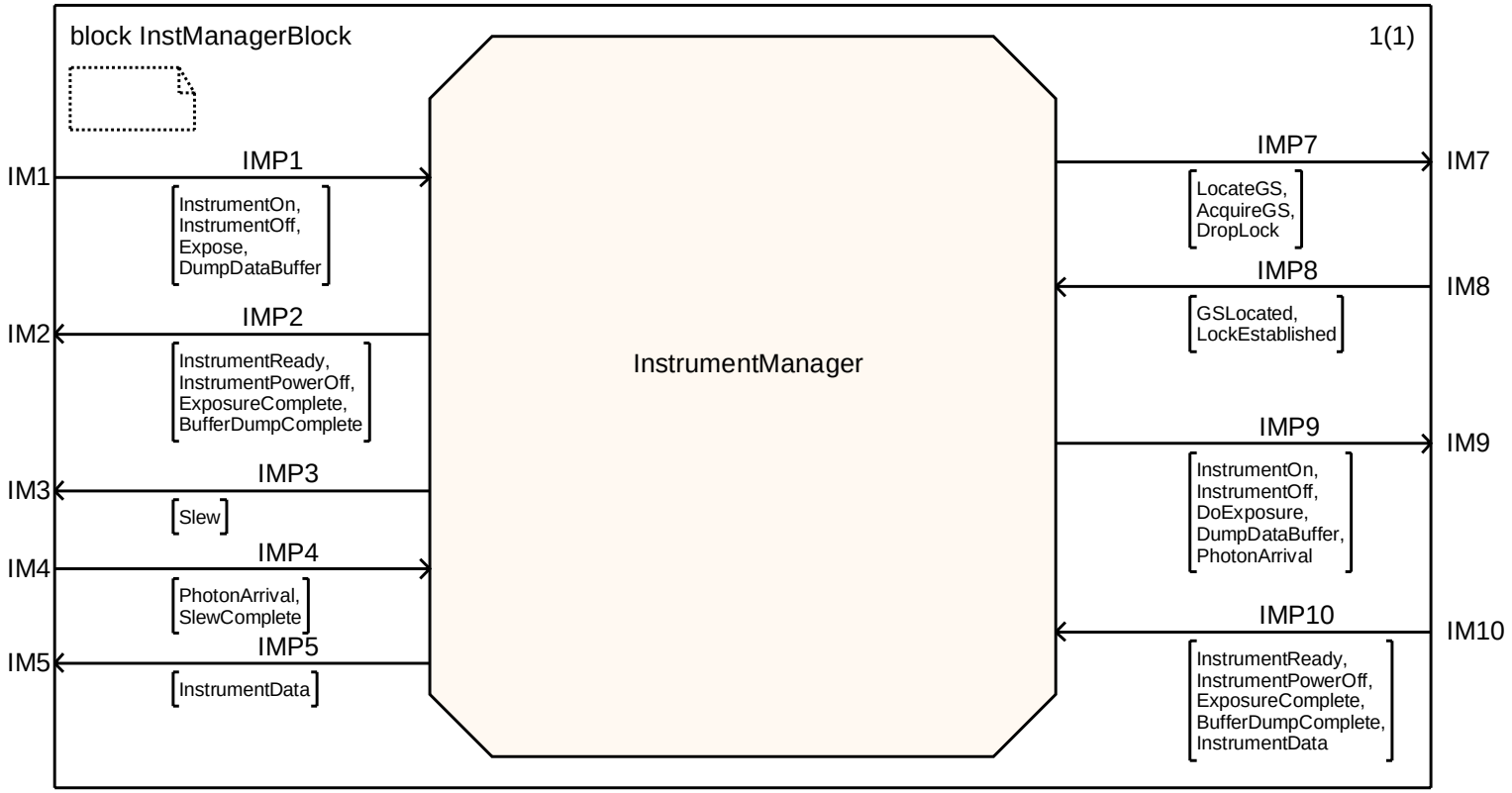
block InstManagerBlock
1(1)
IM1
IMP1
InstrumentOn,
InstrumentOff,
Expose,
DumpDataBuffer
IM2
IMP2
InstrumentReady,
InstrumentPowerOff,
ExposureComplete,
BufferDumpComplete
IM3
IMP3
Slew
IM4
IMP4
PhotonArrival,
SlewComplete
IM5
IMP5
InstrumentData
InstrumentManager
IMP7
LocateGS,
AcquireGS,
DropLock
IM7
IMP8
GSLocated,
LockEstablished
IM8
IMP9
InstrumentOn,
InstrumentOff,
DoExposure,
DumpDataBuffer,
PhotonArrival
IM9
IMP10
InstrumentReady,
InstrumentPowerOff,
ExposureComplete,
BufferDumpComplete,
InstrumentData
IM10

Figure 4-10: InstrumentManager (Level 3)

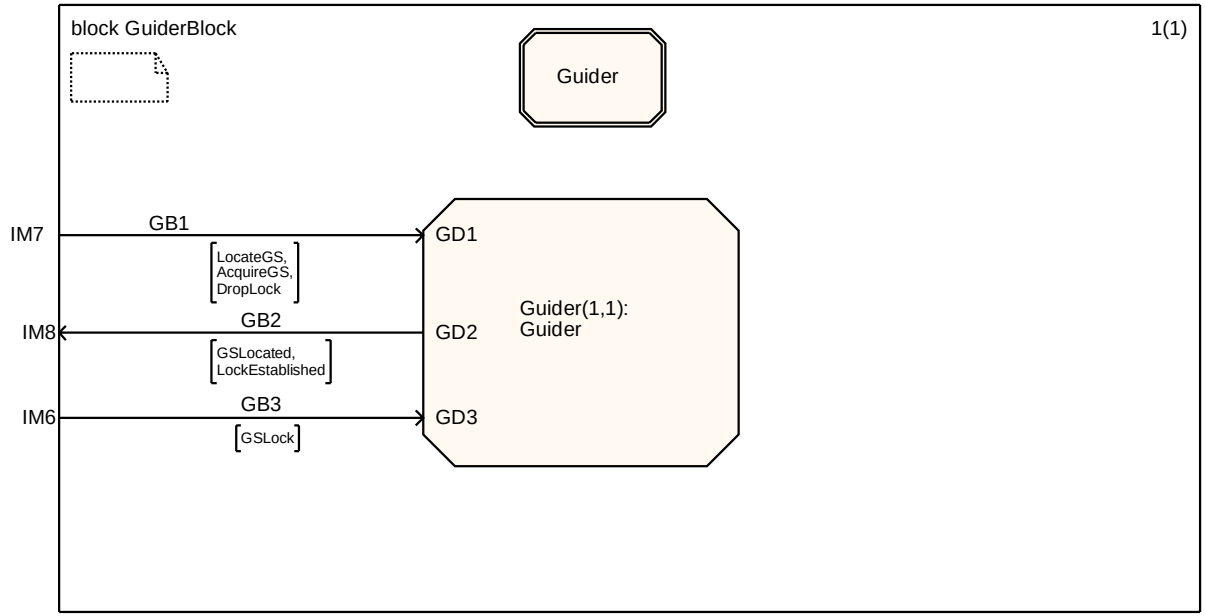

Figure 4-11: GuiderBlock (Level 3)

Figure 4-12 shows the CAM1 block, which manages the instrument behavior. This block has six processes. Five of these represent physical components: InstElectronics, Shutter, FilterAssembly, Detector, and DataBuffer. The large process in the center, CameraManager, is a logical entity that manages the instrument commanding and coordinates the other processes. All these processes are defined as instances of process types, which is the reason for the large number of internal labels (representing interfaces from the process type) that clutter the diagram. The CameraManager is defined as an instance of a special process type called HomingCameraManager that inherits from CameraManager; this will be explained in section 4.5.

The CAM1 block is responsible for four commands from its environment: InstrumentOn, InstrumentOff, DoExposure, and DumpDataBuffer. These all go first to CameraManager for coordination. However, the physical signal, PhotonArrival, is routed directly to the Shutter and then passed through to the FilterAssembly and Detector.

CAM1 is also responsible for five outputs to the environment: InstrumentReady, InstrumentPowerOff, ExopsureComplete, BufferDumpComplete, and InstrumentData. The first four are logical signals and are handled by CameraManager. InstrumentData, however, represents a physical data flow and is generated by DataBuffer when the buffer is dumped.

The internal interfaces between CameraManager and the other processes are all rather simple. InstElectronics handles the InstrumentOn command and generates an InstrumentReady signal when done. The InstrumentOff command is also sent to InstElectronics, but in that case the CameraManager does not wait for a response. The Shutter accepts OpenShutter and CloseShutter commands and responds with ShutterOpen and ShutterClosed. The FilterAssembly accepts PositionFilter and HomeFilter commands and responds with FilterInPlace. The Detector accepts an Integrate command and sends an ExpComplete signal when done. The Detector also accepts a ReadOut command, sends InstrumentData to the DataBuffer, and sends an ROComplete signal when done. Finally, the DataBuffer accepts a DumpDataBuffer command, sends InstrumentData to the environment, and sends a BufferDumpComplete signal when done.

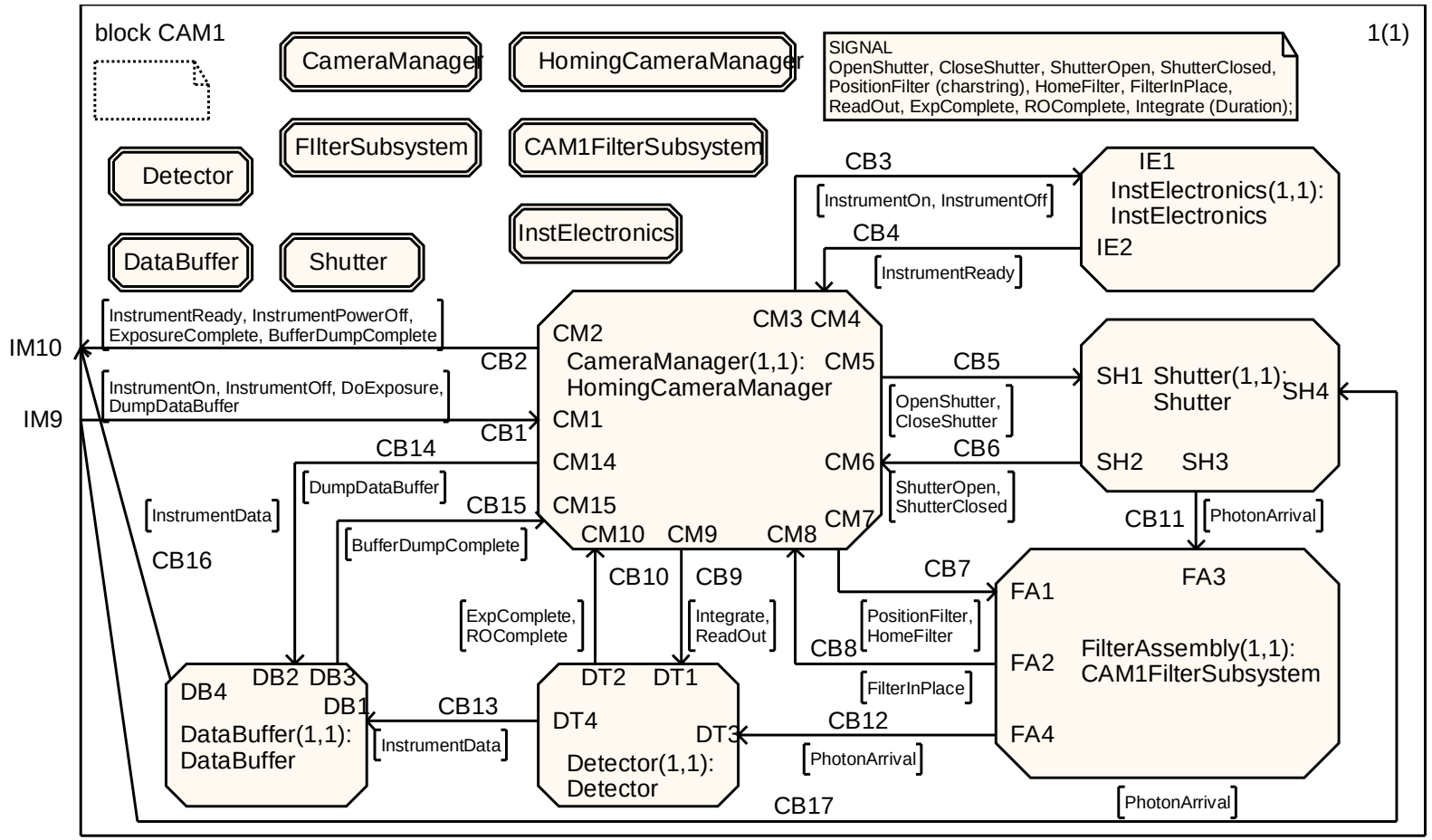
block CAM1
1(1)
CameraManager
HomingCameraManager
SIGNAL
OpenShutter, CloseShutter, ShutterOpen, ShutterClosed,
PositionFilter (charstring), HomeFilter, FilterInPlace,
ReadOut, ExpComplete, ROComplete, Integrate (Duration);
FIlterSubsystem
CAM1FilterSubsystem
Detector
InstElectronics
DataBuffer
Shutter
CB3
InstrumentOn, InstrumentOff
IE1
InstElectronics(1,1):
InstElectronics
IE2
CB4
InstrumentReady
IM10
InstrumentReady, InstrumentPowerOff,
ExposureComplete, BufferDumpComplete
CM2
CM3 CM4
CameraManager(1,1):
HomingCameraManager
CM5
CB2
CB5
SH1 Shutter(1,1):
Shutter
SH4
IM9
InstrumentOn, InstrumentOff, DoExposure,
DumpDataBuffer
CB1
CM1
OpenShutter,
CloseShutter
CB14
CM14
CM6
CB6
SH2 SH3
DumpDataBuffer
ShutterOpen,
ShutterClosed
InstrumentData
CB15
CM15
CM7
CB11
PhotonArrival
CM10 CM9 CM8
BufferDumpComplete
CB16
CB7
FA1
FA3
CB10
CB9
ExpComplete,
ROComplete
Integrate,
ReadOut
PositionFilter,
HomeFilter
CB8
FA2
FilterAssembly(1,1):
CAM1FilterSubsystem
DB4
DB2 DB3
DB1
CB13
DT2 DT1
FilterInPlace
DT4
DT3
CB12
FA4
DataBuffer(1,1):
DataBuffer
InstrumentData
Detector(1,1):
Detector
PhotonArrival
CB17
PhotonArrival

Figure 4-12: CAM1 (Level 3)

4.5 SDL Process-Level Design

The above architecture for the Observaatory defines 11 processes (actually 13, since two are inherited). This section defines the state-machine logic for these processes,

Figure 4-13 shows the DataRecorder process. (My apologies for the appearance of the ovals on these figures; they were exported to Word nice and sharp, but something seems to have been lost in transfer across the network.) The start symbol and all input symbols are labeled **VIRTUAL** to make the behavior overridable. The process uses a local variable to keep track of the amount of data on the recorder. The initial state is Empty. When an InstrumentData signal arrives, the process records the amount of data received and transitions to Readable. If additional data arrive in that state, the amount of data is incremented.

When a DownloadData command arrives in the Readable state (the double lines indicate that this is a priority input), the process outputs a DataAvailable signal with the total data on the recorder. It then resets the amount of data to zero and transitions back to Empty.

Note that not every input is accepted in every state. If a DownloadData command arrives when the state is Empty, no behavior is defined. In that case, the default SDL action is to have the process "consume" the signal and ignore it. The signal does not remain in the input queue waiting for the process to reach a state where it can be handled.

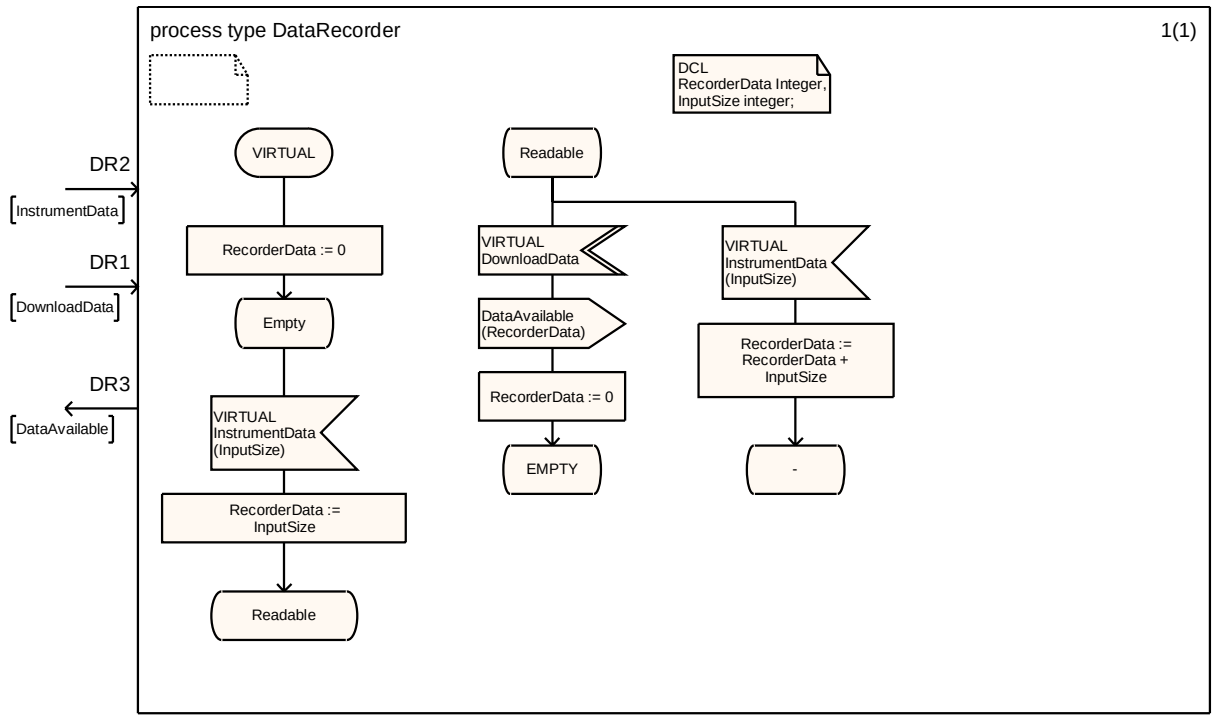

Figure 4-13: DataRecorder

Figure 4-14 shows the AttitudeControl process, which is responsible for the Slew command. The process keeps track of the current telescope pointing. The initial state is Stationary. When a Slew command is received, the process saves the destination pointing and enters the Slewing state. A timer is set to reflect the fact that telescope motions take time (in this model, 10 time units). When the timer expires, the process sets the current pointing to the destination pointing, outputs a SlewComplete signal, and transitions to Stationary.

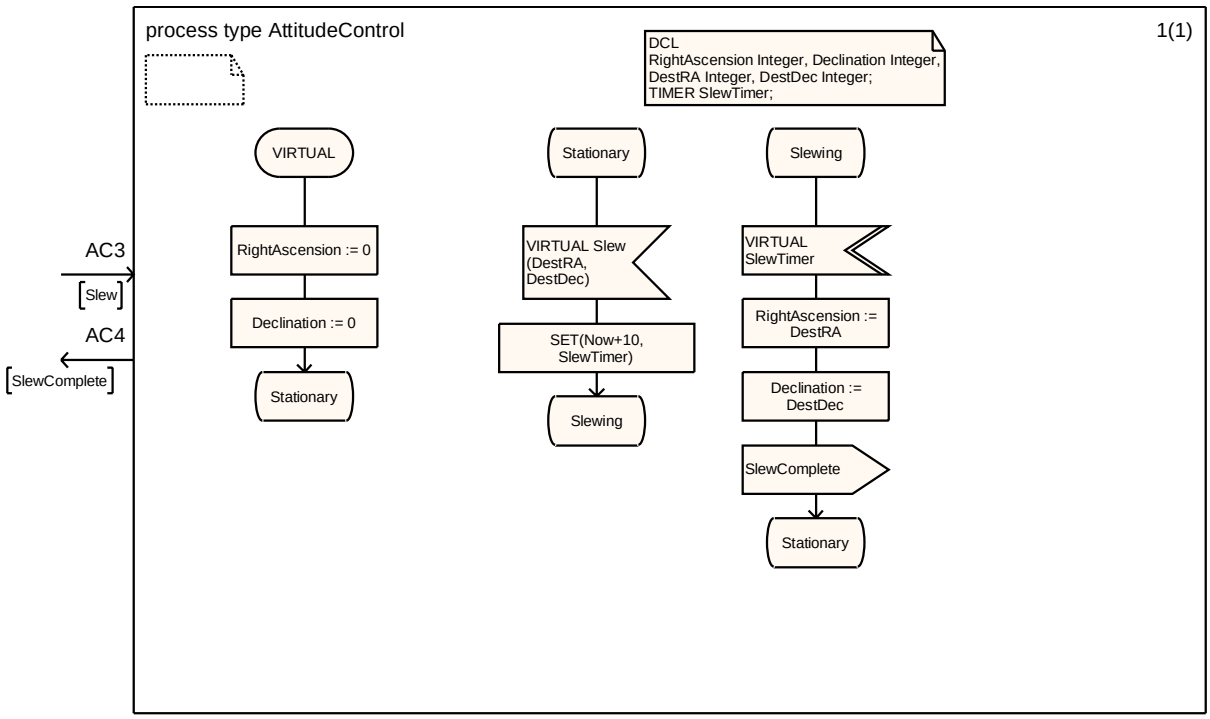

Figure 4-14: AttitudeControl

Figure 4-15 shows the OpticalAssembly process. Since this is a passive component, the flow is trivial. The process is always in the same state (called Focus), and all it does is accept the PhotonArrival signal and pass it back to the environment on a different channel.

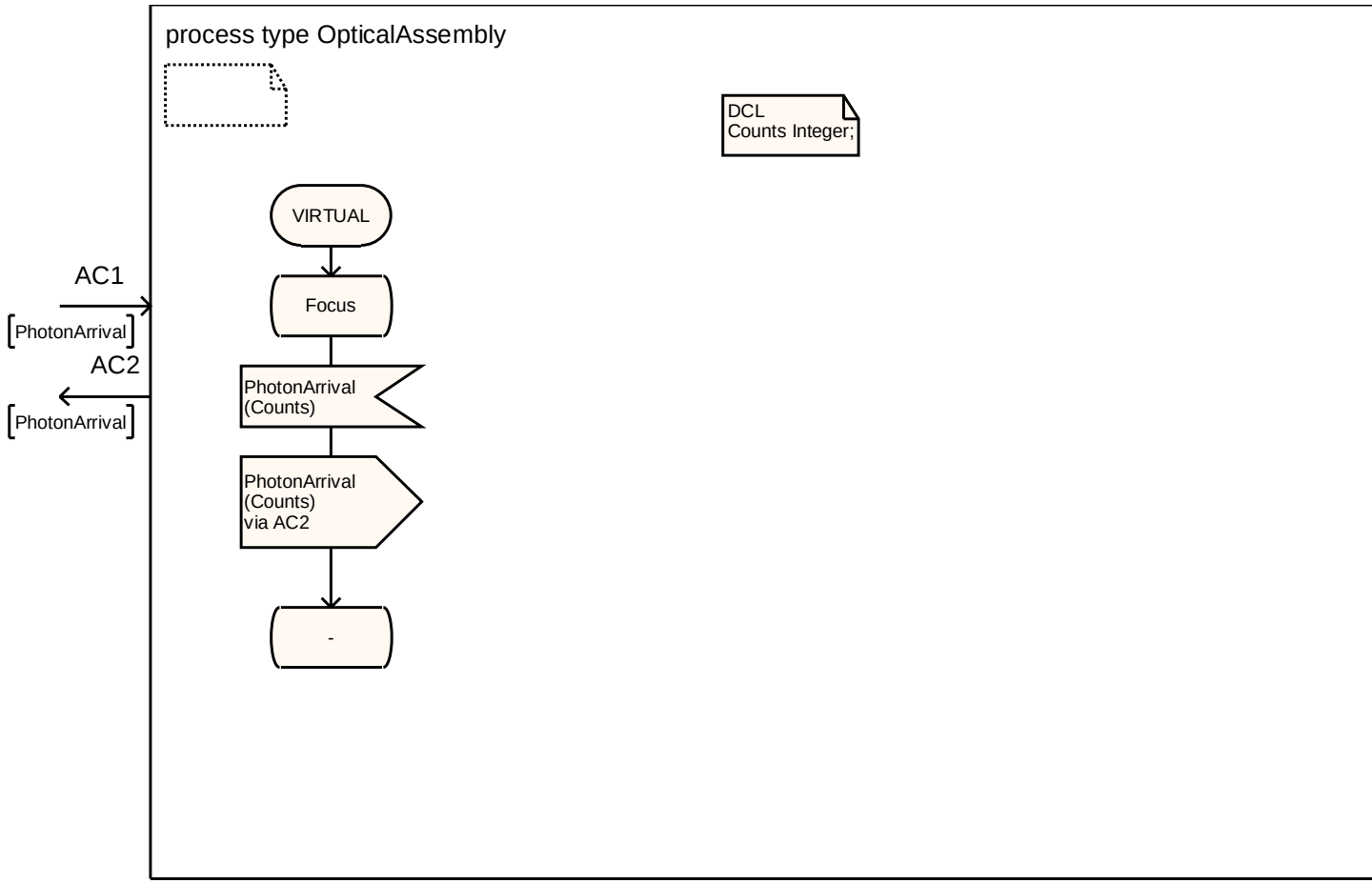

Figure 4-15: OpticalAssembly

Figure 4-16 shows the process type for Guider. It is initialized to the Idle state. When it receives a LocateGS command, it sets a 2-unit timer and transitions to Locating. When the timer expires, it outputs a GSLocated signal and transitions to Ready.

When in the Ready state, the guider can accept an AcquireGS command. It then transitions to Acquiring and waits for the GSLock signal from the environment. When that is received, it outputs the LockEstablished signal and enters the Guiding state. (A more advanced implementation would provide for a timeout in case the LockEstablished signal is never received.)

Once the guider is in the Guiding state, it remains there until it receives a DropLock command, whereupon it transitions to Idle. Again, a possible extension would accept a loss of lock signal from the environment while guiding and notify the environment of the problem.

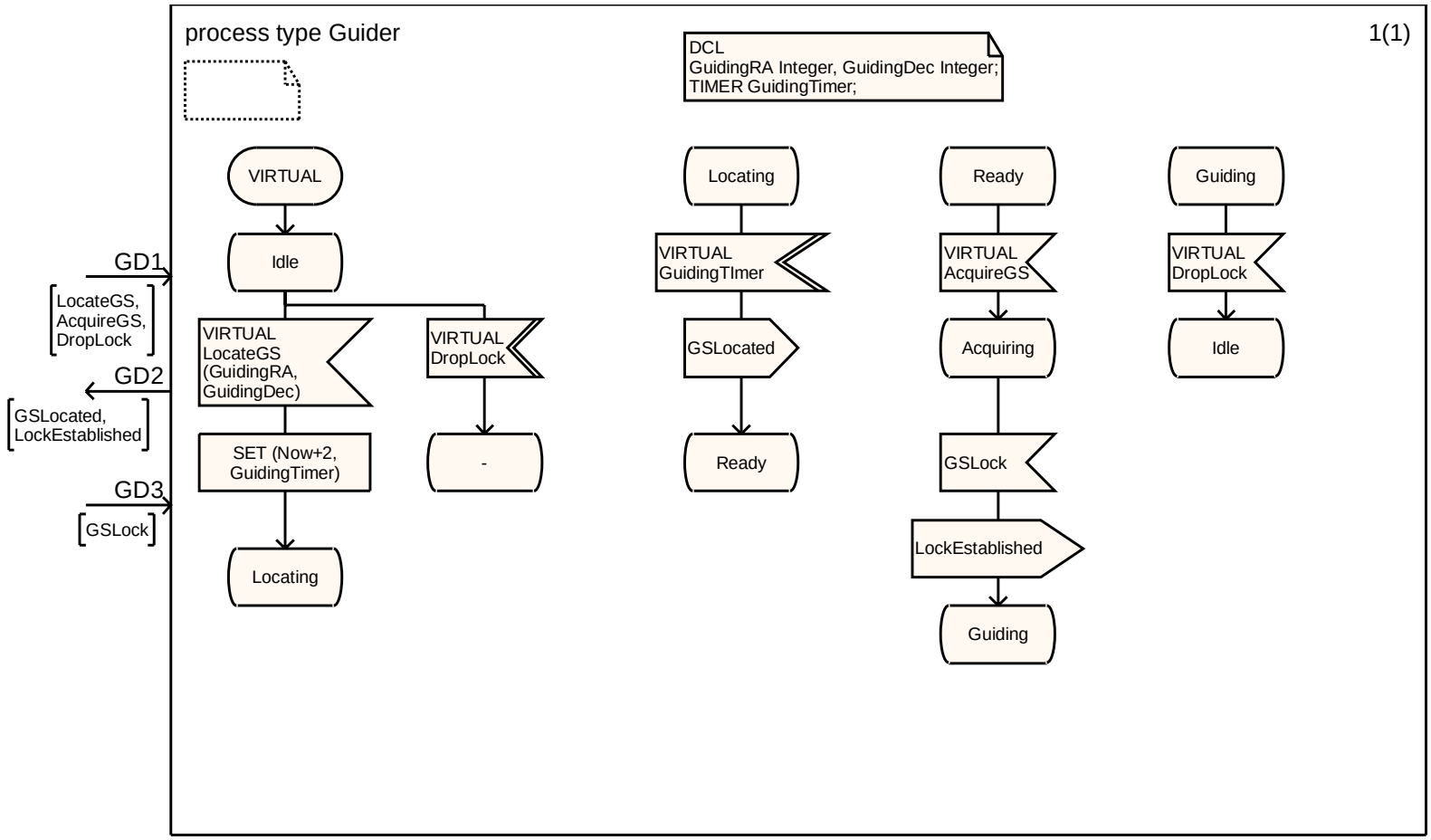
process type Guider
1(1)
DCL
GuidingRA Integer, GuidingDec Integer;
TIMER GuidingTimer;
VIRTUAL
Idle
GD1
LocateGS,
AcquireGS,
DropLock
VIRTUAL
LocateGS
(GuidingRA,
GuidingDec)
VIRTUAL
DropLock
GD2
GSLocated,
LockEstablished
SET (Now+2,
GuidingTimer)
-
GD3
GSLock
Locating
Locating
VIRTUAL
GuidingTImer
GSLocated
Ready
Ready
VIRTUAL
AcquireGS
Acquiring
GSLock
LockEstablished
Guiding
Guiding
VIRTUAL
DropLock
Idle

Figure 4-16: Guider

The InstrumentManager process is shown in Figure 4-17. This process is more complex than those above. From the InstrumentModule block, it is responsible for handling four commands (InstrumentOn, InstrumentOff, Expose, and DumpDataBuffer) and one physical signal (PhotonArrival), as well as numerous status messages. This is also the only process that is not implemented as a reusable process type, mainly because its behavior depends on the instrument configuration of the observatory, which was considered to be inherently application-specific. The initial state of InstrumentManager is Ready (meaning ready for another command). However, the InstrumentOn, InstrumentReady, InstrumentOff, and InstrumentPowerOff commands can be processed and relayed between CAM1 and the environment in any state.

When the Expose command is received, InstrumentManager sends the DropLock signal, issues a Slew command (which eventually goes to the TelescopeBlock), and transitions to Slewing. Recall that slews are modeled to take time. When the SlewComplete signal is received, InstrumentManager issues a LocateGS command to the guider and transitions to GSLocate. When the guider responds with the GSLocated signal, the process issues an AcquireGS command and transitions to GSAcquire. When the LockEstablished signal is received, InstrumentManager issues a DoExposure command to CAM1 and transitions to Exposing. Recall that this depends on receipt of a GSLock signal external to the model. If this signal is never received, InsrumentManager will be stuck in GSAcquire and the exposure will never be executed (see the above comment on the need for a timeout mechanism).

Two input signals are specific to the Exposing state: PhotonArrival and ExposureComplete (the latter has priority if both are present). The PhotonArrival signal is simply passed through to CAM1. The ExposureComplete signal is passed back to the environment and results in a transition to Ready.

When the DumpDataBuffer command is received, InstrumentManager passes it on to CAM1 and enters the Dumping state. Two inputs are specific to this state: InstrumentData (which has priority) and BufferDumpComplete. InstrumentData is passed on to the environment (and eventually the DataRecorder). BufferDumpComplete is passed back to the external environment.

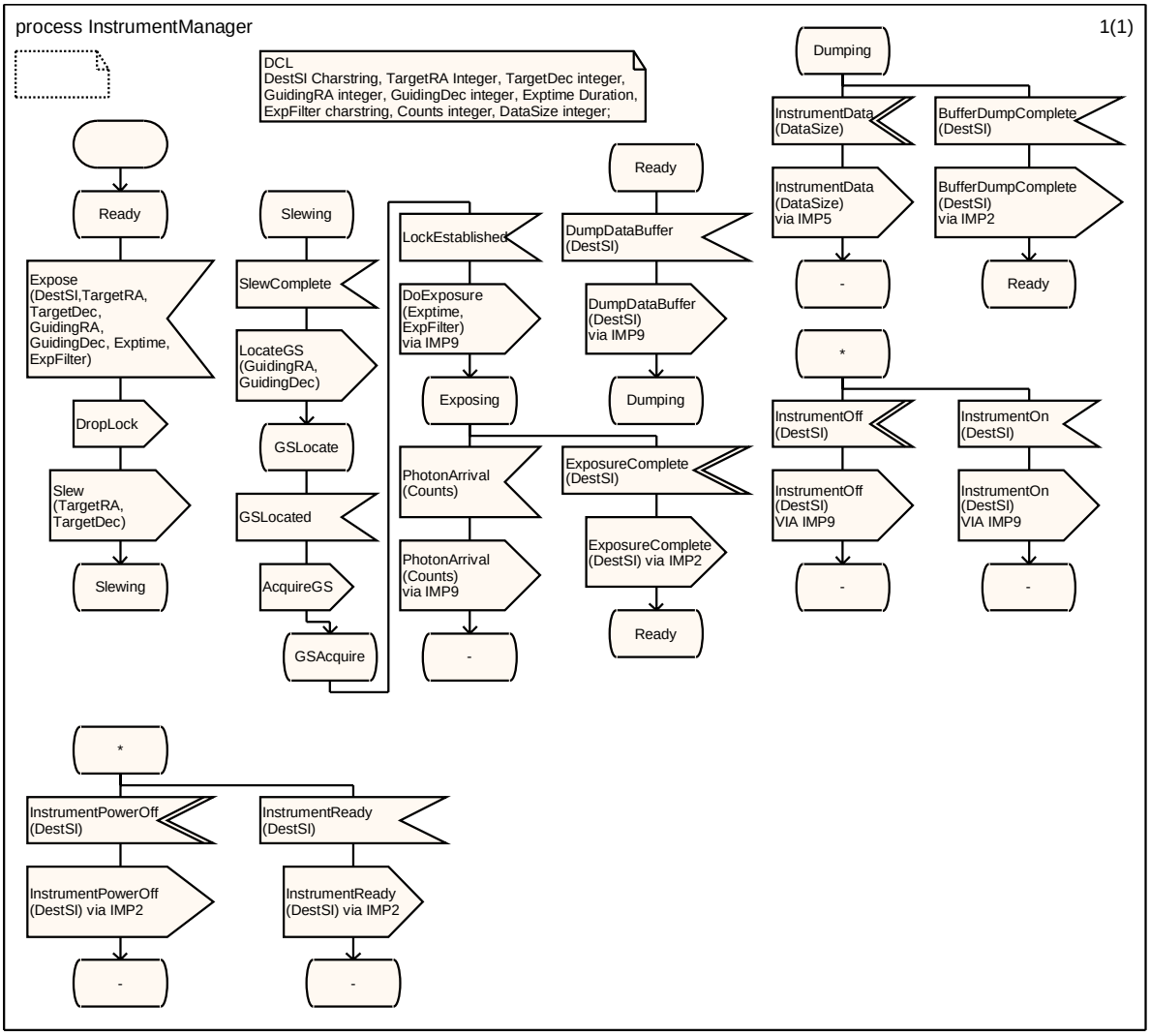
process InstrumentManager
1(1)
DCL
DestSI Charstring, TargetRA Integer, TargetDec integer,
GuidingRA integer, GuidingDec integer, Exptime Duration,
ExpFilter charstring, Counts integer, DataSize integer;
Ready
Expose
(DestSI,TargetRA,
TargetDec,
GuidingRA,
GuidingDec, Exptime,
ExpFilter)
DropLock
Slew
(TargetRA,
TargetDec)
Slewing
Slewing
SlewComplete
LocateGS
(GuidingRA,
GuidingDec)
GSLocate
GSLocated
AcquireGS
GSAcquire
LockEstablished
DoExposure
(Exptime,
ExpFilter)
via IMP9
Exposing
PhotonArrival
(Counts)
PhotonArrival
(Counts)
via IMP9
-
Ready
DumpDataBuffer
(DestSI)
DumpDataBuffer
(DestSI)
via IMP9
Dumping
ExposureComplete
(DestSI)
ExposureComplete
(DestSI) via IMP2
Ready
Dumping
InstrumentData
(DataSize)
InstrumentData
(DataSize)
via IMP5
-
BufferDumpComplete
(DestSI)
BufferDumpComplete
(DestSI)
via IMP2
Ready
*
InstrumentOff
(DestSI)
InstrumentOff
(DestSI)
VIA IMP9
-
InstrumentOn
(DestSI)
InstrumentOn
(DestSI)
VIA IMP9
-
*
InstrumentPowerOff
(DestSI)
InstrumentPowerOff
(DestSI) via IMP2
-
InstrumentReady
(DestSI)
InstrumentReady
(DestSI) via IMP2
-

Figure 4-17: InstrumentManager

Figure 4-18 shows the process for CameraManager, which manages the CAM1 instrument. The initial state is PoweredOff. When the InstrumentOn command is received, CameraManager relays it to the InstElectronics and transitions to WarmingUp. When an InstrumentReady signal is received, the process relays it to the environment and transitions to Ready.

The behavior for turning off the instrument is a little different. The InstrumentOff signal may be processed in any state, and has priority over other inputs. When it is received, CameraManager relays it to the InstElectronics, sends an InstrumentPowerOff signal to the environment without waiting for confirmation from the InstElectronics, and transitions to PoweredOff. The rationale for this deviation from the usual protocol is that there is nothing more to do: the instrument has been turned off, and the environment might as well know that immediately, particularly since it might be a safety-critical situation.

The DoExposure command, which initiates the actual exposure, can only be processed in the Ready state. The first thing that happens is an OpenShutter command and a transition to OpeningShutter. When confirmation is received that the shutter is open, CameraManager sends a PositionFilter command to the FilterAssembly and transitions to MovingFilter. When confirmation is received that the filter is in place, CameraManager sends an Integrate signal to the Detector and transitions to Exposing.

Once the exposure integration begins, the CameraManager has nothing to do until it is complete (note that it is not responsible for handling the physical PhotonArrival signals). When the ExpComplete signal is received, the process issues a CloseShutter signal and transitions to ClosingShutter. When the ShutterClosed signal is received, the process issues a ReadOut command to the Detector and transitions to ReadingOut. When the ROComplete signal is received, CameraManager issues an ExposureComplete signal to the environment and transitions back to Ready.

The remaining command to handle is DumpDataBuffer. CameraManager relays this to the DataBuffer and transitions to Dumping. When the BufferDumpComplete signal is received, CameraManager relays it to the environment and transitions to Ready.

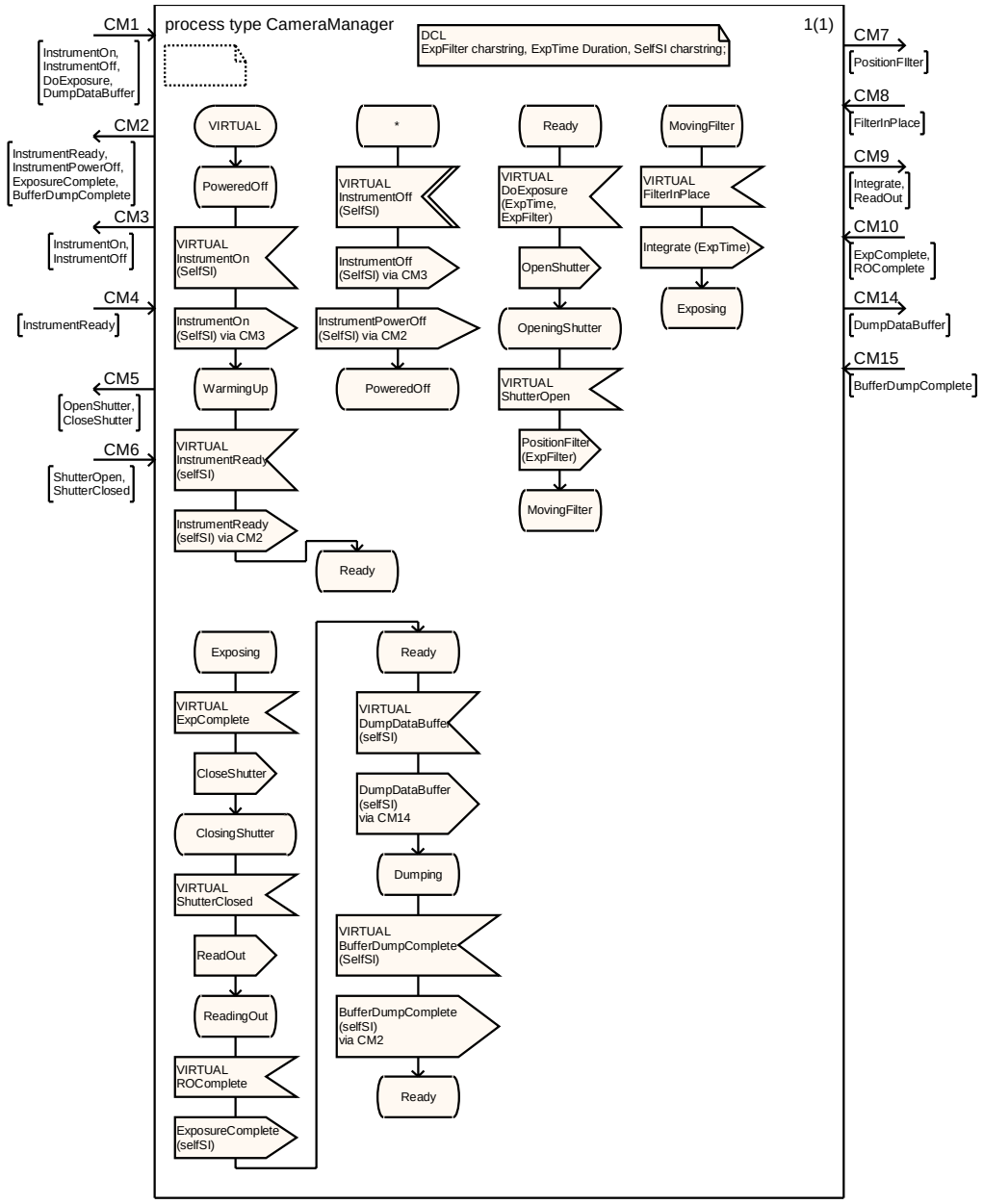
process type CameraManager
1(1)
DCL
ExpFilter charstring, ExpTime Duration, SelfSI charstring;
CM1
InstrumentOn, InstrumentOff, DoExposure, DumpDataBuffer
CM2
InstrumentReady, InstrumentPowerOff, ExposureComplete, BufferDumpComplete
CM3
InstrumentOn, InstrumentOff
CM4
InstrumentReady
CM5
OpenShutter, CloseShutter
CM6
ShutterOpen, ShutterClosed
CM7
PositionFilter
CM8
FilterInPlace
CM9
Integrate, ReadOut
CM10
ExpComplete, ROComplete
CM14
DumpDataBuffer
CM15
BufferDumpComplete
VIRTUAL
PoweredOff
VIRTUAL InstrumentOn (SelfSI)
InstrumentOn (SelfSI) via CM3
WarmingUp
VIRTUAL InstrumentReady (selfSI)
InstrumentReady (selfSI) via CM2
Ready
*
VIRTUAL InstrumentOff (SelfSI)
InstrumentOff (SelfSI) via CM3
InstrumentPowerOff (SelfSI) via CM2
PoweredOff
Ready
VIRTUAL DoExposure (ExpTime, ExpFilter)
OpenShutter
OpeningShutter
VIRTUAL ShutterOpen
PositionFilter (ExpFilter)
MovingFilter
MovingFilter
VIRTUAL FilterInPlace
Integrate (ExpTime)
Exposing
Exposing
VIRTUAL ExpComplete
CloseShutter
ClosingShutter
VIRTUAL ShutterClosed
ReadOut
ReadingOut
VIRTUAL ROComplete
ExposureComplete (selfSI)
Ready
VIRTUAL DumpDataBuffer (selfSI)
DumpDataBuffer (selfSI) via CM14
Dumping
VIRTUAL BufferDumpComplete (SelfSI)
BufferDumpComplete (selfSI) via CM2
Ready

Figure 4-18: CameraManager

The above is a description of generic behavior defined for the CameraManager class. However, in order to experiment with inheritance in SDL, the CAM1 block actually defines the CameraManager process as an instance of a specialized type, called HomingCameraManager. The latter is defined as a very simple fragment of behavior (Figure 4-19). Recall that the basic CameraManager, upon receiving the ExpComplete signal while Exposing, simply issued the CloseShutter command and transitioned to ClosingShutter. Any behavior specified for a state/input combination in an FSM essentially amounts to a method, which may be overridden.

In HomingCameraManager, the intent to override behavior is indicated by prefacing the ExpComplete input (which was defined as **VIRTUAL** in the superclass) with **REDEFINED**. The redefined behavior is to issue a HomeFilter command to the FilterAssembly, which returns the filter wheel to a default state which is appropriate when not exposing. When the FilterAssembly completes the motion and sends the FilterInPlace signal, HomingCameraManager proceeds with the default behavior by issuing the CloseShutter command and transitioning to ClosingShutter. (Actually, it would have been better to close the shutter first and insert the filter homing command only after the ShutterClosed signal is received.)

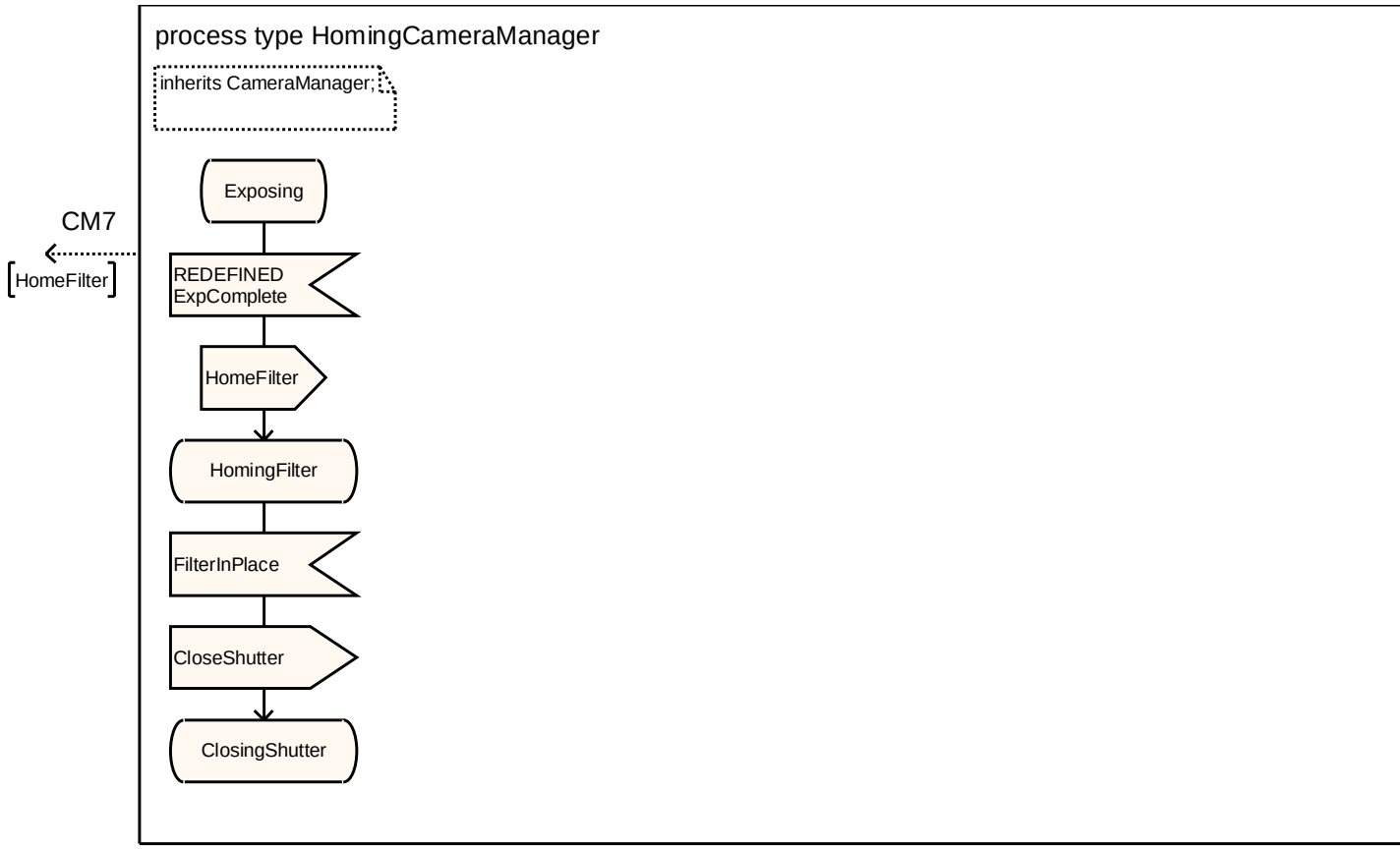

Figure 4-19: HomingCameraManager

The remaining processes, for the instrument components, are simpler. Figure 4-20, for the InstElectronics, handles the InstrumentOn and InstrumentOff commands. The initial state of the instrument is Cold. When the InstrumentOn command is received, the process sets a timer for

100 time units and transitions to WarmingUp. When the timer expires, the InstrumentReady signal is issued and the process enters the Warm state.

A priority input, which can be processed either in Warm or WarmingUp, is the InstrumentOff command. The response is simply to transition to Cold (no output needed).

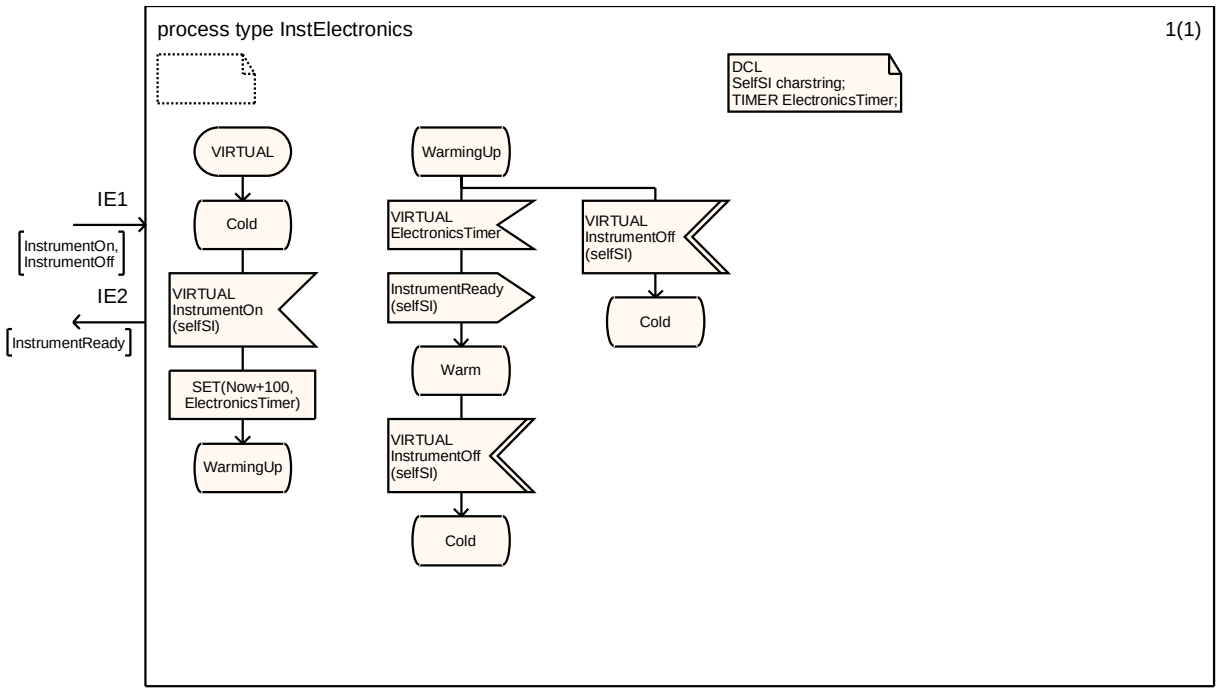

Figure 4-20: InstElectronics

Figure 4-21 shows the Shutter process, which is responsible for processing the OpenShutter and CloseShutter commands and the PhotonArrival physical signal. The initial state is Closed. The OpenShutter command results in an immediate ShutterOpen confirmation signal and a transition to Open. The CloseShutter command results in an immediate ShutterClosed confirmation signal and a transition to Closed.

The PhotonArrival signal can be processed and passed through to the FilterAssembly only when the shutter is Open. This signal has priority over CloseShutter (a mistake, because if there were a continuous stream of photons it would mean the shutter could never be closed!).

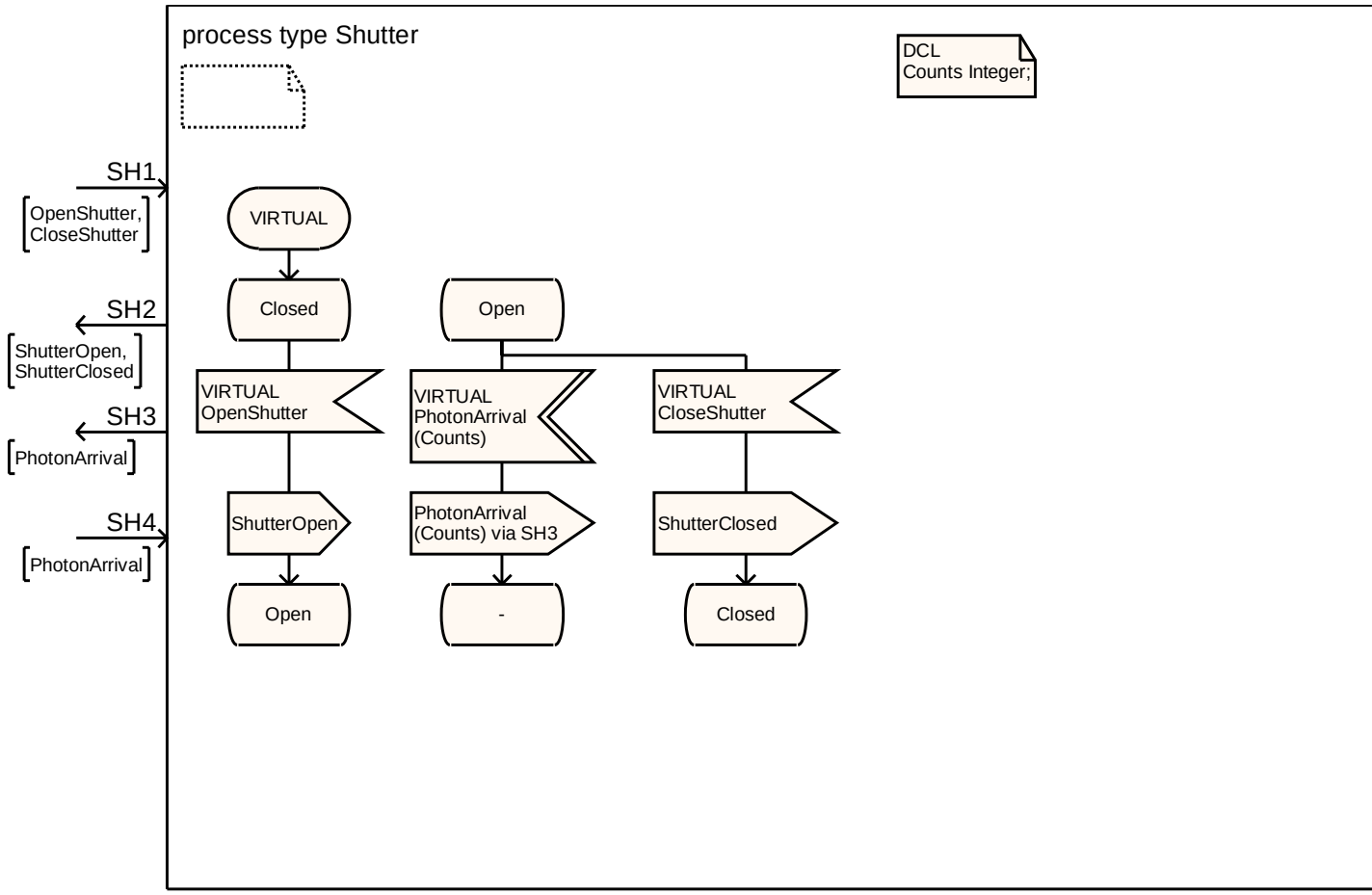

Figure 4-21: Shutter

Figure 4-22 shows the generic process type FilterSubsystem. This process type handles one command, PositionFilter, and one physical signal, PhotonArrival. The process keeps track of the filter position, which is initially UNKNOWN. The state is always Positioned (there is no modeling of the time to move the filter). When the PositionFilter command is received, the new position is saved as the current position and the FilterInPlace signal is issued. The FilterSubsystem is modeled as always transmitting photons, so when the priority PhotonArrival signal is received, it is passed through to the Detector. Once again it is a mistake to give priority to physical signals over commands: the PositionFilter command would be blocked as long as there was a PhotonArrival signal to process.

In the CAM1 block, however, the FilterAssembly is declared to be an instance of CAM1FilterSubsystem (Figure 4-23), which specializes FilterSubsystem in order to support the HomeFilter command. The initialization of the filter position is redefined to CLEAR_FILTER. When the HomeFilter command is received, the current position is reset to CLEAR_FILTER and the FilterInPlace signal is issued. Note that there is no need to declare this input as **REDEFINED**, since it was not supported in the FilterSubsystem superclass. Note also that there is no need to declare the CurrentPosition local variable, since it was declared in the superclass.

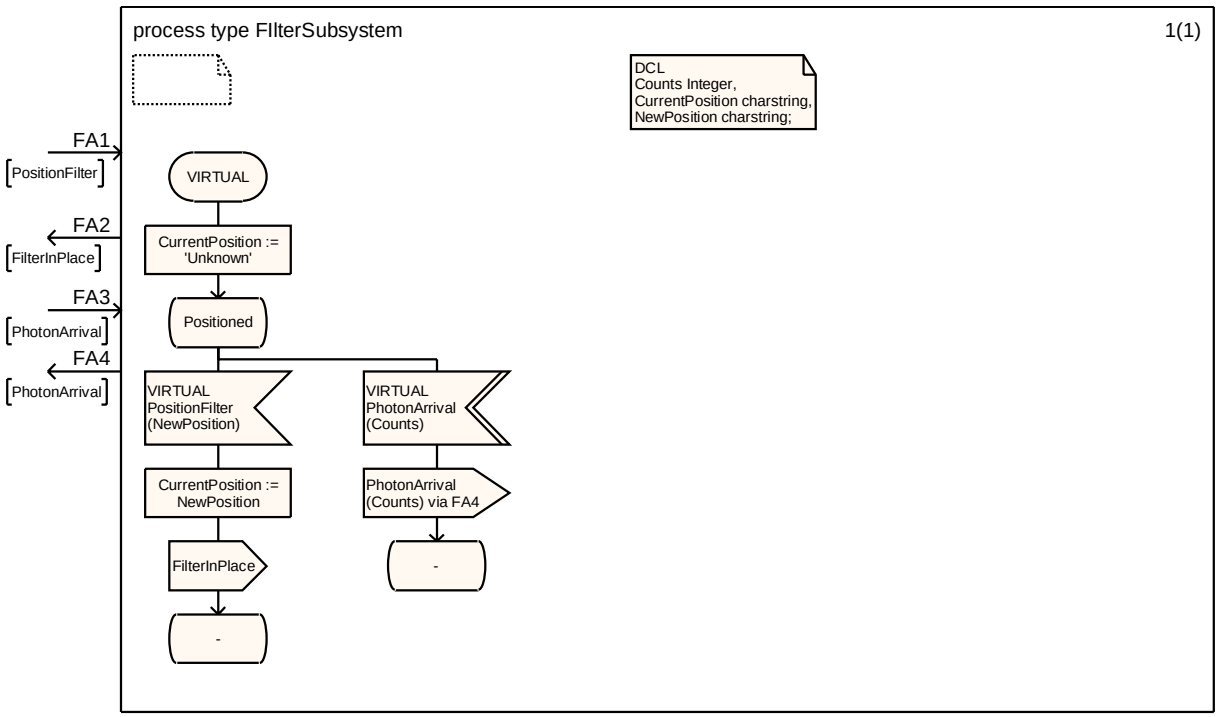

Figure 4-22: FilterSubsystem

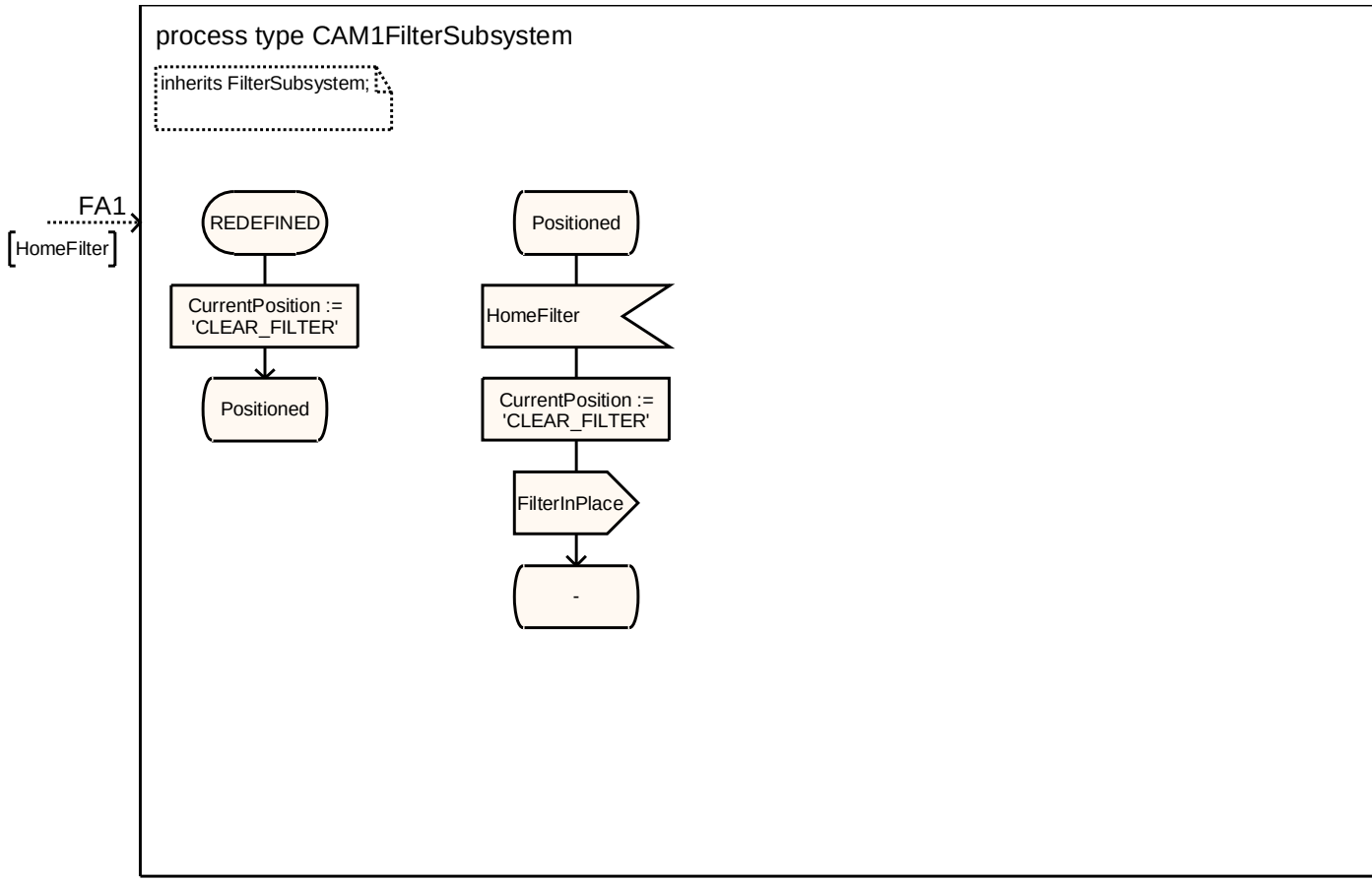

Figure 4-23: CAM1FilterSubsystem

Figure 4-24 shows the process for Detector, which is responsible for two commands, Integrate and ReadOut, and one physical signal, PhotonArrival. The Detector keeps track of the photons received during the exposure. Upon initialization, it sets this count to 0 and enters the Idle state.

When the Integrate command is received, the Detector sets a timer for the ExpTime that was specified as an argument to the command, and transitions to Integrating. (Note that this is the first time in this project that a timer was set with a variable duration; this will only work if the variable is declared to be of the special type Duration.)

When the Detector is in the Integrating state, the physical PhotonArrival signal finally reaches a place where it can be processed (again it is a mistake to make it a priority input). The Detector increments the count by the number of photons received. When the exposure timer expires, the Detector issues the ExpComplete signal to the CameraManager and transitions to DataAvailable. At this point the photon data is stored on the physical detector, but not yet in buffer memory.

When the ReadOut command is received, the Detector issues the InstrumentData signal with the total number of counts, transferring the data to the DataBuffer. The Detector then resets the number of counts to 0, issues the ROComplete signal to the CameraManager, and transitions to Idle, making it ready for a new exposure.

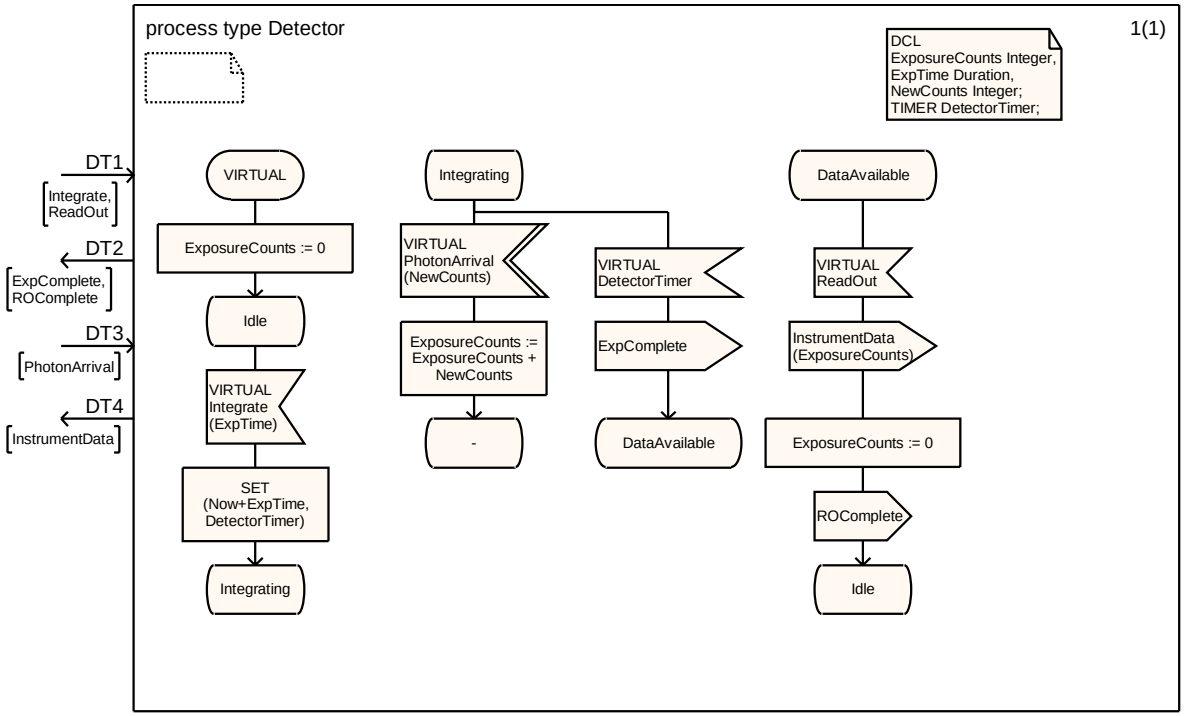

Figure 4-24: Detector

The last process, DataBuffer, is shown in Figure 4-25. This process is similar to Recorder. The buffer keeps track of the number of photons it stores, initially zero. The initial state is Empty. When the InstrumentData signal is received in this state, the DataBuffer sets the buffer count to the size of the instrument data from the signal and transitions to Readable. In this state, additional InstrumentData signals may be processed; these result in the buffer count being incremented by the size of input data.

When the priority input DumpDataBuffer is received in the Readable state, the process issues the InstrumentData signal to dump the buffer to the DataRecorder and resets the buffer count to 0, issues the BufferDumpComplete signal to the CameraManager, and transitions to Empty.

Figure 4-26 shows the finished model as it appears in the Tau/SDL Organizer window.

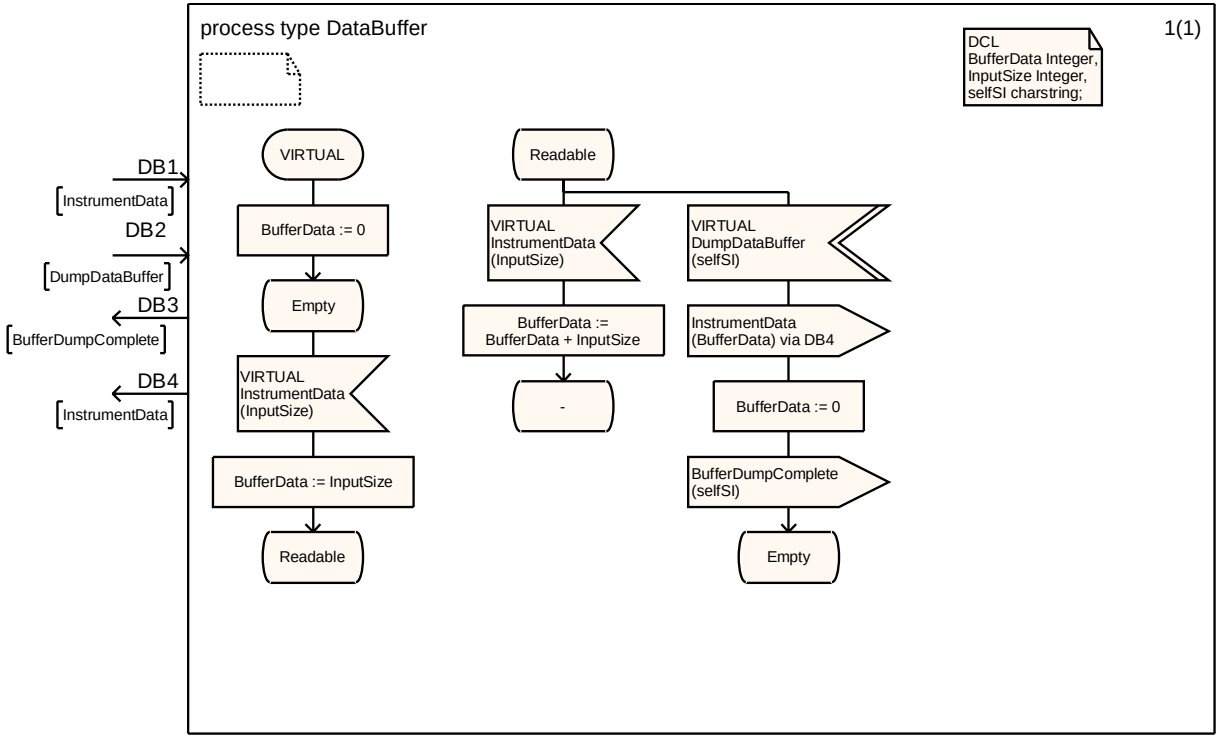

Figure 4-25: DataBuffer

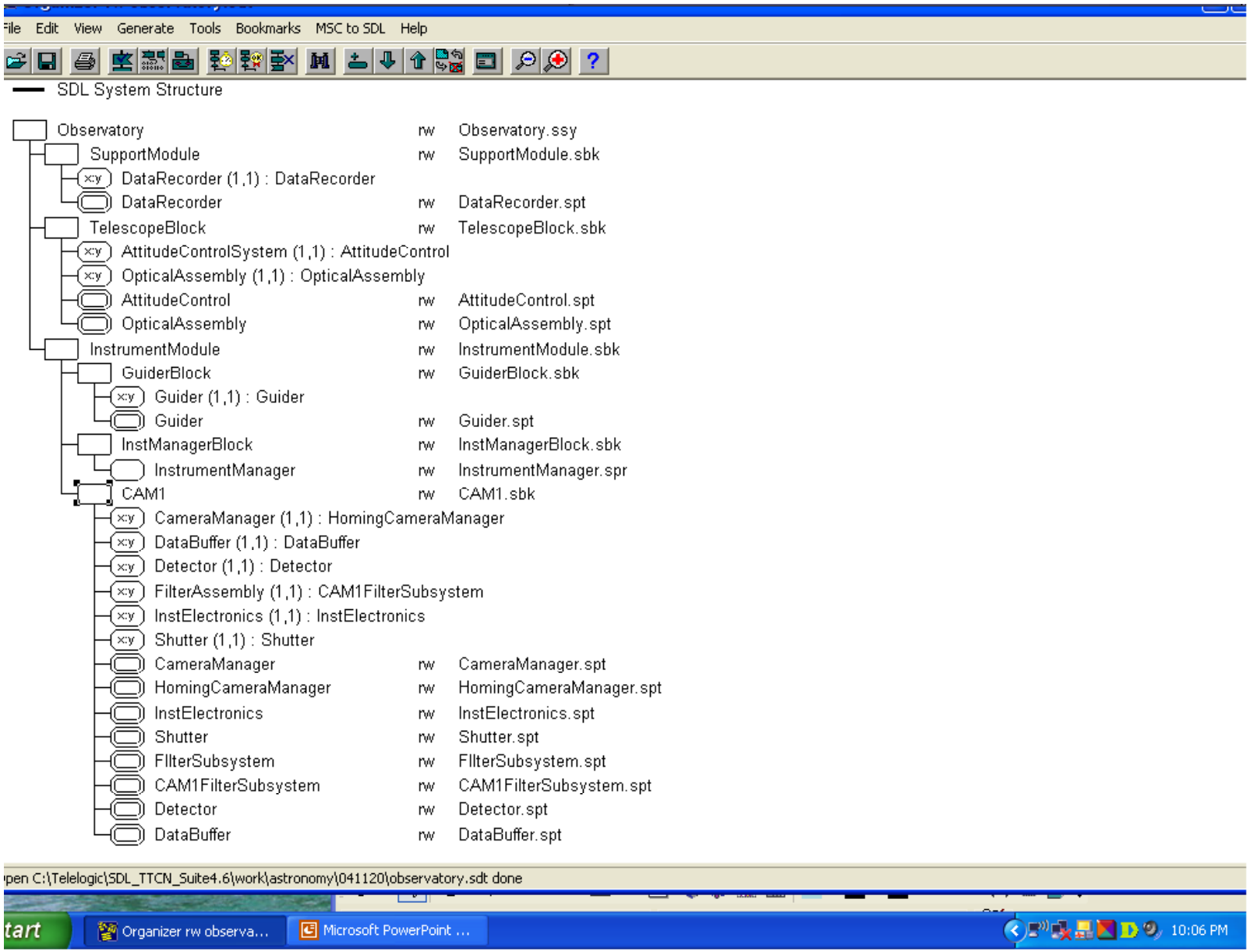
File Edit View Generate Tools Bookmarks MSC to SDL Help
SDL System Structure
Observatory rw Observatory.ssy
SupportModule rw SupportModule.sbk
DataRecorder (1,1) : DataRecorder
DataRecorder rw DataRecorder.spt
TelescopeBlock rw TelescopeBlock.sbk
AttitudeControlSystem (1,1) : AttitudeControl
OpticalAssembly (1,1) : OpticalAssembly
AttitudeControl rw AttitudeControl.spt
OpticalAssembly rw OpticalAssembly.spt
InstrumentModule rw InstrumentModule.sbk
GuiderBlock rw GuiderBlock.sbk
Guider (1,1) : Guider
Guider rw Guider.spt
InstManagerBlock rw InstManagerBlock.sbk
InstrumentManager rw InstrumentManager.spr
CAM1 rw CAM1.sbk
CameraManager (1,1) : HomingCameraManager
DataBuffer (1,1) : DataBuffer
Detector (1,1) : Detector
FilterAssembly (1,1) : CAM1FilterSubsystem
InstElectronics (1,1) : InstElectronics
Shutter (1,1) : Shutter
CameraManager rw CameraManager.spt
HomingCameraManager rw HomingCameraManager.spt
InstElectronics rw InstElectronics.spt
Shutter rw Shutter.spt
FIlterSubsystem rw FIlterSubsystem.spt
CAM1FilterSubsystem rw CAM1FilterSubsystem.spt
Detector rw Detector.spt
DataBuffer rw DataBuffer.spt
pen C:\Telelogic\SDL_TTCN_Suite4.6\work\astronomy\041120\observatory.sdt done
tart
Organizer rw observa...
Microsoft PowerPoint ...
10:06 PM

Figure 4-26: Observatory Model

4.6 Bugs and Limitations

The following modeling problems (listed in decreasing order of severity) were encountered during this project. I was able to work around all of them, but they are documented for future reference. Some of these may be due to my own lack of famliarity with SDL rather than a deficiency in the language or the Tau/SDL tool.

- **Real Numbers**. Use of real numbers in SDL caused the executable application to crash. This also happened in some of Telelogic's own tutorials, but not in the main "DemonGame" tutorial, which uses only integers. The workaround was to redeclare all reals as integers. This detracted from the realism of the simulation in a few places, but did not prevent any logical paths from being modeled.

- **Generic Dispatch**. The instrument commands from the observer specifiy the instrument being used. The intent was that the InstrumentManager process would dispatch to the process designated by that argument. However, I was unable to find a way to do this in SDL. The ID of a dynamically created process can be saved when the process is created, but I couldn't figure out how to invoke a static process by reference (character strings didn't work). This didn't matter for the project, since there is only one instrument anyway, but in a real observatory there would typically be several instruments and it would be desirable to address them directly rather than having to go through the equivalent of an extended case statement.

- **Services**. One problem with graphical programming is the need to keep diagrams small enough to fit on a page. For the most complex processes, such as InstrumentManager and CameraManager, it would have been desirable to break them up into separate diagrams. SDL is supposed to offer a construct called "services" that is designed for this purpose[25], but Tau/SDL did not appear to support it.

- **Specialization to instances**. There does not appear to be a way to specialize a process type to a process instance, rather than having to define a subtype. This would have been very useful for the specialization of FilterSubsystem. The process that accepts the "home filter" command cannot really be generic (because the home position must be specified, and this could be different for each instrument), but I could not specialize to an instance, so I had to create the artificial process type CAM1FilterSubsystem.

5 Validation and Verification Using Executable Model

5.1 System Simulation

Once an executable application is generated from a model, Tau/SDL brings up a Simulator user interface, which can be customized for the particular application. The user can send signals from the environment, single step through the model, run until there are no more transitions to execute, or run until a certain value of the modeled system time. Breakpoints can be set at interesting points in the process. Execution can be traced textually at a variety of verbosity settings and graphically by bringing up the applicable SDL diagram when execution is stopped. Another means of tracing execution is to record a detailed MSC of all process state transitions and signals exchanged between processes and between processes and the environment.

Commonly sent signals are saved for convenience using a macro facility, which is invoked via a button. Figure 5-1 shows a typical screen shot during the Observatory simulation. The SDL diagram for Detector is shown at the point it consumes the Integrate signal.

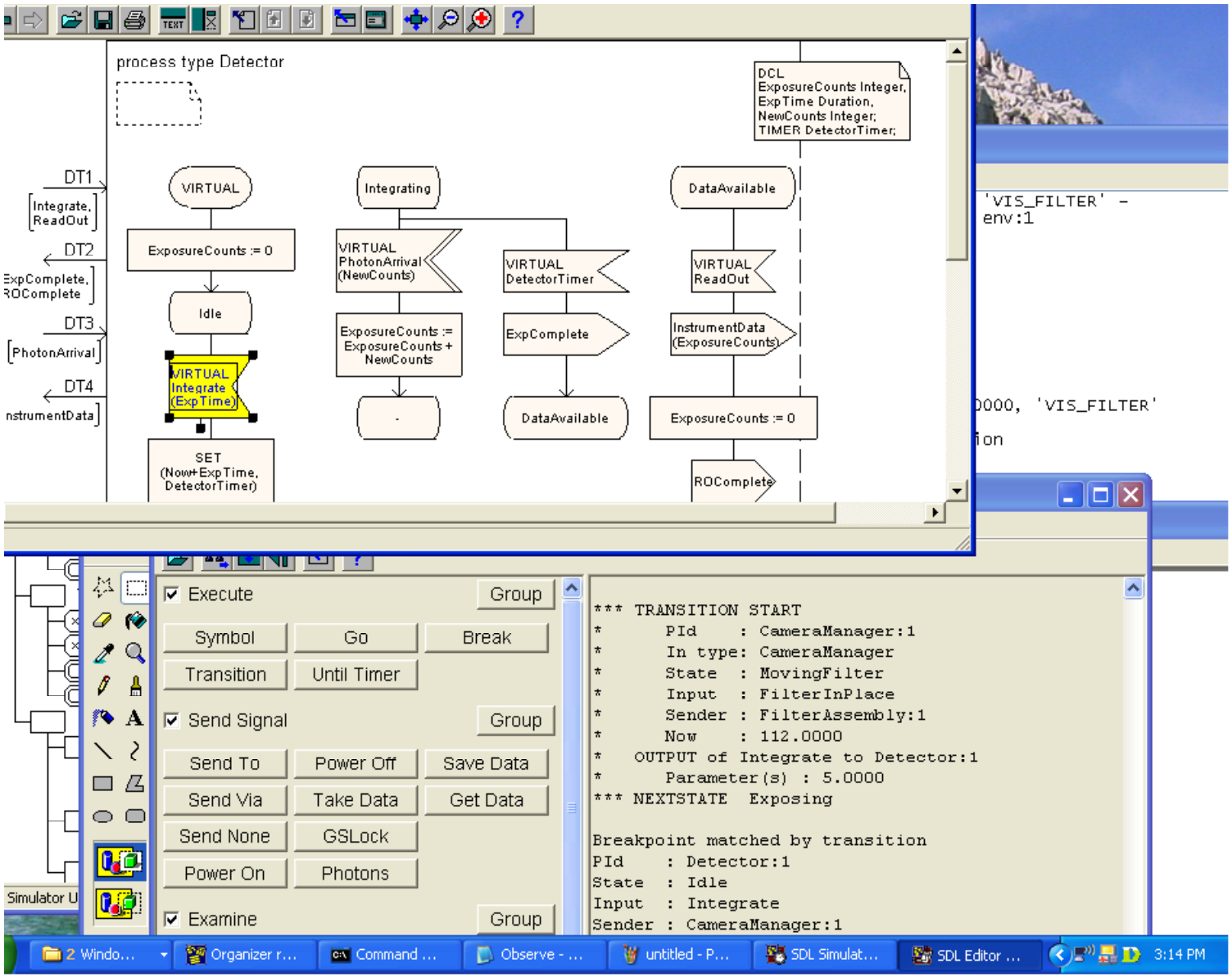
process type Detector
DCL
ExposureCounts Integer,
ExpTime Duration,
NewCounts Integer;
TIMER DetectorTimer;
DT1
Integrate,
ReadOut
DT2
ExpComplete,
ROComplete
DT3
PhotonArrival
DT4
nstrumentData
VIRTUAL
ExposureCounts := 0
Idle
VIRTUAL
Integrate
(ExpTime)
SET
(Now+ExpTime,
DetectorTimer)
Integrating
VIRTUAL
PhotonArrival
(NewCounts)
ExposureCounts :=
ExposureCounts +
NewCounts
VIRTUAL
DetectorTimer
ExpComplete
DataAvailable
DataAvailable
VIRTUAL
ReadOut
InstrumentData
(ExposureCounts)
ExposureCounts := 0
ROComplete
'VIS_FILTER' -
env:1
0000, 'VIS_FILTER'
ion
Execute
Group
Symbol
Go
Break
Transition
Until Timer
Send Signal
Group
Send To
Power Off
Save Data
Send Via
Take Data
Get Data
Send None
GSLock
Power On
Photons
Examine
Group
Simulator U
*** TRANSITION START
* PId : CameraManager:1
* In type: CameraManager
* State : MovingFilter
* Input : FilterInPlace
* Sender : FilterAssembly:1
* Now : 112.0000
* OUTPUT of Integrate to Detector:1
* Parameter(s) : 5.0000
*** NEXTSTATE Exposing
Breakpoint matched by transition
PId : Detector:1
State : Idle
Input : Integrate
Sender : CameraManager:1
2 Windo...
Organizer r...
Command ...
Observe - ...
untitled - P...
SDL Simulat...
SDL Editor ...
3:14 PM

Figure 5-1: Observatory Simulator User Interface

The simulator was used to "sanity check" the model using a simple observing scenario with a single photon event (Appendix A). The scenario begins with the instrument on. The Expose command is sent from the environment and the **Go** command is given. The model runs until the guider reaches the Acquiring state and then stops, because as anticipated, it has to wait for a GSLock signal from the environment. After that signal is sent followed by another **Go** command, the system runs until it hits the Integrate breakpoint shown in Figure 5-1. This breakpoint is necessary because the PhotonArrival events have to come in from the environment. If the simulation is not stopped, the exposure would just complete with no data.

(Note: I considered including the target and guiding source in the model in order to automate this part of the simulation. This would have also had advantages in simulating failure scenarios. Instead of sending a loss of lock signal to the guider, for example, it would be possible to capriciously delete the guiding source and let the guider process itself conclude that lock had been lost. However, I decided against this due to the added complexity and because I wasn't sure about the conceptual implications of trying to validate a model that includes elements external to the system.)

After the breakpoint was hit, the system was run for two more time units and then given a PhotonArrival signal with 10 counts, followed by another **Go** command. This time the system runs until the ExposureComplete signal is returned to the environment. After sending a DumpDataBuffer command followed by **Go**, the system returns the expected BufferDumpComplete signal. The DownloadData command followed by **Go** then results in a DataAvailable signal with 10 counts.

A detailed process-level MSC was also generated for this simulation. The diagram is not shown because it is much larger than a single page, but it amounts to a graphical representation of the trace in Appendix A.

5.2 Architecture Validation

The next step is to validate the model in a more automated way, using a separate executable application called a Validator. The difference is that the simulator is used to **run** the model under specified inputs, while the purpose of the Validator is to explore the **potential** behavior of the model using state-space search techniques ("what-if" analysis). The Validator can be used both to explore the system in an unguided mode looking for errors, and to perform a guided exploration in order to check whether a given system-level MSC is satisfiable.

Figure 5-2 shows the Validator user interface after initialization. All 11 processes have been created and set to the start state.

The first thing to try with a Validator is to perform an unguided state-space search and look for errors. Tau/SDL explores the space at random up to a predefined depth (which allows the user to trade off thoroughness and speed). The initial run generated seven reports with warnings on

"implicit signal consumption", as shown in Figure 5-3. This means that a signal can be sent to a process when it is not in a state that accepts input from that signal, so the signal is ignored. (I didn't realize this was a problem when I developed the model, but I presume it is bad SDL practice, else the tool wouldn't go to the trouble of warning me about it.)

Validator reports are represented as process-level MSCs giving a graphical trace of the sequence of events that led to the problem. One of the seven reports (selected for simplicity because it was wholly internal to a process) is detailed in Figure 5-4. Note that this MSC has 12 columns, one for each process plus one for the environment. (There is an unimportant error that prevents the initial state from being shown for the DataBuffer and DataRecorder processes.) In this scenario, the instrument is turned on and then turned off again before it has warmed up, so that when the warmup timer expires in the InstElectronics process, the process is in the Cold state where it cannot process the signal. This is the correct behavior (we certainly don't want the instrument going on after it has been turned off), but it would have been cleaner to reset the timer after turning off the instrument so that the signal didn't get generated in the first place.

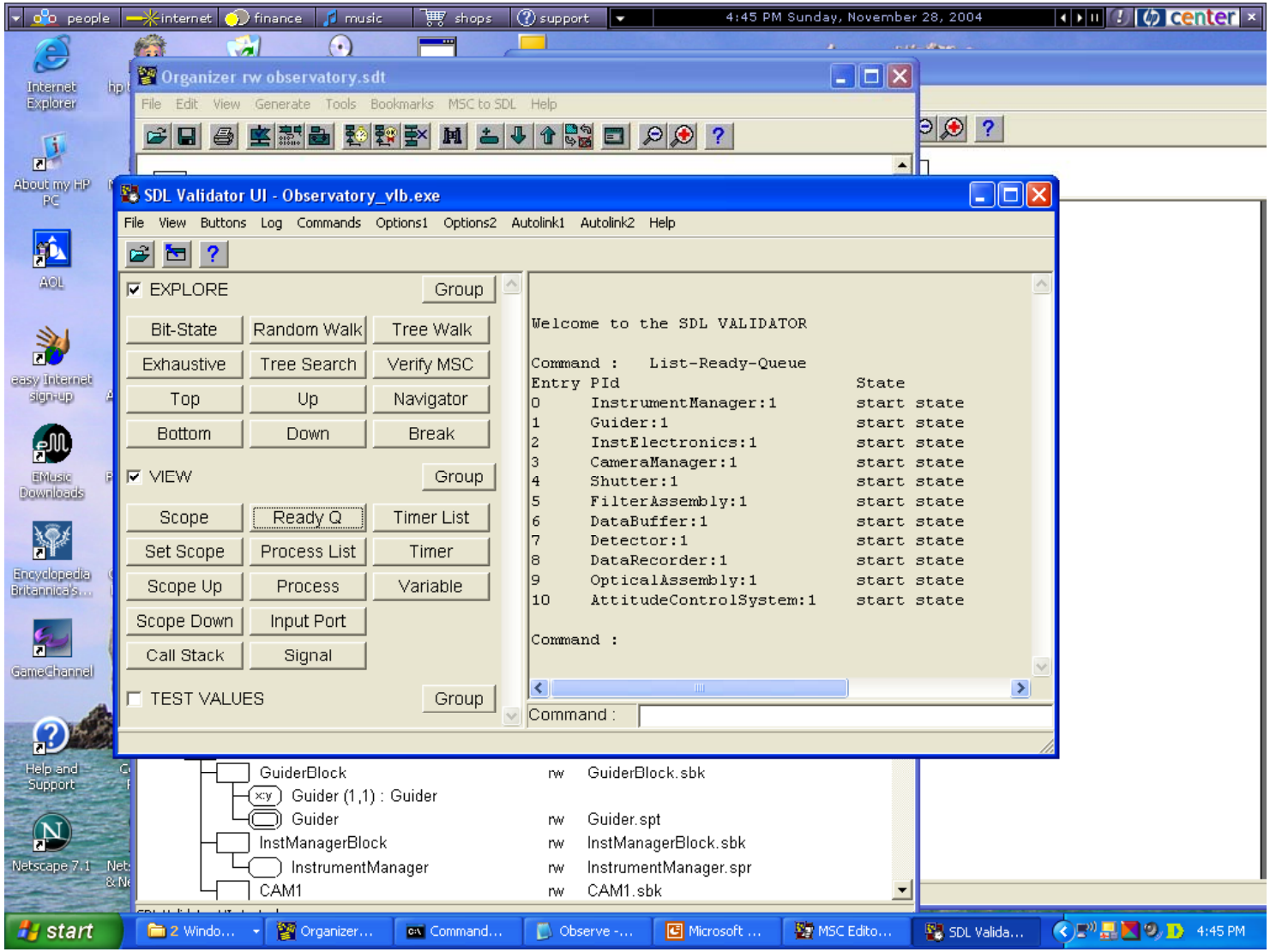

Figure 5-2: Observatory Validator User Interface

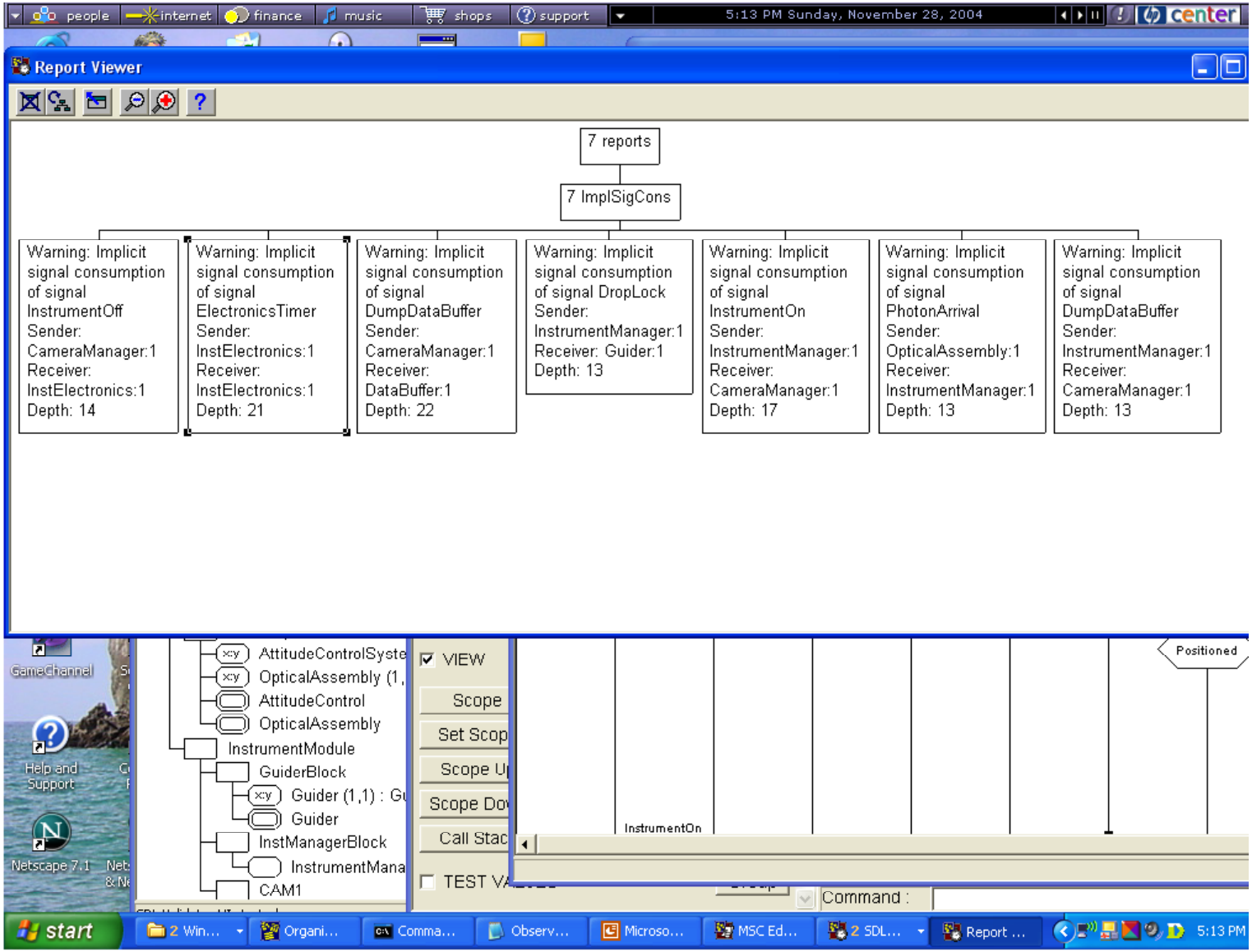

Figure 5-3: Observatory Validator Report Summary

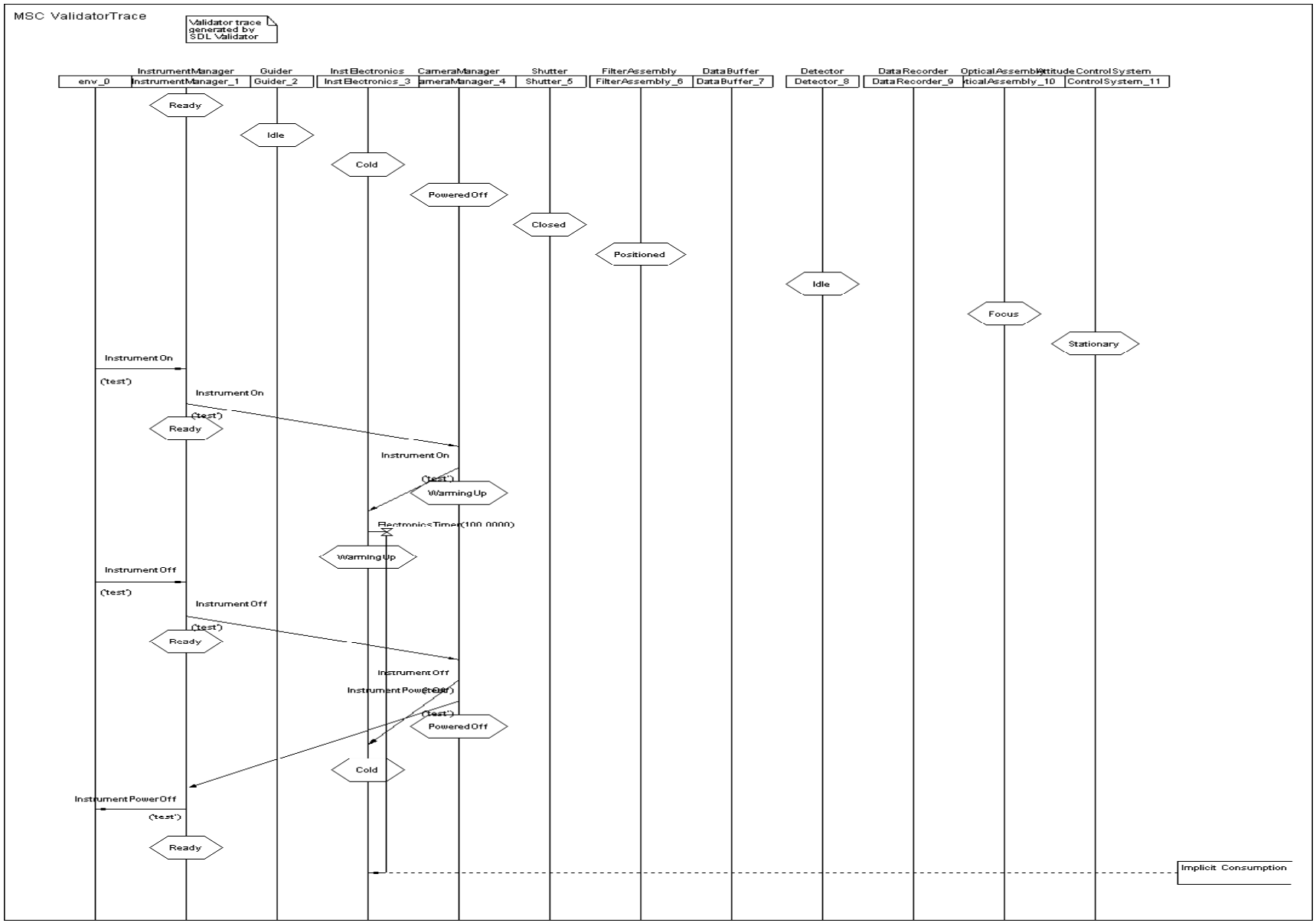
MSC ValidatorTrace
Validator trace generated by SDL Validator
env_0
InstrumentManager_1
Guider_2
InstElectronics_3
CameraManager_4
Shutter_5
FilterAssembly_6
DataBuffer_7
Detector_8
DataRecorder_9
OpticalAssembly_10
AttitudeControlSystem_11
Ready
Idle
Cold
PoweredOff
Closed
Positioned
Idle
Focus
Stationary
InstrumentOn
('test')
InstrumentOn
('test')
Ready
InstrumentOn
('test')
WarmingUp
ElectronicsTimer(100.0000)
WarmingUp
InstrumentOff
('test')
InstrumentOff
('test')
Ready
InstrumentOff
InstrumentPowerOff
('test')
PoweredOff
Cold
InstrumentPowerOff
('test')
Ready
Implicit Consumption

Figure 5-4: MSC Report on Implicit Signal Consumption

The tool also reports the percentage of "symbol coverage" (that is, paths executed in the model) as the result of a search. Unfortunately, for this particular model I was unable to get coverage greater than 36% as the result of unguided exploration. It wasn't an issue of maximum search depth; the algorithm simply completed its search and terminated without exploring most of the model.

The next step is to validate the model against the use cases (system-level MSCs) developed at the start of the process (secton 4.2). The tool searches for a path through the state space that will produce the expected outputs given the specified inputs. Figure 5-5 gives a screen shot of the validation results for the trivial use case TurnOnInstrument. The tool reports that the MSC is successfully verified (it is a matter of semantics whether one thinks of this as verifying the MSC or validating the model) and produces a report in the form of a detailed process-level MSC that shows the execution trace that would be produced by running the scenario. Since this is **not** a simulation, there are no side effects from this search.

Figure 5-6 shows the validation results for the main success use case Observe. Once again the system reports successful validation and produces an MSC report to demonstrate it, though in this case some other reports are produced as well. The tool reports 86% symbol coverage, demonstrating that the predefined scenario is far more effective in exercising the system than unguided exploration.

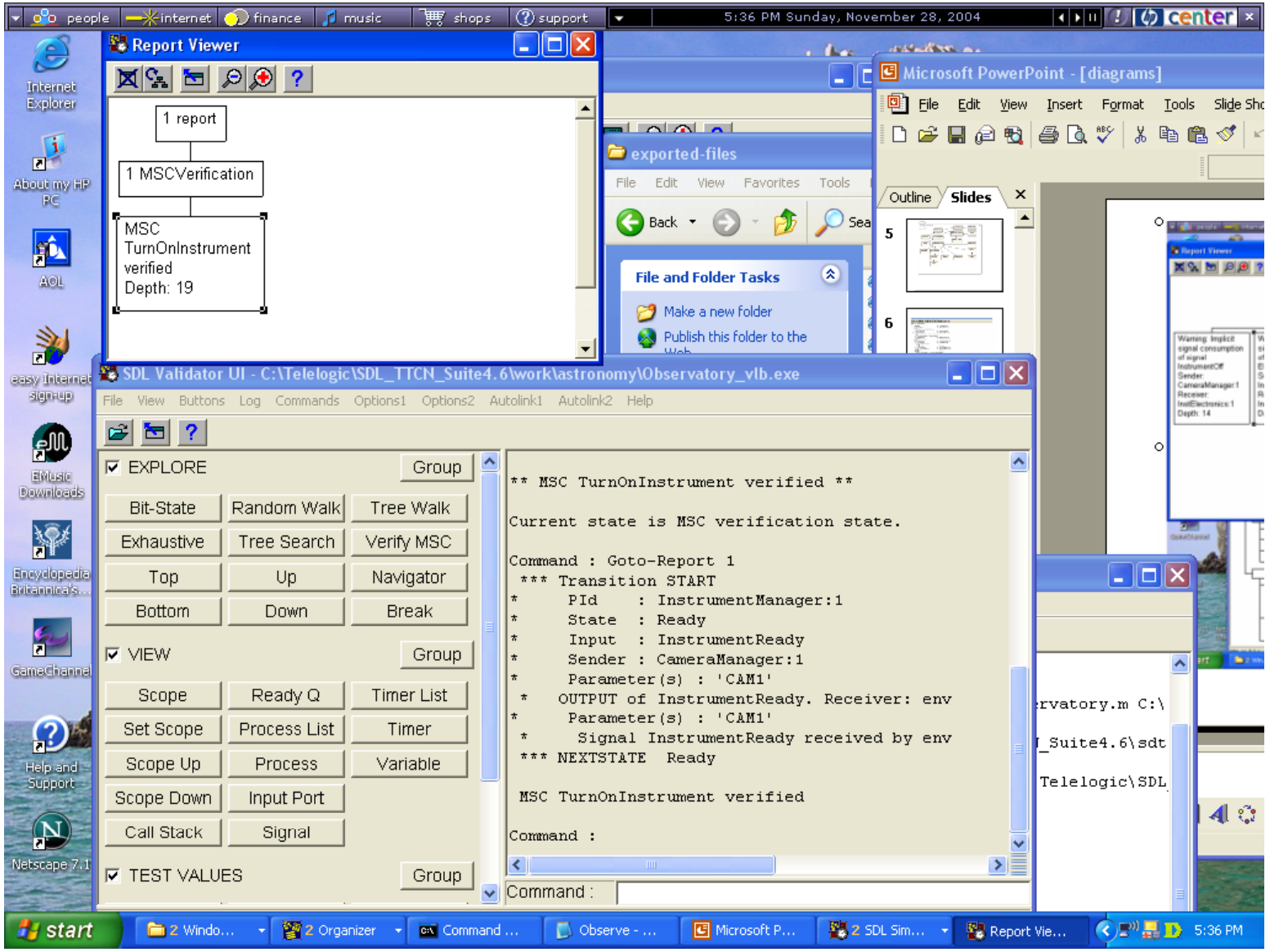

Figure 5-5: Model Validation against MSC TurnOnInstrument

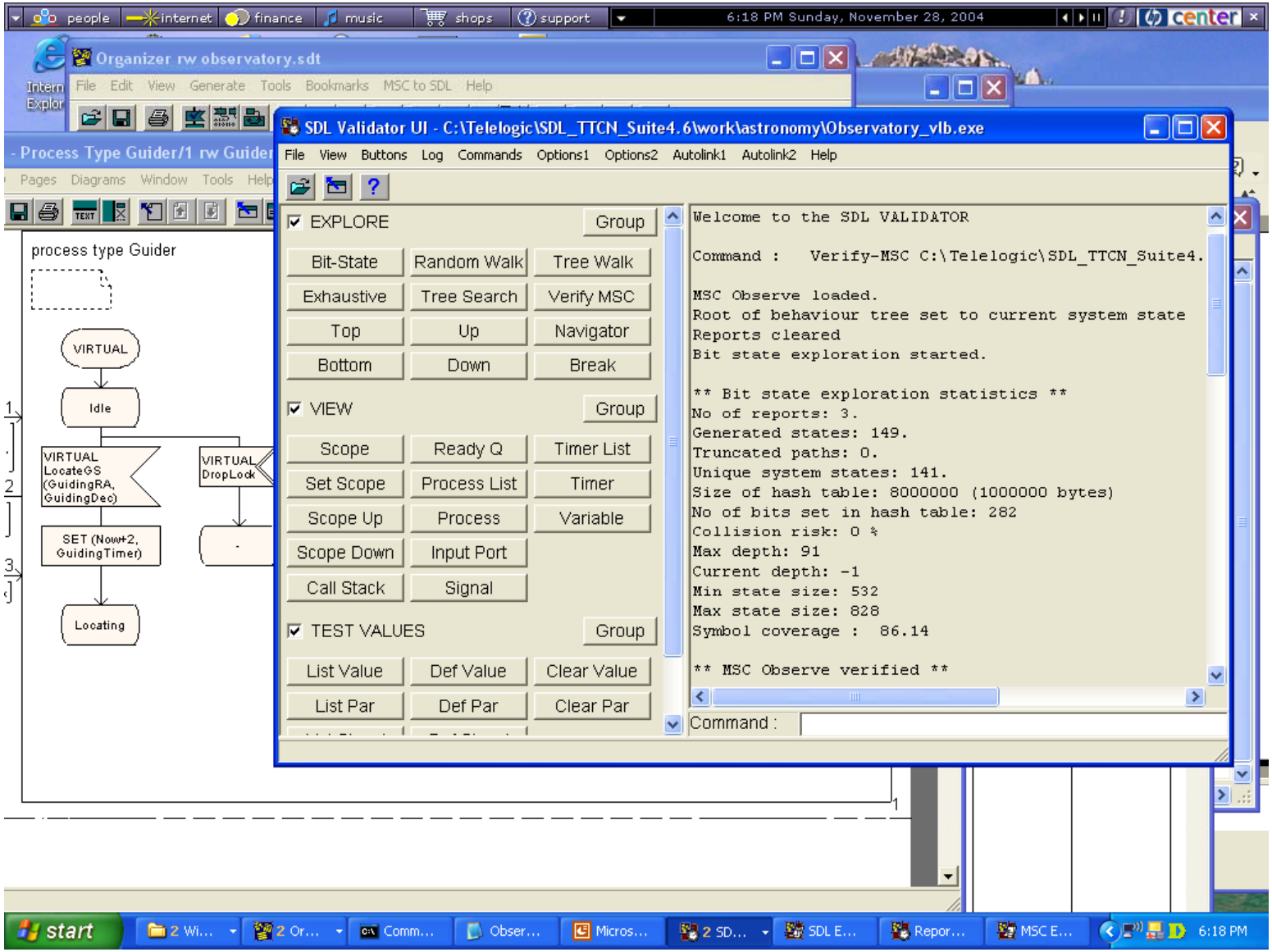

Figure 5-6: Model Validation against MSC Observe

5.3 Test Case Generation

Test case generation in Tau/SDL begins with creation of a TTCN test suite template from the model. The test suite is then filled in with test cases and associated declarations. Tau/SDL provides several different options for generating test cases: this can be done by capturing the results from a simulation, by converting an MSC use case, or as the result of random exploration. (Of course, it is also possible to edit the TTCN directly to create or modify test cases by hand.) Only the MSC method, a simple and automated approach, was attempted in this project.

This method is similar to the process of validating the model against a system-level MSC, discussed above. Once again the Validator application is opened and the system searches for a path through the model that satisfies the MSC. This time, however, instead of just generating a report showing how the MSC could be implemented, the tool generates a TTCN test case from the MSC and inserts it into the test suite. This will only work if the validation step is successful. After saving the test case to a TTCN file, the new test case appears in the TTCN view.

Figure 5-7 shows the TTCN structure that results from applying this method to the three MSCs from section 4.2. The test case section near the bottom of the screen shows that three test cases have been inserted, one for each use case. Also note that six points of control and observation (PCOs) have been defined, and that these have the same names as the external interface channels defined in the top-level Observatory SDL diagram.

Once such a test suite has been generated, it can be compiled into C, printed, or exported into HTML (as here). The full test suite is given in Appendix B. A detailed examination of the test cases (even without knowledge of TTCN) shows that the test suite is the result of integrating the MSCs into the SDL structure. The following parts of the suite have been populated:

- **PCO declarations**: as mentioned above, this declares the six channels which serve as points of control and observation to the Observatory in the SDL model.

- **Test Component Declarations**: this declares the signals that can be passed along those channels, along with the typed parameters for each signal.

- **Alias Definitions**: This declares an “alias” for each specific signal that is sent to the Observatory or expected from the Observatory through the PCOs according to the MSCs. The aliases give the signal name together with parameters. If a signal appears more than once (as is the case for TurnOnInstrument and InstrumentReady, which begin all three use cases), the alias is reused rather than defining a duplicate.

- **Test Cases**: The test cases are represented as sequence of events. Each event has a behavior description identifying the PCO, the signal passed through the PCO, the alias used to define the details of the signal, and whether the signal is an input to the system (“!”) or an expected output (“?”).

Test cases follow the structure of the MSCs. Thus, for example, the **Observe** test case begins with the InstrumentOn command, and ends with an expected DataAvailable signal with a value of 120. If the expected events all occur in the right order, the test case passes. Otherwise, it fails.

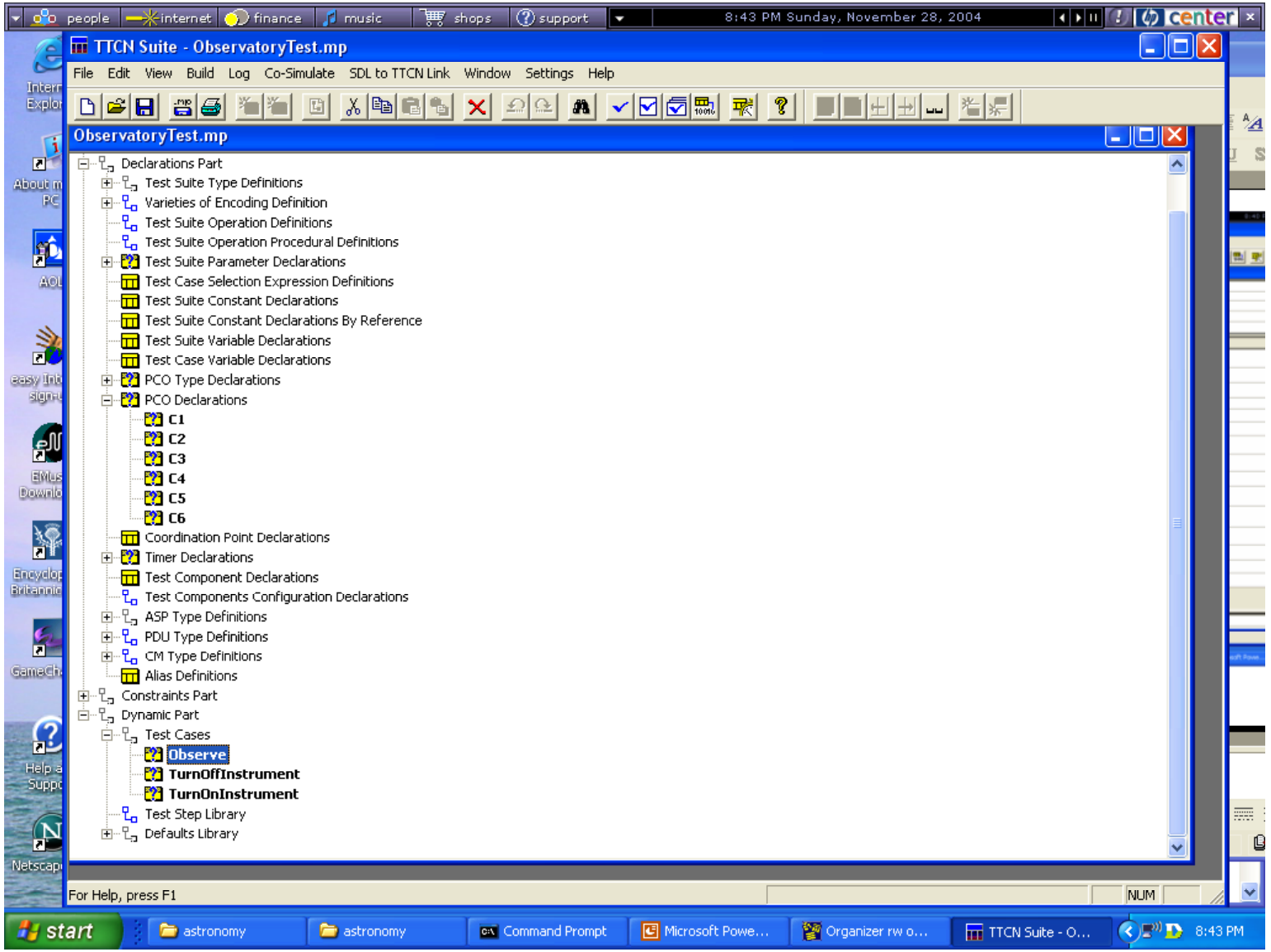

Figure 5-7: Observatory TTCN Test Suite Structure

6 Future Work

This project has demonstrated the use of a commercial tool to animate an SDL model with a complex architecture, validate the model against use cases, and generate test cases from the combination of the model and a set of use cases. Much work remains to be done to better understand the notations involved and the process arising from them. In order to further investigate the practical usefulness of these techniques, a number of issues remain to be addressed:

1. There are a host of technical issues regarding how this process would scale to a real systems enginering environment. In addition to addressing the bugs and language issues from section 4.6, there are a number of software engineering issues. First, how can SDL be used with abstract data types? In a real system architecture, data types tend to be complex and abstract, not defined at the level of programming language declarations as in this project. The Telelogic documentation refers to the Abstract Syntax Notation (ASN) standard for addressing this issue, but I didn't have time to investigate further. Second, the naming of channels between SDL agents. Wherever two agents communicate laterally or hierarchically, these names have to be visible to both so that one diagram can refer to another, but if the namespace is global I could easily see this becoming difficult to manage in a large system. Some guidance on naming conventions would be useful. Third, in a large model it would be desirable to have different engineers on a team work on different parts of the model at the same time. This raises issues of concurrent access and configuration management; it is not clear how Tau/SDL addresses them.

2. There are also questions about automating use of the tool itself. Recall that in the Observatory simulation, a breakpoint had to be set in order to send the necessary signals from the environment. In the simulator user interface, breakpoints have to be redefined each time a new simulation is started, so this got old very quickly. Ideally one should be able to define "simulation scripts" for repetition and regression testing of a simulation, rather than having to use the GUI each time. The documentation mentions a co-simulation facility whereby a TTCN test suite is used to drive a simulation; this is worth investigating further.

3. Most fundamentally, there are questions about the effectiveness of the automated V&V techniques used. At a minimum, to say a model has been "validated" each path should have been exercised at least once. In this example, unguided exploration was ineffective in covering the model, and while use case execution did better, it was not clear how to get to 100% coverage. Turning to verification, the technique of generating test cases from MSCs is good at covering the main success scenarios and establishing usability, but faults tend to occur on unexpected combinations of inputs, and MSC-based testing will miss these. One approach is simply to write additional MSCs for each contingency, but this will run into diminishing returns. Test heuristics (a technique used in the Burton thesis) can be very useful here, and it is not clear how these can be represented using Tau/SDL. For example, it was noted in section 4 that several processes were implemented in a way that could lead to deadlock on bright targets. It is not reasonable to expect a black-box testing technique to anticipate this type of fault, but a good test engineer would think of

stressing the system by looking at extreme cases, such as how the system handles targets that are either very faint or very bright. A test case for bright targets would have caught this particular fault, and also determined how well the system responded to a high input data rate. Further investigation is needed on how to automate generation of contingency and stressed scenarios.

Appendix A. Execution Trace for Observe Simulation

```
Command : output-via Expose 'CAM1' 180 30 180 30 5 'VIS_FILTER' -
Signal Expose was sent to InstrumentManager:1 from env:1
Process scope : InstrumentManager:1

Command : Go

*** TRANSITION START
*     PId    : InstrumentManager:1
*     State  : Ready
*     Input  : Expose
*     Sender : env:1
*     Now    : 100.0000
*     Parameter(s) : 'CAM1', 180, 30, 180, 30, 5.0000, 'VIS_FILTER'
*   OUTPUT of DropLock to Guider:1
*     The signal caused an immediate null transition
*   OUTPUT of Slew to AttitudeControlSystem:1
*     Parameter(s) : 180, 30
*** NEXTSTATE  Slewing

*** TRANSITION START
*     PId    : AttitudeControlSystem:1
*     State  : Stationary
*     Input  : Slew
*     Sender : InstrumentManager:1
*     Now    : 100.0000
*     Parameter(s) : 180, 30
*   SET on timer SlewTimer at 110.0000
*** NEXTSTATE  Slewing

*** TIMER signal was sent
*   Timer    : SlewTimer
*   Receiver : AttitudeControlSystem:1
*** Now      : 110.0000

*** TRANSITION START
*     PId    : AttitudeControlSystem:1
*     State  : Slewing
*     Input  : SlewTimer
*     Sender : AttitudeControlSystem:1
*     Now    : 110.0000
*   ASSIGN  RightAscension := 180
*   ASSIGN  Declination := 30
*   OUTPUT of SlewComplete to InstrumentManager:1
*** NEXTSTATE  Stationary

*** TRANSITION START
*     PId    : InstrumentManager:1
*     State  : Slewing
*     Input  : SlewComplete
*     Sender : AttitudeControlSystem:1
*     Now    : 110.0000
*   OUTPUT of LocateGS to Guider:1
*     Parameter(s) : 180, 30
```

*** NEXTSTATE GSLocate

*** TRANSITION START
* PId : Guider:1
* State : Idle
* Input : LocateGS
* Sender : InstrumentManager:1
* Now : 110.0000
* Parameter(s) : 180, 30
* SET on timer GuidingTimer at 112.0000
*** NEXTSTATE Locating

*** TIMER signal was sent
* Timer : GuidingTimer
* Receiver : Guider:1
*** Now : 112.0000

*** TRANSITION START
* PId : Guider:1
* State : Locating
* Input : GuidingTimer
* Sender : Guider:1
* Now : 112.0000
* OUTPUT of GSLocated to InstrumentManager:1
*** NEXTSTATE Ready

*** TRANSITION START
* PId : InstrumentManager:1
* State : GSLocate
* Input : GSLocated
* Sender : Guider:1
* Now : 112.0000
* OUTPUT of AcquireGS to Guider:1
*** NEXTSTATE GSAcquire

*** TRANSITION START
* PId : Guider:1
* State : Ready
* Input : AcquireGS
* Sender : InstrumentManager:1
* Now : 112.0000
*** NEXTSTATE Acquiring

Command : output-via GSLock -
Signal GSLock was sent to Guider:1 from env:1
Process scope : Guider:1

Command : Go

*** TRANSITION START
* PId : Guider:1
* State : Acquiring
* Input : GSLock
* Sender : env:1
* Now : 112.0000

```
*   OUTPUT of LockEstablished to InstrumentManager:1
*** NEXTSTATE  Guiding

*** TRANSITION START
*      PId    : InstrumentManager:1
*      State  : GSAcquire
*      Input  : LockEstablished
*      Sender : Guider:1
*      Now    : 112.0000
*   OUTPUT of DoExposure to CameraManager:1
*      Parameter(s) : 5.0000, 'VIS_FILTER'
*** NEXTSTATE  Exposing

*** TRANSITION START
*      PId    : CameraManager:1
*      In type: CameraManager
*      State  : Ready
*      Input  : DoExposure
*      Sender : InstrumentManager:1
*      Now    : 112.0000
*      Parameter(s) : 5.0000, 'VIS_FILTER'
*   OUTPUT of OpenShutter to Shutter:1
*** NEXTSTATE  OpeningShutter

*** TRANSITION START
*      PId    : Shutter:1
*      State  : Closed
*      Input  : OpenShutter
*      Sender : CameraManager:1
*      Now    : 112.0000
*   OUTPUT of ShutterOpen to CameraManager:1
*** NEXTSTATE  Open

*** TRANSITION START
*      PId    : CameraManager:1
*      In type: CameraManager
*      State  : OpeningShutter
*      Input  : ShutterOpen
*      Sender : Shutter:1
*      Now    : 112.0000
*   OUTPUT of PositionFilter to FilterAssembly:1
*      Parameter(s) : 'VIS_FILTER'
*** NEXTSTATE  MovingFilter

*** TRANSITION START
*      PId    : FilterAssembly:1
*      In type: FIlterSubsystem
*      State  : Positioned
*      Input  : PositionFilter
*      Sender : CameraManager:1
*      Now    : 112.0000
*      Parameter(s) : 'VIS_FILTER'
*   ASSIGN  CurrentPosition := 'VIS_FILTER'
*   OUTPUT of FilterInPlace to CameraManager:1
*** NEXTSTATE  Positioned
```

```
*** TRANSITION START
*      PId    : CameraManager:1
*      In type: CameraManager
*      State  : MovingFilter
*      Input  : FilterInPlace
*      Sender : FilterAssembly:1
*      Now    : 112.0000
*    OUTPUT of Integrate to Detector:1
*      Parameter(s) : 5.0000
*** NEXTSTATE  Exposing

Breakpoint matched by transition
PId    : Detector:1
State  : Idle
Input  : Integrate
Sender : CameraManager:1
Now    : 112.0000

Command : Proceed-Until 114.0

*** TRANSITION START
*      PId    : Detector:1
*      State  : Idle
*      Input  : Integrate
*      Sender : CameraManager:1
*      Now    : 112.0000
*      Parameter(s) : 5.0000
*    SET on timer DetectorTimer at 117.0000
*** NEXTSTATE  Integrating

Command : output-via PhotonArrival 10 -
Signal PhotonArrival was sent to OpticalAssembly:1 from env:1
Process scope : OpticalAssembly:1

Command : Go

*** TRANSITION START
*      PId    : OpticalAssembly:1
*      State  : Focus
*      Input  : PhotonArrival
*      Sender : env:1
*      Now    : 114.0000
*      Parameter(s) : 10
*    OUTPUT of PhotonArrival to InstrumentManager:1
*      Parameter(s) : 10
*** NEXTSTATE  Focus

*** TRANSITION START
*      PId    : InstrumentManager:1
*      State  : Exposing
*      Input  : PhotonArrival
*      Sender : OpticalAssembly:1
*      Now    : 114.0000
*      Parameter(s) : 10
*    OUTPUT of PhotonArrival to Shutter:1
*      Parameter(s) : 10
```

```
*** NEXTSTATE  Exposing

*** TRANSITION START
*      PId    : Shutter:1
*      State  : Open
*      Input  : PhotonArrival
*      Sender : InstrumentManager:1
*      Now    : 114.0000
*      Parameter(s) : 10
*   OUTPUT of PhotonArrival to FilterAssembly:1
*      Parameter(s) : 10
*** NEXTSTATE  Open

*** TRANSITION START
*      PId    : FilterAssembly:1
*      In type: FIlterSubsystem
*      State  : Positioned
*      Input  : PhotonArrival
*      Sender : Shutter:1
*      Now    : 114.0000
*      Parameter(s) : 10
*   OUTPUT of PhotonArrival to Detector:1
*      Parameter(s) : 10
*** NEXTSTATE  Positioned

*** TRANSITION START
*      PId    : Detector:1
*      State  : Integrating
*      Input  : PhotonArrival
*      Sender : FilterAssembly:1
*      Now    : 114.0000
*      Parameter(s) : 10
*   ASSIGN  ExposureCounts := 10
*** NEXTSTATE  Integrating

*** TIMER signal was sent
*   Timer    : DetectorTimer
*   Receiver : Detector:1
*** Now      : 117.0000

*** TRANSITION START
*      PId    : Detector:1
*      State  : Integrating
*      Input  : DetectorTimer
*      Sender : Detector:1
*      Now    : 117.0000
*   OUTPUT of ExpComplete to CameraManager:1
*** NEXTSTATE  DataAvailable

*** TRANSITION START
*      PId    : CameraManager:1
*      In type: HomingCameraManager
*      State  : Exposing
*      Input  : ExpComplete
*      Sender : Detector:1
*      Now    : 117.0000
```

```
*   OUTPUT of HomeFilter to FilterAssembly:1
*** NEXTSTATE  HomingFilter

*** TRANSITION START
*      PId    : FilterAssembly:1
*      In type: CAM1FilterSubsystem
*      State  : Positioned
*      Input  : HomeFilter
*      Sender : CameraManager:1
*      Now    : 117.0000
*   ASSIGN  CurrentPosition := 'CLEAR_FILTER'
*   OUTPUT of FilterInPlace to CameraManager:1
*** NEXTSTATE  Positioned

*** TRANSITION START
*      PId    : CameraManager:1
*      In type: HomingCameraManager
*      State  : HomingFilter
*      Input  : FilterInPlace
*      Sender : FilterAssembly:1
*      Now    : 117.0000
*   OUTPUT of CloseShutter to Shutter:1
*** NEXTSTATE  ClosingShutter

*** TRANSITION START
*      PId    : Shutter:1
*      State  : Open
*      Input  : CloseShutter
*      Sender : CameraManager:1
*      Now    : 117.0000
*   OUTPUT of ShutterClosed to CameraManager:1
*** NEXTSTATE  Closed

*** TRANSITION START
*      PId    : CameraManager:1
*      In type: CameraManager
*      State  : ClosingShutter
*      Input  : ShutterClosed
*      Sender : Shutter:1
*      Now    : 117.0000
*   OUTPUT of ReadOut to Detector:1
*** NEXTSTATE  ReadingOut

*** TRANSITION START
*      PId    : Detector:1
*      State  : DataAvailable
*      Input  : ReadOut
*      Sender : CameraManager:1
*      Now    : 117.0000
*   OUTPUT of InstrumentData to DataBuffer:1
*      Parameter(s) : 10
*   ASSIGN  ExposureCounts := 0
*   OUTPUT of ROComplete to CameraManager:1
*** NEXTSTATE  Idle

*** TRANSITION START
```

```
*      PId    : DataBuffer:1
*      State  : Empty
*      Input  : InstrumentData
*      Sender : Detector:1
*      Now    : 117.0000
*      Parameter(s) : 10
*   ASSIGN  BufferData := 10
*** NEXTSTATE  Readable

*** TRANSITION START
*      PId    : CameraManager:1
*      In type: CameraManager
*      State  : ReadingOut
*      Input  : ROComplete
*      Sender : Detector:1
*      Now    : 117.0000
*   OUTPUT of ExposureComplete to InstrumentManager:1
*      Parameter(s) : 'CAM1'
*** NEXTSTATE  Ready

*** TRANSITION START
*      PId    : InstrumentManager:1
*      State  : Exposing
*      Input  : ExposureComplete
*      Sender : CameraManager:1
*      Now    : 117.0000
*      Parameter(s) : 'CAM1'
*   OUTPUT of ExposureComplete to env:1
*      Parameter(s) : 'CAM1'
*** NEXTSTATE  Ready

Command : output-via DumpDataBuffer 'CAM1' -
Signal DumpDataBuffer was sent to InstrumentManager:1 from env:1
Process scope : InstrumentManager:1

Command : Go

*** TRANSITION START
*      PId    : InstrumentManager:1
*      State  : Ready
*      Input  : DumpDataBuffer
*      Sender : env:1
*      Now    : 117.0000
*      Parameter(s) : 'CAM1'
*   OUTPUT of DumpDataBuffer to CameraManager:1
*      Parameter(s) : 'CAM1'
*** NEXTSTATE  Dumping

*** TRANSITION START
*      PId    : CameraManager:1
*      In type: CameraManager
*      State  : Ready
*      Input  : DumpDataBuffer
*      Sender : InstrumentManager:1
*      Now    : 117.0000
*      Parameter(s) : 'CAM1'
```

```
*   OUTPUT of DumpDataBuffer to DataBuffer:1
*      Parameter(s) : 'CAM1'
*** NEXTSTATE  Dumping

*** TRANSITION START
*      PId    : DataBuffer:1
*      State  : Readable
*      Input  : DumpDataBuffer
*      Sender : CameraManager:1
*      Now    : 117.0000
*      Parameter(s) : 'CAM1'
*   OUTPUT of InstrumentData to InstrumentManager:1
*      Parameter(s) : 10
*   ASSIGN  BufferData := 0
*   OUTPUT of BufferDumpComplete to CameraManager:1
*      Parameter(s) : 'CAM1'
*** NEXTSTATE  Empty

*** TRANSITION START
*      PId    : InstrumentManager:1
*      State  : Dumping
*      Input  : InstrumentData
*      Sender : DataBuffer:1
*      Now    : 117.0000
*      Parameter(s) : 10
*   OUTPUT of InstrumentData to DataRecorder:1
*      Parameter(s) : 10
*** NEXTSTATE  Dumping

*** TRANSITION START
*      PId    : CameraManager:1
*      In type: CameraManager
*      State  : Dumping
*      Input  : BufferDumpComplete
*      Sender : DataBuffer:1
*      Now    : 117.0000
*      Parameter(s) : 'CAM1'
*   OUTPUT of BufferDumpComplete to InstrumentManager:1
*      Parameter(s) : 'CAM1'
*** NEXTSTATE  Ready

*** TRANSITION START
*      PId    : DataRecorder:1
*      State  : Empty
*      Input  : InstrumentData
*      Sender : InstrumentManager:1
*      Now    : 117.0000
*      Parameter(s) : 10
*   ASSIGN  RecorderData := 10
*** NEXTSTATE  Readable

*** TRANSITION START
*      PId    : InstrumentManager:1
*      State  : Dumping
*      Input  : BufferDumpComplete
*      Sender : CameraManager:1
```

```
*      Now    : 117.0000
*      Parameter(s) : 'CAM1'
*   OUTPUT of BufferDumpComplete to env:1
*      Parameter(s) : 'CAM1'
*** NEXTSTATE  Ready

Command : output-via DownloadData -
Signal DownloadData was sent to DataRecorder:1 from env:1
Process scope : DataRecorder:1

Command : Go

*** TRANSITION START
*      PId    : DataRecorder:1
*      State  : Readable
*      Input  : DownloadData
*      Sender : env:1
*      Now    : 117.0000
*   OUTPUT of DataAvailable to env:1
*      Parameter(s) : 10
*   ASSIGN  RecorderData := 0
*** NEXTSTATE  Empty

Command :
```

Appendix B. Observatory TTCN Test Suite

ObservatoryTest.mp

Test Suite Overview

- Test Suite Structure
- Test Case Index
- Test Step Index
- Default Index

- Declarations Part
  - Test Suite Type Definitions
    - Simple Type Definitions
    - Structured Type Definitions
    - ASN.1 Type Definitions
      - CharString
    - ASN.1 Type Definitions By Reference
  - Varieties of Encoding Definition
    - Encoding Definitions
    - Encoding Variations
    - Invalid Encoding Definitions
  - Test Suite Operation Definitions
  - Test Suite Operation Procedural Definitions
  - Test Suite Parameter Declarations
    - PIX_T_Global
  - Test Case Selection Expression Definitions
  - Test Suite Constant Declarations
  - Test Suite Constant Declarations By Reference
  - Test Suite Variable Declarations
  - Test Case Variable Declarations
  - PCO Type Declarations
    - PCO_Type
  - PCO Declarations

        - ASN.1 PDU Constraint Declarations
    - CM Constraint Declarations
        - TTCN CM Constraint Declarations
        - ASN.1 CM Constraint Declarations
- Dynamic Part
    - Test Cases
        - Observe
        - TurnOffInstrument
        - TurnOnInstrument
    - Test Step Library
    - Defaults Library
        - OtherwiseFail

Test Suite Structure

Suite Name	ObservatoryTest.mp		
Standards Ref			
PICS Ref			
PIXIT Ref			
Test Method(s)			
Comments			
Test Group Reference	**Selection Ref**	**Test Group Objective**	**Page Nr**
Detailed Comments			

Test Case Index

Test Group Reference	**Test Case Id**	**Selection Ref**	**Description**	**Page Nr**
	Observe			
	TurnOffInstrument			
	TurnOnInstrument			
Detailed Comments				

Test Step Index

Test Step Group Reference	**Test Step Id**	**Description**	**Page Nr**
Detailed Comments			

Default Index

Default Group Reference	Default Id	Description	Page Nr
	OtherwiseFail		
Detailed Comments			

Simple Type Definitions

Type Name	Type Definition	Type Encoding	Comments
Detailed Comments			

CharString

Type Name	CharString
Encoding Variation	
Comments	
Type Definition	
IA5String	
Detailed Comments	

ASN.1 Type Definitions By Reference

Type Name	Type Reference	Module Identifier	Encoding Variation	Comments	Type Definition
Detailed Comments					

Encoding Definitions

Encoding Rule Name	Reference	Default	Comments
Detailed Comments			

Test Suite Parameter Declarations

Parameter Name	Type	PICS/PIXIT Ref	Comments
PIX_T_Global	INTEGER		
Detailed Comments			

Test Case Selection Expression Definitions

Expression Name	Selection Expression	Comments
Detailed Comments		

Test Suite Constant Declarations

Constant Name	Type	Value	Comments
Detailed Comments			

Test Suite Constant Declarations By Reference

Constant Name	Type	Value Reference	Module Identifier	Comments	Value
Detailed Comments					

Test Suite Variable Declarations

Variable Name	Type	Value	Comments
Detailed Comments			

Test Case Variable Declarations

Variable Name	Type	Value	Comments
Detailed Comments			

PCO Type Declarations

PCO Type	Role	Comments
PCO_Type	LT	
Detailed Comments		

PCO Declarations

PCO Name	PCO Type	Role	Comments
C1	PCO_Type	LT	
C2	PCO_Type	LT	
C3	PCO_Type	LT	
C4	PCO_Type	LT	
C5	PCO_Type	LT	
C6	PCO_Type	LT	
Detailed Comments			

Coordination Point Declarations

CP Name	Comments
Detailed Comments	

Timer Declarations

Timer Name	Duration	Unit	Comments
T_Global	PIX_T_Global	s	
Detailed Comments			

Test Component Declarations

Component Name	Component Role	Nr PCOs	Nr CPs	Comments
Detailed Comments				

InstrumentOn

ASP Name	InstrumentOn
PCO Type	PCO_Type
Comments	

Type Definition

SEQUENCE {
  charString1 CharString
}

Detailed Comments	

InstrumentOff

ASP Name	InstrumentOff
PCO Type	PCO_Type
Comments	

Type Definition

SEQUENCE {
  charString1 CharString
}

Detailed Comments	

Expose

ASP Name	Expose
PCO Type	PCO_Type
Comments	

Type Definition

SEQUENCE {
  charString1 CharString,
  iNTEGER2 INTEGER,
  iNTEGER3 INTEGER,
  iNTEGER4 INTEGER,
  iNTEGER5 INTEGER,
  dURATION_NOT_IMPLEMENTED6 DURATION_NOT_IMPLEMENTED,
  charString7 CharString
}

Detailed Comments	

DumpDataBuffer

ASP Name	DumpDataBuffer
PCO Type	PCO_Type
Comments	

Type Definition
SEQUENCE {

charString1 CharString }
Detailed Comments

InstrumentReady

ASP Name	InstrumentReady
PCO Type	PCO_Type
Comments	
Type Definition	

SEQUENCE {
charString1 CharString
}	

Detailed Comments	

InstrumentPowerOff

ASP Name	InstrumentPowerOff
PCO Type	PCO_Type
Comments	
Type Definition	

SEQUENCE {
charString1 CharString
}	

Detailed Comments	

ExposureComplete

ASP Name	ExposureComplete
PCO Type	PCO_Type
Comments	
Type Definition	

SEQUENCE {
charString1 CharString
}	

Detailed Comments	

BufferDumpComplete

ASP Name	BufferDumpComplete
PCO Type	PCO_Type
Comments	
Type Definition	

SEQUENCE {
charString1 CharString
}	

Detailed Comments	

Detailed Comments	

GSLock

ASP Name	GSLock
PCO Type	PCO_Type
Comments	
Type Definition	
SEQUENCE { }	
Detailed Comments	

DownloadData

ASP Name	DownloadData
PCO Type	PCO_Type
Comments	
Type Definition	
SEQUENCE { }	
Detailed Comments	

PhotonArrival

ASP Name	PhotonArrival
PCO Type	PCO_Type
Comments	
Type Definition	

SEQUENCE {
  iNTEGER1 INTEGER
}	

Detailed Comments	

DataAvailable

ASP Name	DataAvailable
PCO Type	PCO_Type
Comments	
Type Definition	

SEQUENCE {
  iNTEGER1 INTEGER
}	

Detailed Comments	

ASN.1 ASP Type Definitions By Reference

ASP Name	PCO Type	Type Reference	Module Identifier	Comments	Type Definition
Detailed Comments					

ASN.1 PDU Type Definitions By Reference

PDU Name	PCO Type	Type Reference	Module Identifier	Enc Rule	Enc Variation	Comments	Type Definition
Detailed Comments							

Alias Definitions

Alias Name	Expansion	Comments
Detailed Comments		

cObserve_001

Constraint Name	cObserve_001
Group	
ASP Type	DataAvailable
Derivation Path	
Comments	
Constraint Value	
{ iNTEGER1 120 }	
Detailed Comments	

cObserve_002

Constraint Name	cObserve_002
Group	
ASP Type	DownloadData
Derivation Path	
Comments	
Constraint Value	
{ }	
Detailed Comments	

cObserve_003

Constraint Name	cObserve_003
Group	
ASP Type	BufferDumpComplete
Derivation Path	
Comments	
Constraint Value	
{ charString1 "CAM1" }	
Detailed Comments	

cObserve_004

Constraint Name	cObserve_004
Group	
ASP Type	DumpDataBuffer
Derivation Path	
Comments	
Constraint Value	
{ charString1 "CAM1" }	
Detailed Comments	

cObserve_005

Constraint Name	cObserve_005
Group	
ASP Type	ExposureComplete
Derivation Path	
Comments	
Constraint Value	
{ charString1 "CAM1" }	
Detailed Comments	

cObserve_006

Constraint Name	cObserve_006
Group	
ASP Type	PhotonArrival
Derivation Path	
Comments	
Constraint Value	
{ iNTEGER1 25 }	
Detailed Comments	

cObserve_007

Constraint Name	cObserve_007
Group	
ASP Type	PhotonArrival
Derivation Path	
Comments	
Constraint Value	
{ iNTEGER1 28 }	
Detailed Comments	

cObserve_008

Constraint Name	cObserve_008
Group	
ASP Type	PhotonArrival
Derivation Path	
Comments	
Constraint Value	
{ iNTEGER1 21 }	
Detailed Comments	

cObserve_009

Constraint Name	cObserve_009
Group	
ASP Type	PhotonArrival
Derivation Path	
Comments	
Constraint Value	
{ iNTEGER1 22 }	
Detailed Comments	

cObserve_010

Constraint Name	cObserve_010
Group	
ASP Type	PhotonArrival
Derivation Path	
Comments	
Constraint Value	
{ iNTEGER1 24 }	
Detailed Comments	

cObserve_011

Constraint Name	cObserve_011
Group	
ASP Type	GSLock
Derivation Path	
Comments	
Constraint Value	
{ }	
Detailed Comments	

cObserve_012

Constraint Name	cObserve_012
Group	
ASP Type	Expose
Derivation Path	
Comments	
Constraint Value	
{ charString1 "CAM1", iNTEGER2 180, iNTEGER3 30, iNTEGER4 180, iNTEGER5 30, dURATION_NOT_IMPLEMENTED6 5.0000, charString7 "VIS_FILTER" }	
Detailed Comments	

cObserve_013

Constraint Name	cObserve_013
Group	
ASP Type	InstrumentReady
Derivation Path	
Comments	
Constraint Value	
{ charString1 "CAM1" }	
Detailed Comments	

cObserve_014

Constraint Name	cObserve_014
Group	
ASP Type	InstrumentOn
Derivation Path	
Comments	
Constraint Value	
{ charString1 "CAM1" }	
Detailed Comments	

cTurnOffInstrument_001

Constraint Name	cTurnOffInstrument_001
Group	
ASP Type	InstrumentPowerOff
Derivation Path	
Comments	
Constraint Value	
{ charString1 "CAM1" }	
Detailed Comments	

cTurnOffInstrument_002

Constraint Name	cTurnOffInstrument_002
Group	
ASP Type	InstrumentOff
Derivation Path	
Comments	

Constraint Value
{ charString1 "CAM1" }

Detailed Comments	

Observe

Test Case Name	Observe
Group	
Purpose	
Configuration	
Default	OtherwiseFail
Comments	
Selection Ref	
Description	

Nr	Label	Behaviour Description	Constraints Ref	Verdict	Comments
1		START T_Global			
2		C1 ! InstrumentOn	cObserve_014		
3		C2 ? InstrumentReady	cObserve_013		
4		C1 ! Expose	cObserve_012		
5		C5 ! GSLock	cObserve_011		
6		C6 ! PhotonArrival	cObserve_010		
7		C6 ! PhotonArrival	cObserve_009		
8		C6 ! PhotonArrival	cObserve_008		
9		C6 ! PhotonArrival	cObserve_007		
10		C6 ! PhotonArrival	cObserve_006		
11		C2 ? ExposureComplete	cObserve_005		
12		C1 ! DumpDataBuffer	cObserve_004		
13		C2 ? BufferDumpComplete	cObserve_003		
14		C3 ! DownloadData	cObserve_002		
15		C4 ? DataAvailable CANCEL T_Global	cObserve_001	PASS	

Detailed Comments	

TurnOffInstrument

Test Case Name	TurnOffInstrument
Group	
Purpose	
Configuration	
Default	OtherwiseFail
Comments	
Selection Ref	
Description	

Nr	Label	Behaviour Description	Constraints Ref	Verdict	Comments
1		START T_Global			
2		C1 ! InstrumentOn	cObserve_014		
3		C2 ? InstrumentReady	cObserve_013		
4		C1 ! InstrumentOff	cTurnOffInstrument_002		
5		C2 ? InstrumentPowerOff CANCEL T_Global	cTurnOffInstrument_001	PASS	

Detailed Comments	

TurnOnInstrument

Test Case Name	TurnOnInstrument
Group	
Purpose	
Configuration	
Default	OtherwiseFail
Comments	
Selection Ref	
Description	

Nr	Label	Behaviour Description	Constraints Ref	Verdict	Comments
1		START T_Global			
2		C1 ! InstrumentOn	cObserve_014		
3		C2 ? InstrumentReady CANCEL T_Global	cObserve_013	PASS	

Detailed Comments	

OtherwiseFail

Default Name	OtherwiseFail
Group	
Objective	

Comments					
Description					
Nr	**Label**	**Behaviour Description**	**Constraints Ref**	**Verdict**	**Comments**
1		C1 ?OTHERWISE		FAIL	
2		C2 ?OTHERWISE		FAIL	
3		C5 ?OTHERWISE		FAIL	
4		C3 ?OTHERWISE		FAIL	
5		C6 ?OTHERWISE		FAIL	
6		C4 ?OTHERWISE		FAIL	
7		?TIMEOUT		FAIL	
Detailed Comments					

7 References and Web Resources

[1] Simon Burton, "Automated Generation of High Integrity Test Suites from Graphical Specifications", Ph.D. thesis, University of York, 2003.
[2] Jeffrey O'Grady, *System Validation and Verification*, 1998, p. 4.
[3] Burton, p. 15.
[4] http://en.wikipedia.org/wiki/Finite_state_machine
[5] Hans-Erik Eriksson, Magnus Penker, Brian Lyons and David Fado, *UML 2 Toolkit*, 2004.
[6] Burton, p. 14.
[7] http://www.utdallas.edu/~rmili/lab/vistamain.htm
[8] http://www.cs.waikato.ac.nz/~marku/formalmethods.html
[9] http://www-2.cs.cmu.edu/~nitpick/
[10] http://vl.zuser.org/#tools
[11] Burton, p. 15.
[12] Andreas Mitschele-Thiel, *Systems Engineering with SDL*, 2001, p. 3.
[13] Mitschele-Thiel, p. 144.
[14] Specification and Description Language (SDL) ITU-T Z.100, 11/2000.
[15] http://www.sdl-forum.org/SDL/index.htm
[16] Mitschele-Thiel, p. 162.
[17] Mitschele-Thiel, p. 143.
[18] http://www-306.ibm.com/software/awdtools/developer/rosexde/
[19] http://www.iec.org/online/tutorials/ttcn/index.html
[20] http://www.telelogic.com/products/tau/sdl/index.cfm
[21] Mitschele-Thiel, p. 142.
[22] http://www.solinet.com/safire.htm
[23] http://www.cinderella.dk/
[24] http://www.ucs.com.pt/ez/Products.htm
[25] Mitschele-Thiel, p. 157.