

Copyright 2008 Society of Photo-Optical Instrumentation Engineers.

This paper was (will be) published in Proceedings of SPIE Astronomical Telescopes and Instrumentation 2008 and is made available as an electronic reprint (preprint) with permission of SPIE. One print or electronic copy may be made for personal use only. Systematic or multiple reproduction, distribution to multiple locations via electronic or other means, duplication of any material in this paper for a fee or for commercial purposes, or modification of the content of the paper are prohibited.

Workstation Software Framework

L. Andolfato, R. Karban
ESO, Karl-Schwarzschild-Strasse 2, Garching, Germany

ABSTRACT

The Workstation Software Framework (WSF) is a state machine model driven development toolkit designed to generate event driven applications based on ESO VLT software. State machine models are used to generate executables. The toolkit provides versatile code generation options and it supports Mealy, Moore and hierarchical state machines. Generated code is readable and maintainable since it combines well known design patterns such as the State and the Template patterns. WSF promotes a development process that is based on model reusability through the creation of a catalog of state machine patterns.

Keywords: Model driven development, State machines, Catalog of state patterns, Design patterns

1. INTRODUCTION

Applications for the Very Large Telescopes (VLT) and the Very Large Telescope Interferometer (VLTI) are based on a common proprietary middle-ware platform (VLT Software [1]) which provides common services such as messaging, persistency, error handling and logging.

Since most control applications are reactive systems in which events trigger actions, an object oriented event-driven programming library (Event Handling Toolkit or EVH, [2]) was established to provide a common approach for handling commands, signals and timeouts. The library helped to develop collaborative applications, in particular in the domain of coordinating hierarchies of systems.

However, experience has demonstrated that developers found many different ways to build applications using the EVH library and the architecture of these applications appeared to be inconsistent. In particular the implementation of the state machine logic, needed to describe the behavior of the control software, was deeply mixed with the code implementing the actions. A change in the sequence of events could affect existing actions and a modification of an action could introduce side effects in the state machine logic. For such applications, requirement changes are difficult to implement and maintenance costs are very high.

This experience led to the idea of developing a tool able to facilitate the implementation of applications based on state machines sharing a common software architecture.

2. THE FRAMEWORK

In order to enforce a common software architecture, it was decided to develop a state machine framework, called Workstation Software Framework (WSF), based on solid design patterns where events, states, transitions and actions can be easily plugged in by the developer.

An application based on WSF can be built using one of the following methods:

- **Manually:** by extending the classes provided by the framework with the missing states, events, transitions, actions and data handling classes.
- **Using code generation from a text file:** by writing the description of the state machine in a text file (using WSF notation), generating the state machine (using WSF tools), and adding the code for the implementation of actions and data handling classes (Fig. 1).
- **Using code generation from UML models:** by modeling graphically the state machine using one of the supported UML modeling tools (Rational ROSE, Enterprise Architect or MagicDraw), generating the state machine (using WSF tools), and adding the code for the implementation of actions and data handling classes (Fig. 1).

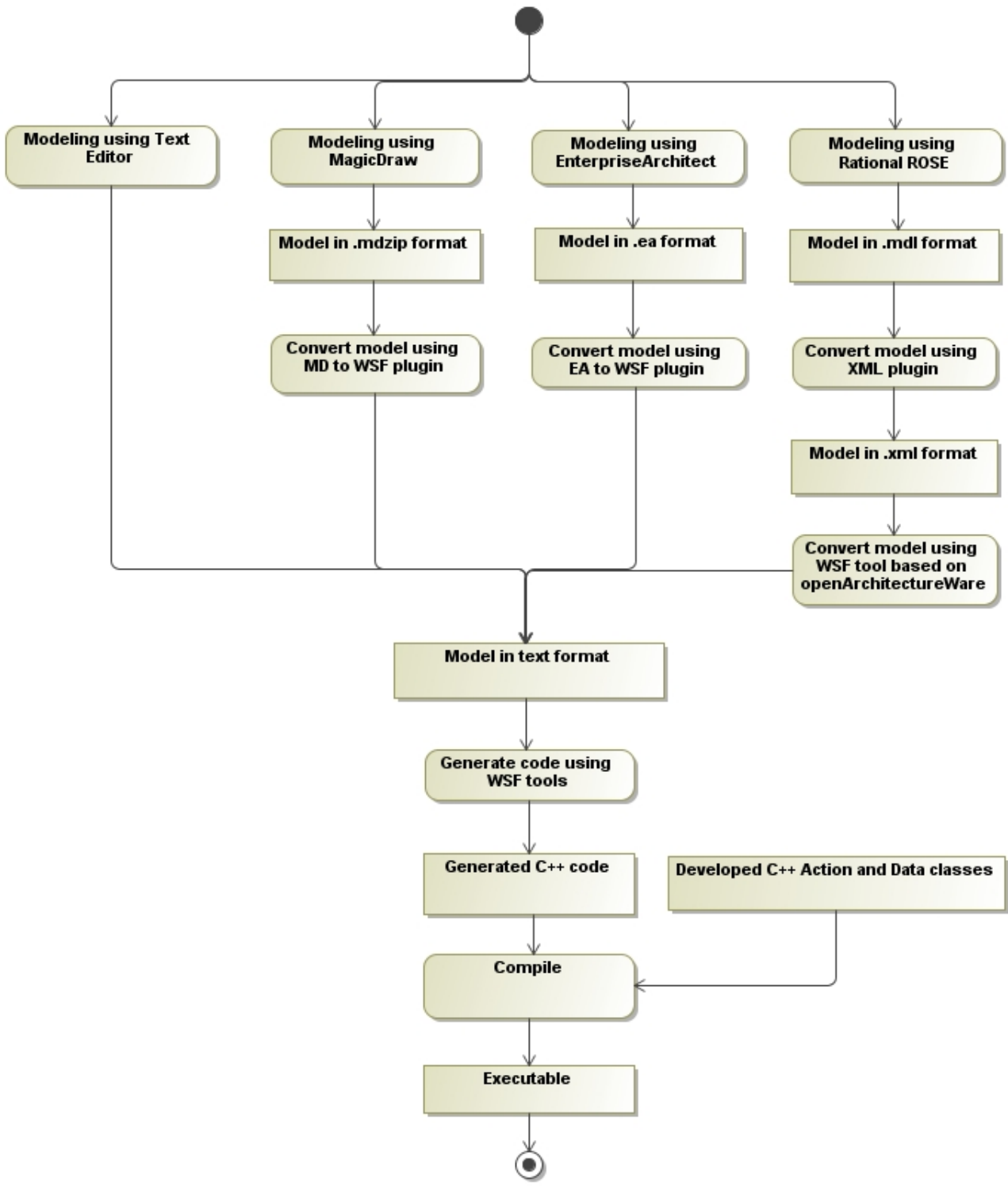


Figure 1 - Model transformations.

While the manual extension of the classes provided by the framework is useful only in the case when WSF tools are not available, the possibility of code generation from a text file frees the developer from tool dependency.

Applications generated by WSF share an architecture based on design patterns which guarantee scalability and maintainability.

2.1 Architecture

The architecture of WSF applications is based mainly on three design patterns [3]: the State pattern, a variation of the Command pattern and the Template pattern. These three design patterns are combined together avoiding circular dependencies.

The State pattern is used to implement the state machine logic. It allows an easy implementation of hierarchical state machines [4] and Mealy State machines [5]. It is also possible to achieve a partial implementation of Moore State machine [5] limited to the leave states. This approach was preferred to the classical event-state table implementation because the clean design simplifies maintenance. The event-state table implementation, where the state machine model is described by a matrix of Events x States and the entries are pointers to functions, has advantage of allowing run-time reconfiguration of the transitions but disadvantage of a complex and often unreadable implementation.

The Command pattern, together with the State pattern, allows separation of the action implementation from the state machine logic. In WSF an action is a method of the Action class (called Command class in the Command pattern). States can execute actions, during a transition or when entering/leaving the state, by invoking one of the standard methods of the class: "Init", "Execute", "Complete", "Abort", "Reject" or "Supersede". Thus the dependency between the action implementation and the state transition is simply a method invocation without parameters. Complex actions (macro commands) can be built upon several simpler action methods belonging to different action classes using a concept similar to the Composite pattern [2].

The Template pattern is used as mechanism to propagate events to the state machine. The supported events are: commands received by the application, replies to commands sent by the applications, expiration of timers, database notifications, I/O events (file I/O, socket I/O, etc.), UNIX signals and internal events generated by the application.

In addition to the design patterns mentioned above, two data classes, one used to access configuration information (usually stored in the configuration database or in configuration files) and the other to access run-time data, represent the interface between user defined data structures and action classes.

The presented architecture requires state machine diagrams to characterize the dynamic behavior of the applications and class diagrams to describe data structures that are needed by the application. These requirements define a specific development process for applications based on WSF.

2.2 Development process

The development process of a WSF application is based on iterations over the following steps:

- Identification/refinement of the state machine model and data classes
- Generation from the state machine model of the state and event classes using WSF tools
- Implementation of the action and data classes

The identification of the state machine model can be done using different techniques. We have found useful to analyze the requirements (Fig. 2) using UML sequence diagrams at application level where the applications to be developed are the objects of the sequence (Fig. 3).

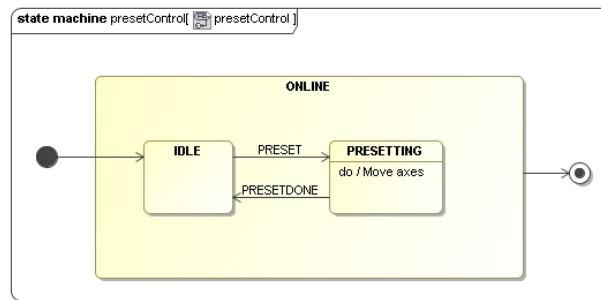


Figure 4: State machine model.

The design is completed with class diagrams that describe the data processed by the actions. Although not supported in the current version of WSF, the class diagram could also contain the relations between the actions and the data classes and between complex and simple actions. These relations could be used to increase the amount of generated code.

In the second step of the development process the developer can generate the application using the tools provided by WSF. The generated code represents an executable application with the complete state machine logic implementation but with empty action and data classes. At this point events could be generated to trigger state transitions and debug/test the state machine model.

In the last step of the process the developer finalizes the implementation of the application by writing the missing parts: the action and data classes.

Model and implementation of data and action classes can be refined in several iterations until the application requirements become stables. The code generation step can be repeated at any time since the generated code does not overwrite the code written by the developer.

Synchronization of the state machine model with the source code can be assured by forcing the code generation every time the application is compiled.

As for the development process, applications based on WSF can take advantages of the code generation feature to simplify the maintenance process.

2.3 Maintenance process

Changes to a WSF based application can be grouped into three categories:

- Changes that affect states and transitions only
- Changes that affect action and/or data implementation
- Changes that affect state machine model and action/data implementation

In the first case, the application can be updated simply by re-generating the code from a new model. In the second case the modification affects only the code written by the developer, therefore the developer has to fix the action/data classes. The last case is a combination of the first two and it involves the re-generation of the code and the reimplementation of action/data classes.

Perfective maintenance, e.g. due to changes in requirements, is made easier by the adoption of the design patterns which provides an expandable and scalable architecture.

2.4 Scalability

WSF can be used to build applications of different sizes because the overhead in execution time and memory allocation introduced by the framework scales reasonably with the size of the model. The size of a WSF application depends on the number of state, event, action and data classes. The number of state, event and action classes depends on the number of states (including composite states), events and actions defined in the model.

Each instance of a state class is a lightweight singleton object. States transitions are simply invocations of virtual methods and therefore the execution time depends on how polymorphism is implemented by the compiler and on how deep the hierarchy of the composite states is.

Event classes are just wrappers of EVH event classes. The time needed to propagate the event to the state machine logic is constant and it is basically the time needed to perform three methods invocations (from the EVH callback to the event object, from the event object to the state machine controller and from the state machine controller to the state object).

Actions and data interface classes are created at application's startup. Their size and performances depend on the implementation which is the responsibility of the developer. However, while the invocation of the methods of the data objects is coded directly by the developer, the invocation of the methods of actions objects is defined in the state machine model and it is carried out automatically by the framework. Since action objects are stored in a map container of the C++ standard template library, the time needed to access an action method is, according to [6], $O(\log n)$ where n is the number of action objects. A faster implementation could be easily obtained by replacing the map container with the hash_map container which should provide constant access time [6].

Consequently, the overhead introduced by the framework is very limited and it depends mainly on the software platform available.

2.5 Software platform and reusability

Applications generated by WSF run on top of the VLT Software platform, which is based on:

- Scientific Linux
- C++ GNU compiler, libraries and development tools
- VLT Software (including EVH)

Applications can be deployed on different software platforms by porting WSF tools, generating the code from the state machine model (which is platform independent) and re-implementing actions and data classes.

Within the same software platform, C++ implementation of actions and data classes can be reused in different applications thanks to the well defined interface between the state machine logic and the actions and data classes.

Since state machine models are platform independent, they can be easily reused at design level. Moreover, state machine models can be abstracted to be a generic solution to recurring design problem.

3. STATE PATTERNS

While modeling the state machine for different types of applications we realized that quite often similar design problems could be solved using the same state machine model. The idea is similar to the one introduced by the Gang of Four (Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides) with the traditional software design patterns: create a catalog of state machine models (state patterns) that can be reused to solve common problems when modeling the dynamic behavior of an application. It is interesting to note that the applications WSF generates are based on design patterns while the state machine models used by WSF to generate the applications are based on state patterns.

In the next paragraph, as an example of state patterns, the Sequence of Actions pattern is presented.

3.1 Sequence of Actions

Pattern Name: Sequence of Actions

Problem: In control systems there is often a need to configure several devices or components (HW or SW) following a specific order. In addition it might be required to be able to monitor what is the status of execution of the sequence.

The generic problem is how to be able to execute a sequence of actions and to be able to know, while executing the sequence of actions, what is the current status of the system.

Solution: The process in charge of executing the sequence of actions can be modeled using one state for each step of the sequence. The status of execution is given by returning the name of the current state of the system.

Structure: The basic structure of this state pattern is shown in Fig. 5.

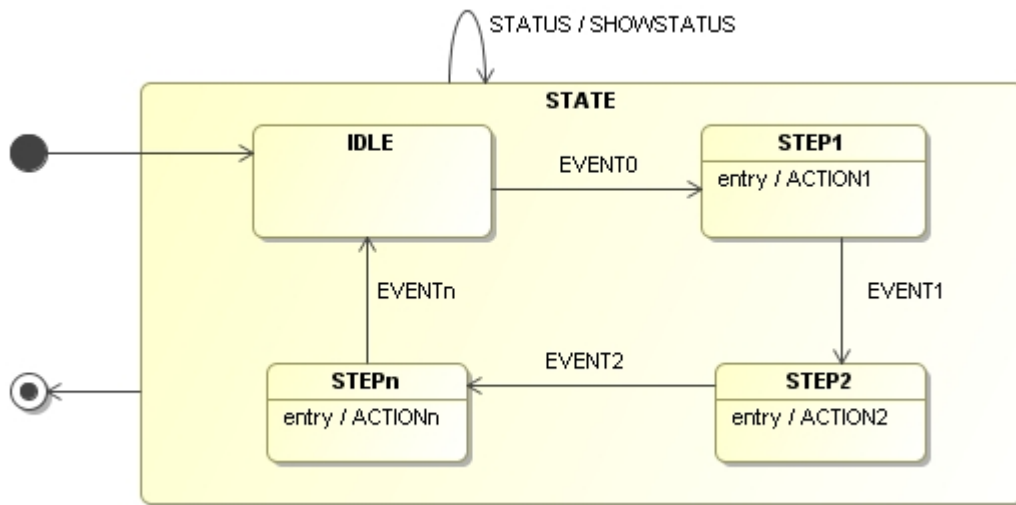


Figure 5 – Sequence of Actions

There are two variations of this basic state pattern: one which allows to skip the execution of some actions depending on a given configuration (Configurable Sequence of Actions, Fig. 6), and the other that allows to restart the sequence of actions from the last action which failed (Fault Tolerant Sequence of Actions, Fig. 7).

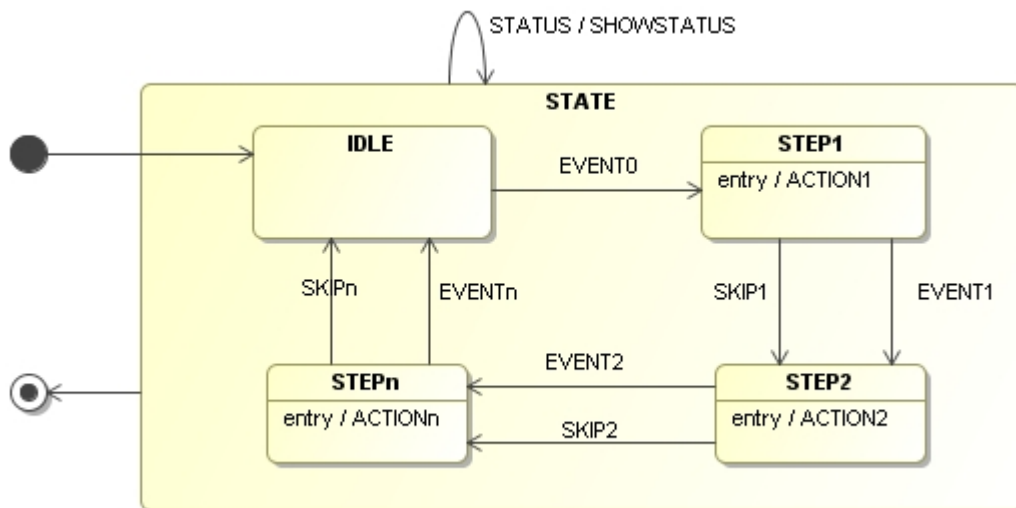


Figure 6 – Configurable Sequence of Actions

In the Configurable Sequence of Actions variation, the implementation of the actions (ACTION1, ACTION2, ... ACTIONn) includes a check of the configuration (read from file, memory, database, etc.) and the decision on whether to generate the SKIP internal event (to skip the step) or to continue with the normal execution of the action.

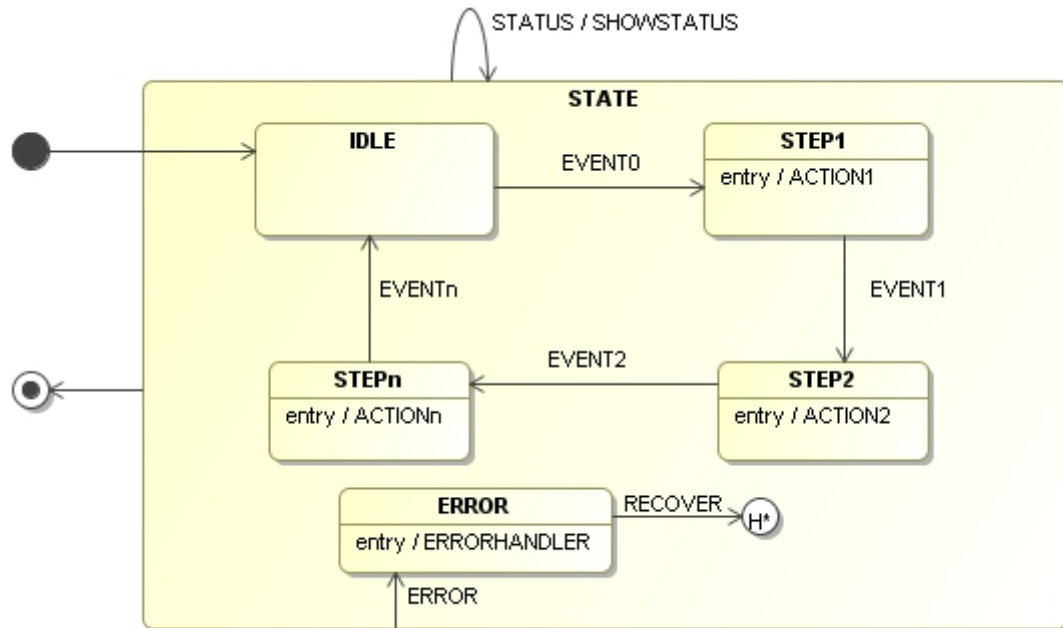


Figure 7 – Fault Tolerant Sequence of Actions

In the Fault Tolerant Sequence of Actions variation, the implementation of the actions (ACTION1, ACTION2, ... ACTIONn) starts with saving the current state in the history state. In case of error, the action generates an ERROR event which will move the system into ERROR state. When entering the ERROR state, the ERORHANDLER action will log the error and try to re-execute the step by generating the RECOVER event.

Examples: The Sequence of Actions state pattern can be used to model the preset sequence of a telescope. A telescope preset sequence is usually made of several actions that have to be executed sequentially. Some of the steps may have to be ignored if the type of observation does not require some telescope control software features or hardware components. The progress of the preset is usually monitored via a GUI and often, for performance reasons, if the sequence fails, it is required to restart from the last successful step of the sequence.

4. APPLICATIONS

The following projects have used WSF to develop workstation applications:

- Phase Referenced Imaging and Mirco-arcsecond Astrometry (PRIMA) facility for VLTI.
- Active Phasing Experiment (APE).
- New General detector Controller (NGC).
- Delay Line Control Software: maintenance and alignment application for Delay Lines (DL).

PRIMA [9] has been a test bed project for WSF. The size of the project (12 new processes spread on four workstations) and different types of applications required (control loops, configuration and coordination of real-time processes running on real-time platform, broadcasting of commands) have allowed tuning of the framework until a stable version could be released to other projects.

The effectiveness of state machine model reusability has been proved with the APE project [10] where similar applications were developed to control four different wave-front sensors. Out of 12 APE workstation applications, four are dedicated to control the calibration of the sensors and acquisition procedures and four to process the data produced by the sensors. The four data processing applications share exactly the same state machine model. Only some of the actions and data structures are different. The control applications share most of the state machine model and some of the actions in order to configure devices common to all sensors.

Finally the NGC project [8] gave the opportunity to compare two applications sharing the same functional requirements but developed with and without WSF. The New General detector Controller software is the successor of FIERA [7] offering additional functionalities. FIERA was developed before WSF was available and counts 51412 lines of code. The portion of NGC covering FIERA functionalities sums up to 51234 lines of code. The interesting fact is that while the total amount of code is the same, only 11% of the 51234 lines of NGC code has been written by the developer.

Table 1 shows that the percent of code written by developers (mainly actions and data classes) in different projects is on average below 30%.

Table 1 – Statistics on generated code.

Project	Number of developed applications based on WSF	Average number of states per application	Average number of transitions per application	Average number of lines of code per application	% Hand-crafted code per application
PRIMA	12	21	72	24004	21.24
APE	12	36	105	35020	25.08
NGC	4	17	34	16021	17.31
DL	1	26	68	24391	18.55

5. CONCLUSIONS

WSF is a framework which can help with the development and maintenance of event driven applications based on the ESO VLT software platform. It forces the developer to model the dynamic behavior of the application using state machine and it decreases the development time by generating a large portion of the code. Applications developed using WSF share the same architecture which helps in reducing maintenance costs.

Future developments of WSF include: refactoring of WSF by separating the Platform Independent Model (PIM) part from the Platform Specific Model (PSM) part to facilitate the porting of WSF to different software platforms e.g. the Atacama Large Millimetre Array Software Platform (Linux, GNU C++, Java, ALMA Common Software), publishing of a catalog of state patterns for control applications, and increase of the portion of generated code by adding the relations between actions, events and data classes in the model.

6. ACKNOWLEDGMENTS

We would like to thank Claudio Cumani for providing the data from FIERA and NGC projects, Andrea Balestra and Michele Zamparelli for the development of WSF plug-ins for EnterpriseArchitect and MagicDraw respectively, and David Terret for all the bugs he reported.

REFERENCES

- [1] Filippi, G., “VLT Common Software Overview”, ESO technical report <ftp://ftp.eso.org/pub/vlt/vlt/pub/releases/APR2003/vol-1a/VLT-MAN-ESO-17200-0888.pdf> (1995).
- [2] Chiozzi, G., Knudstrup, J., “VLT Software, CCS Event Toolkit - EVH”, <ftp://ftp.eso.org/pub/vlt/vlt/pub/releases/APR2003/vol-1b/VLT-MAN-ESO-17210-0771.pdf> (2001).
- [3] Gamma, E., Helm, R., Johnson, R., Vlissides, J., [Design Patterns], Addison-Wesley Longman (1994).
- [4] Harel, D., “Statecharts: A visual formalism for complex systems”, Science of Computer Programming, 231 (1987).
- [5] Hopcroft, J., Ullman, J., D., [Introduction to Automata Theory, Languages, and Computation], Addison-Wesley (1976).
- [6] Johnson, T., “C++ ST: Hash Containers and Performance”, Dr. Dobb’s Portal <http://www.ddj.com/cpp/198800559> (2007).
- [7] Cumani, C., Donaldson, R., “FIERA CCD Controller, Software Functional Specifications”, ESO technical report VLT-SPE-ESO-13640-1266 (1997).

- ^[8] Balestra, A., Cumani, C., Stegmeier, J., “NGC Software Requirements”, ESO technical report VLT-SPE-ESO-13660-3670 (2006).
- ^[9] Andolfato, L., Karban, R., “PRIMA Supervisor Software Requirements Specification”, ESO technical report VLT-SPE-ESO-15736-3059 (2003).
- ^[10] Karban, R., Surdej, I., “APE Control Software Requirements Specification”, ESO technical report ELT-SPE-ESO-04680-0001 (2006).