Run-time Model Extraction of a Java-based UAV Controller

Manjusha Choorakuzil[1], Adam Czerniejewski[1]
Jevitha K.P.[2], Swaminathan J.[2]
Lukasz Ziarek[1], Bharat Jayaraman[1]
1 - University at Buffalo
2 - Amrita University

Abstract. This case study discusses the structural and behavioral properties of jUAV, a Java-based adaptation of Paparazzi-UAV, the open-source system that consists of ground station software and autopilot systems for both autonomous and manual flying of unmanned aerial vehicles. Although the system has been popular and used widely, there hasnt been a high-level analysis of its structure and behavior. We carry out such an analysis by extracting finite-state automata from execution of the Java-based flight controller of the UAV. Our experiments were carried out using the open-source Eclipse plugin JIVE, which automatically extracts an execution trace from the execution of Java program. Using this execution trace and JIVEs state diagram feature, we were able to construct finite state machine (FSM) that clarify important properties of the controllers cyclic behavior. We employ byte-code instrumentation (BCI) to efficiently obtain long execution traces (on the order of a million execution events), and we present the automata for key use cases, such as circling around a point, and show how its correctness properties can be stated. The paper also presents experimental results to clarify that the overhead caused BCI does not adversely affect the real-time behavior of the periodic tasks that the UAV needs to perform.

1 JUAV: Java-based Paparazzi UAV

Model-checking [3] is an established method for the verification of safety-critical software and hardware systems, and several practical tools have evolved over the years to support this methodology. Often, design-time models are not available although there is a need to reason about software correctness. Even when they exist, there is often a big conceptual gap between design-time models and their implementation. Hence it is desirable to extract from a run-time execution of a system run-time models that are amenable to formal analysis, either to validate that the run-time models adhere to design-time models or to check that the run-time models have the desired properties. In this paper, we explore model extraction for the embedded autopilot of Paparazzi UAV [1, 2] (see Figure 1). Although Paparazzi UAV is a popular open source platform in use today, there hasn't been a high level analysis of its structural and behavioral properties. We feel that the embedded autopilot is an ideal candidate to show the usefulness

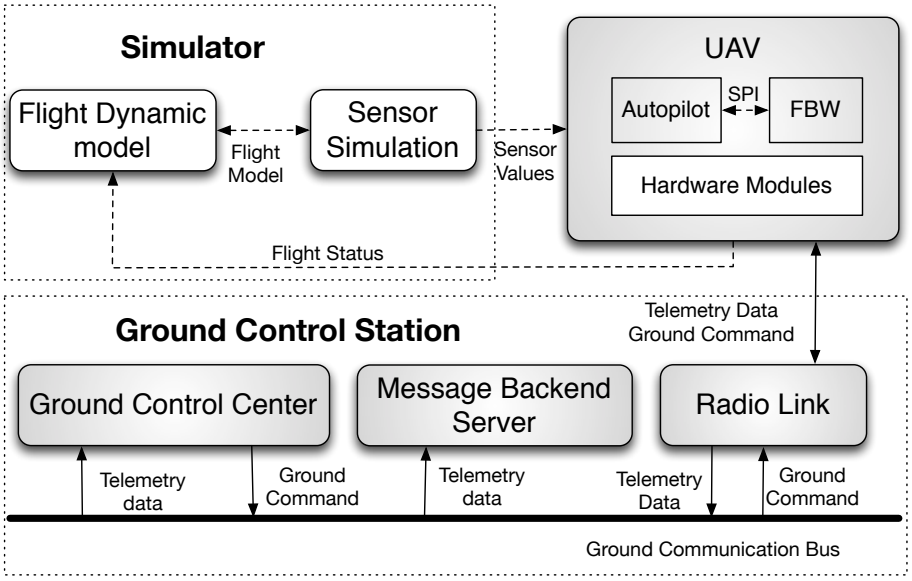


Fig. 1. Paparazzi UAV Architecture

of model extraction since it consists of many complex subsystems, including navigation, stabilization, and guidance. The system we analyze in this paper, called jUAV [4], is a direct port of the Paparazzi UAV's embedded autopilot to Java. depicts the architecture of Paparazzi UAV. In this paper we use run-time model extraction on the embedded autopilot of jUAV and reason about its run-time behavior. The embedded autopilots of both Paparazzi UAV and jUAV are amenable to high-level finite state machine modeling because their execution follows a repetitive and cyclic pattern.

Our run-time model is extracted from executing the jUAV program. Our experiments were carried out using the JIVE system [5] which automatically extracts an execution trace from a run of a Java program. Using this execution trace and a specification of key attributes of interest, JIVE synthesizes a finite state machine summarizing the changes to these attributes. The main contribution of this paper is in providing high-level finite-state models clarifying the behavior of the autopilot subsystem of the unmanned aerial vehicle. These models may also be used to verify properties of the UAV for typical navigational commands, such as flying to a point, hovering over a point, or circling a point. We bring out the cyclic and periodic nature of the tasks in the autopilot program and find correlations between executions of its various subtasks.

2 Model Extraction using JIVE

JIVE extracts a sequence of run-time events for a run of a Java program. The events include *new object instantiation, thread start/end, method call/exit, field*

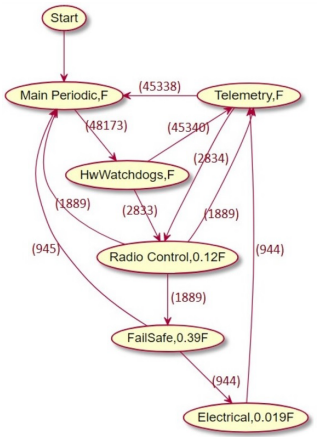
read/write, local variable read/write, line step, etc. From the standpoint of state diagram generation only *field write* events and *method call/exit* events are of main interest. There could be many useful models of a program, each with respect to different set of entities of interest to a programmer. Therefore, we let the user specify the key attributes of interest in various classes and/or objects. This information, together with the execution trace of the program, enables us to formulate a simple algorithm shown below for state diagram construction. (There are undoubtedly refinements of this algorithm, but we omit their discussion due to limited space.)

Input: Execution trace E[], Key attributes $\{x_1, \cdots, x_m\}$
Output: State graph G(V,E)
start = $\langle x_1 =?, \cdots, x_m =? \rangle$;
V $\leftarrow$ {start};
current_state $\leftarrow$ start;
foreach *Field Write event ev in E[]* **do**
    Let $ev = \{x = val\}$;
    if $x \notin \{x_1, \cdots, x_m\}$ **then**
        continue;
    else
        Let $x = x_j$ for some j $\in \{1, \cdots, m\}$;
        Let *val* be the value assigned to x in *ev*;
        next_state $\leftarrow$ current_state[$x \leftarrow val$];
        V $\leftarrow$ V $\cup$ next_state;
        E $\leftarrow$ E $\cup$ (current_state, next_state);
        current_state $\leftarrow$ next_state;
return G;

Fig. 2. Algorithm for State Diagram Construction

Using the above method, we have extracted a number of finite-state models to clarify different aspects of the operation of jUAV. We highlight two such models. The first is depicted in Figure 3 which shows that different tasks are executed at different frequency intervals. If the *Main Periodic* task is executed at a frequency 'F', the frequency of various periodic tasks such as *Telemetry, Radio Control,* and *Electrical* can be described as a suitable fraction of'F'. The state diagram in Figure 3 shows the actual repetition counts for every transition generated from executing the program, and the relative frequency values are obtained from these counts. This table shows that the basic operation of jUAV agrees with the published frequencies (in Hz) given in the last column of the table.

The second finite-state model is depicted in Figure 5 which corresponds to the UAV performing the use-case of going to a point and then circling around it, as depicted in Figure 4. Each state has two components: the symbolic name of the UAV's current direction, obtained by a mapping of its coordinates (GPS readings), and the distance from the center of the circle. Figure 5 shows the



FUNCTION NAME	RAW COUNT	RELATIVE FREQUENCY	FREQUENCY (in Hz)
Main Periodic	48173	F	512
Hardware Watchdogs	48173	F	512
Telemetry	48173	F	512
FailSafe	1889	0.39F	20
Electrical	944	0.019F	10
Radio Control	5667	0.12F	60

Fig. 3. Frequency Analysis of Periodic Tasks

result of mapping these coordinates. This model enables us to verify that the UAV was indeed circling aroud a point. For simplicity, we have illustrated just eight compass directions in the state machine. Assuming that each symbolic name such as *NorthEast, East, SouthEast*, etc., refers to a compass direction of the UAV, D is the distance between the point to the center of the circle, and k is the radius of the circle $\pm$T, where 'T' is the tolerance, it is straightforeward to formulate the property of circling around a point.

3 Performance Analysis

jUAV has a core set of periodic tasks (Figure 3) which need to run after a minimum time period has elapsed. In our experiments, these periodicities are treated as deadlines. For instance, the key periodic tasks of electrical, fail-safe, radio control, telemetry modules and main periodic must be complete every 100ms, 50 ms, 16.6 ms, 1.9 ms respectively. To evaluate jUAV's response, we compare the deadlines with and without byte code instrumentation.

We show a cumulative distribution function (CDF) in which vertical lines are drawn to indicate the deadlines of the periodic tasks: electrical, failsafe, radio control and telemetry modules. The CDG shows that jUAV meets all these deadlines with and without byte code instrumentation (for execution trace generation). A second observation is that jUAV without BCI meets the deadlines imposed by main periodic (deadline of 1.9 ms), but jUAV with BCI misses this deadline. This is not surprising since the instrumentation code periodically writes out all events that were logged during the execution of main periodic cycle. These periodic file-writes caused additional delays. To reduce the delays, we experimented with different filters which control the frequency of these file-writes. We noticed that the graph became smoother as the filter-size increases. The graph shown above is for a filter size of 50,000, which means a file write event

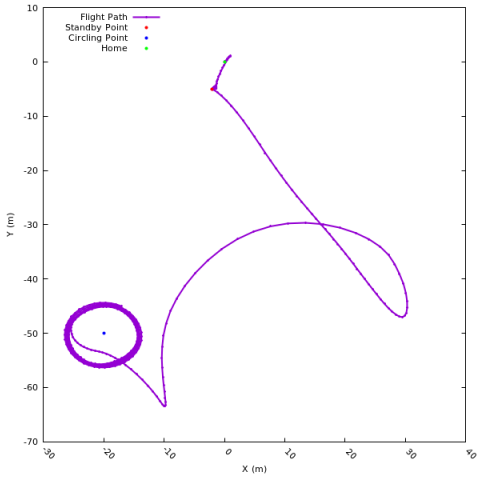


Fig. 4. Flight Path for jUAV

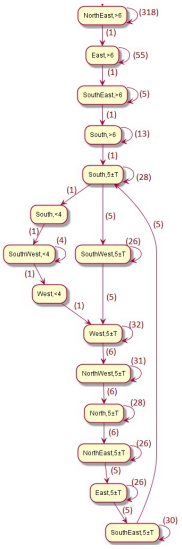


Fig. 5. Flight Directions

occurs after every 50,000 logged events. We plan to make the logging process more efficient to significantly reduce the variance between the two graphs.

4 Conclusions and Further Work

We have presented a case study of extracting high-level models clarifying the runtime behavior of jUAV, a Java-based adaptation of the well-known Paparazzi system for flying unmanned aerial vehicles. Despite the large code base for jUAV – about 10,000 lines of Java code – the core operation of the flight controller is cyclic in nature and hence finite state automata serve as good high-level models for clarifying the behavior of the system. These automata were extracted from from run-time execution traces of the jUAV system. We used the state diagram extraction feature of the JIVE system in conjunction with Byte Code Instrumentation for efficient generation of large execution traces. We able to show that the cyclic behavior of jUAV aligned with the design specification of the C-based Paparazzi UAV with respect to the frequency with which various important functions are performed. Additionally, we extracted automata for several key use cases and illustrated the case of circling around a point in the paper.

As part of our current and future work, we are investigating how to extract automata-based models of complex programs efficiently and with as minimal user intervention as possible, and also online generation of automata as the program executes. A key issue in analyzing run-time behavior is how representative is a single run of the system. In our case study, the execution trace for use-case of

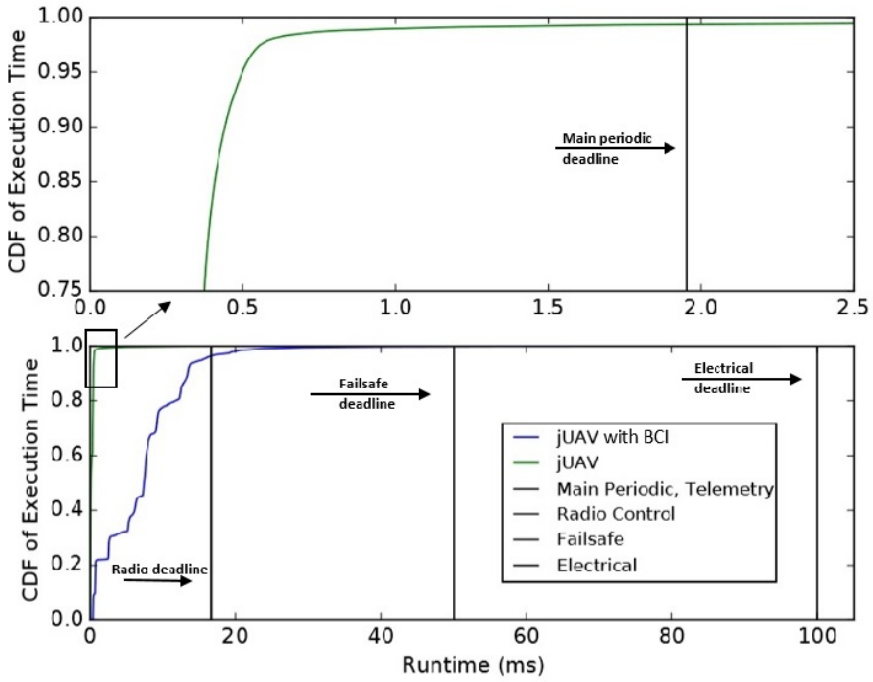


Fig. 6. Performance with and without BCI

circling around a point captured over 48,000 iterations of the system and visited numerous times all the major components needed for this use-case. Sometimes, this need not be the case and we might need to take union the automata from multiple runs to obtain more comprehensive mode for the use-case at hand.

References

1. Paparazzi project. `http://paparazzi.enac.fr/wiki/Main_Page`
2. Brisset, P., Drouin, A., Gorraz, M., Huard, P., Tyler, J.: The Paparazzi Solution: Rapport Technique (2006)
3. Clarke, E.M., Emerson, E.A., Sistla, A.P.: Automatic verification of finite-state concurrent systems using temporal logic specifications. ACM Trans. Program. Lang. Syst. 8(2), 244–263 (1986), `http://doi.acm.org/10.1145/5397.5399`
4. Czerniejewski, A., Dantu, K., Ziarek, L.: jUAV: A Real-Time Java UAV Autopilot. In: 2018 Second IEEE International Conference on Robotic Computing (IRC). pp. 258–261 (Jan 2018)
5. Ziarek, L., Jayaraman, B., Lessa, D., Jayaraman, S.: Runtime visualization and verification in JIVE. In: Proceedings of Runtime Verification - 16th International Conference, RV 2016, Madrid, Spain. pp. 493–497. Springer (2016)