A Lightweight Real-Time Executable Finite State Machine Model for Coordination in Robotic Systems

Markus Klotzbücher, Herman Bruyninckx
Department of Mechanical Engineering, Katholieke Universiteit Leuven
Celestijnenlaan 300B, 3001 Leuven (Heverlee), Belgium

Abstract—We describe our experience in developing a practical, lightweight, real-time executable model as an internal domain specific language using the Lua scripting language. We show that the problems of memory allocation and garbage collection can be substantially simplified compared to general purpose languages by exploiting knowledge of the modeled domain. We illustrate this for rFSM, a hierarchical finite state machine model used for coordination in the robotics domain.

Index Terms—real-time, coordination, garbage collection, domain-specific-language, component-based

I. INTRODUCTION

Building internal domain specific languages (DSL) is an efficient way to implement new programming languages by constructing these on top of an existing host language [1]. One advantage is avoiding to have to develop a new syntax and an associated parser. But more importantly internal DSL permit the construction of languages which can be directly executed as interpreted scripts in the host language without the need for additional transformation steps. This is a big advantage compared to more static model driven engineering approaches like MDA [2] and xUML [3] which typically require a model transformation and/or compilation before being executable.

rFSM (reduced Finite State Machine) is a minimal variant of UML Statecharts providing hierarchical, parallel and history states. States can specify entry and exit programs which are executed when entering and exiting the state respectively. The do program is executed while a state machine resides in a state and no events triggering outgoing transitions occur. An important concept of UML Statecharts is the run-to-completion step which implies that transitions between states are executed atomically and can not be interrupted. Transitions can therefore only be triggered after the entry program of a state has been executed. This condition is called a stable configuration.

Traditionally dealing with hard real-time constraints always implied an implementation in lower level languages as C, C++ or Ada. More recently several hard real-time Java implementations have emerged [4] providing deterministic temporal behavior. For the rFSM implementation we chose the Lua extension language [5] which is well suited for this task due to its single threaded nature, small memory footprint and precisely controllable incremental garbage collector.

rFSM is designed for modeling Coordination in component based robotic systems [6] [7]. However due to its portable and lightweight implementation it is not limited to robotics and can be easily applied to other domains too.

A. Coordination with Finite State Machines

It is an increasingly acknowledged best practice for building robust and reusable systems to modularize according to the aspects of Communication, Computation, Configuration and Coordination [8] [9] [10]. The aspect of Communication defines how system parts communicate with each other and the characteristics of the communication. Computations are the basic and reusable behaviors and define what is communicated. Configuration specifies which Computations form a system, their properties and interconnections. At last Coordination defines when Computations interact which Computations are part of an interaction and the protocols of interaction which are followed.

Finite State Machines (FSM) have been used for modeling complex and reactive systems for several decades [11] [12] [13] and are well suited for modeling Coordination for several reasons. Firstly the FSM formalism is well understood and expressive while still being simple to understand. This is an important requirement in robotics where many users are not software engineers. Secondly FSM are supported by a wide range of modeling tools and can be formally verified. At last the hierarchical types of FSM such as Harel Statecharts and the therefrom derived UML Statecharts have good compositional properties which allow for incrementally developing large Coordination models by composition of simpler ones.

Fig. 1. Coordination of Component Assemblies

An example of a simple robotics coordination problem is
illustrated by the component assembly shown in figure 1. A ball swinging on a string is to be followed by a robot arm. The 2D ball positions extracted from two camera images by BallExtractor components are passed to the estimation component. The estimated value is then sent to the RobotController which actuates the robot arm. Now the situation is possible that the ball swings out of the observed camera range. In this case the desired behavior is that the robot arm stops at the last estimated ball position. However different estimation models will show different behaviors; for instance a constant velocity model will predict the ball motion to continue with the last estimated velocity while a constant position model will continue to predict the last observed position.

A naive solution to this problem would be to add a feature to the estimator to stop the robot controller once the ball has left the camera range. This solution however will severely limit the reusability of this component and make it impossible to replace it with a different estimation component which does not make these application dependent assumptions. A better solution is to introduce a Coordination component which encapsulates this policy. The state machine for such a component is shown in figure 2.

![Ball-tracking Coordination state machine](image)

Instead of making assumptions about the component layout the estimation component raises an event when the ball is not being tracked anymore. The coordination component then reacts to this event and transitions to the pause state in which the robot is stopped. When the ball enters the camera range and the estimator begins tracking the ball again, a second event is raised to transition back to the following state and restarting the robot controller in the exit program of the pause state.

B. Contributions

The contribution of this work is describing an approach for creating a lightweight, real-time executable FSM model by showing how domain specific knowledge can be exploited for efficiently dealing with memory allocation and collection.

Our approach has several advantages. The use of a garbage collected language for executing the model eliminates the large class of pointer related errors and memory leaks. This is especially relevant for Coordination which has higher requirements of robustness and reliability than regular Computations. Secondly our approach unifies simulation and final system, thus avoiding the problem of the two behaving inconsistently. At last the dynamic nature of scripting languages readily supports dynamic changes like hot swapping of code or even modifications to the structure of the state machine, thereby supporting the principles of evolving systems [9].

C. Outline

The rest of this paper is structured as follows. Section II describes related work. In section III we describe how we deal with the problem of real-time memory management for the rFSM DSL. Section IV describes advanced extensions which are enabled by our approach and section V gives some experimental results. We conclude and summarize in chapter VI.

II. RELATED WORK

The UML language has long been used for modeling real-time systems [14] [15], however mostly as a specification and modeling language without the intent to execute the model. More recently the UML MARTE profile [16] (Modeling and Analysis of Real-time and Embedded systems) extends the UML modeling language with features specific to real-time and embedded systems. [17] shows how executable specifications can be generated from these models, yet additional transformations are necessary.

With respect to the scheduling of garbage collection the work on time triggered garbage collection [18] is relevant as it describes an alternative approach to the classic allocation driven collection scheme. In contrast to our work the general purpose language of Java is targeted while our state driven approach is specific to the target domain of Coordination.

III. MEMORY MANAGEMENT

The main challenge for using scripting languages in a real-time context is dealing with the potentially unbounded timing behavior of allocation and recuperation of memory. While it is possible to create simple scripting languages which pre-allocate all memory [6] as a result of the static allocation these languages are restricted in terms of expressivity. With languages providing higher levels of expressivity which are necessary for building internal DSL one has to face both real-time allocation and garbage collection.

In this work we do not propose new techniques for addressing these issues but rather show how making use of domain knowledge combined with existing techniques can allow for substantially reducing the complexity of allocation and garbage collection issues.

A. Real-time memory allocation

Traditional real-time systems preallocate all required memory at initialization time, as allocations and deallocations at run-time may result in unbounded temporal behavior. For allocation in scripting languages this is not practical as the ability to dynamically create and dispose of objects makes the use of a scripting language worthwhile in the
first place. Fortunately results in real-time allocation such as [19] and more recently [20] allow memory allocations with guarantees for worst case execution time. For the rFSM implementation we use the TLSF memory allocator [20] with the Lua language.

Nevertheless it is important to recognize that using a real-time allocator is still a form of pre-allocation, as the allocations are served from a pre-allocated memory pool. Increasing the pool size might require falling back on a non-real-time safe system memory allocator or under low memory conditions might not be possible at all. Consequently the amount of memory necessary has to be known in advance which in practice is often not entirely possible. For the rFSM implementation the memory consumption depends on:

1) the static size of the state machine graph (states, transitions, user code)
2) the dynamic memory consumption of state machine core and the user supplied entry, do and exit programs.

The first can be effectively estimated as it increases linearly with the amount of states and transitions as shown in figure 3

![Fig. 3. rFSM static memory usage](image)

The graph shows the static memory consumption (in KiB) for rFSM instances of n states and 2*n states and transitions. The second however is less straightforward, as predicting the memory usage of user supplied programs is not easily possible. Fortunately for the domain of Coordination we can assume the following: i) the user programs consist mainly of calls to low-level C/C++ code for Coordinating the actual computations and ii) no large amounts of data are transferred between the scripting language and low-level C code. This way the amount of dynamic memory required is very small compared to the overall usage and can be neglected. Instead depending on how critical the implemented functionality is a safety margin of additional memory is allocated. We realize that such allocations of safety margins might not be feasible on very small embedded systems. However for modern robotics systems which are typically well equipped with memory this is in general not a problem.

### B. Allocation Failure Strategies

Besides taking proactive measures against running out of memory by predicting memory usage and allocating safety margins, a robust FSM implementation must nevertheless be able to deal gracefully with low memory conditions. For the rFSM implementation this is achieved by pre-allocating a second block of memory called the “emergency pool”. If the memory of the regular pool is running low, an internal event is raised by the state machine core which can be used to transition to a safe state. This is illustrated in figure 4.

![Fig. 4. Transition to safe state triggered by low memory event](image)

During normal operation the FSM resides in the operational state or one of its substates (omitted for readability). If the FSM core runs short on regular memory, the event e_low_mem is raised. This event then can be used to transition to a state safe_mode in which the system can deal with the situation by making adjustments or performing manual garbage collection after stopping the robot in a safe manner. Note how the low-memory condition can be dealt with within the state machine model without the need for external handlers etc. The emergency pool serves to guarantee that enough memory is available for performing the emergency transition in spite of the regular memory being exhausted.

### C. Real-time garbage collection

Real-time garbage collection has been intensively researched over the last decades [21] [22] [23] with a lot of attention directed towards concurrent collection in multi-threaded systems. For the Coordination domain we make the following simplifying assumptions. Firstly Coordination does not require true parallel execution. This is because Coordination actions commonly consist of short asynchronous commands to lower level Computations as for instance switching controllers, sending a new setpoint to a trajectory generator or commanding a gripper to close. In fact it is desirable for achieving deterministic behavior to atomically execute these programs. Hence even the user programs of parallel states are executed in a serialized manner by means of coroutines for cooperative multitasking. This way the complexity and nondeterminism of parallel garbage collection can be avoided.
The second assumption can be made about when garbage collection shall be performed. For an UML like FSM this is naturally after the run-to-completion step has finished. More precisely this is after a transition has taken place and the state machine has reached a stable configuration but before the in-state do program is executed.

One prerequisite for our implementation is an incremental garbage collector which can be disabled and manually invoked by the host language. Automatic garbage collection is then disabled and incremental collections are executed by the FSM implementation after each run-to-completion step. As the overall memory consumption is monitored by the allocator itself and low memory events are raised for dealing with this condition no extra validation is necessary to ensure that the incremental collector is keeping up with the allocations. Due to the deterministic and single threaded nature transition coverage testing can reliably reveal such conditions.

IV. DYNAMIC EXTENSIONS

The use of a scripting language enables the following state machines extensions.

1) Live code upgrades: For long running, evolving systems it is desirable to be able to make changes to the live system at runtime. As rFSM is implemented as a Lua based DSL it inherits the properties of its host language, in which loading new modules, replacing existing code and cleaning up of unreferenced functions is supported.

2) Dynamic FSM: It is a common practice today to allow hot-plugging of hardware to running systems by dynamically creating the required components. What is less straightforward is including the required coordination for these new components. This is because coordination activities can often not be viewed in isolation but rather must be integrated into the existing coordination, ideally without disrupting ongoing actions. rFSM can accommodate such changes by allowing runtime structural modifications like

- adding a nested state to the existing state machine
- replacing a nested state
- adding a new transitions between states

Interestingly performing such changes constitutes a Coordination problem itself as for instance it is not permitted to remove a state while its programs are being executed. Currently these extensions have not been sufficiently explored yet and are subject to further work.

V. EXPERIMENTAL EVALUATION

This section gives some experimental results for evaluation the temporal behavior of different state machines executed by rFSM. All tests were run using the Xenomai [24] real-time framework on a Pentium M 1600 MHz.

For evaluating the worst case timing behavior of the FSM implementation there are two important measures inherent to the rFSM core and to the specific FSM instance respectively:

1) The worst case time required for transitioning the state machine from the reception of an event to reaching a stable configuration.

2) The worst case duration of an incremental garbage collection step which is executed after the entry program of a state.

The first is important for determining the overall worst case duration of a transition which can be calculated by summing up this value with the execution time of the exit programs, transition effect and entry programs.

The second is important as it defines the minimal in-state delay for exiting a newly entered state again. Together with the transition time this value can be used for analyzing the timing behavior of a path of transitions of a FSM.

All tests were performed by executing transitions forming a circular chain of states of different hierarchical depths and using a different number of states. Figure 5 shows three exemplary states of a depth of three. The default connector of each nested state connects to its substate which causes each transition to start from the most nested state and end on the most nested state of the successor. No user programs are supplied except for calls to Lua POSIX real-time bindings used to measure the timing behavior. For each benchmark 10000 transition steps were executed.

Figure 6 shows the duration required for the rFSM core to transition between two states. Two things can be observed. Firstly, the worst case transition time for deeper nested hierarchical states is higher as the rFSM core must exit and enter more states until the next stable configuration is reached. Secondly, the total amount of states in the state machine does not influence the transition duration. The glitches seen in the test of a hierarchical depth of 10 are reproducible in subsequent test runs and can most likely be attributed to the exceeding of a CPU cache.

In contrast to transition times the worst case duration of garbage collection does depend on the overall size of the state machine as can be seen in figure 7. This is due to the Lua garbage collector which although incremental, performs collections of the Lua table type atomically. As the table sizes
VI. CONCLUSIONS

We have presented a executable state machine model for the domain of coordination in complex robotic systems implemented as a Lua internal domain specific language. Besides freeing the user from the burden of manual memory management our implementation facilitates dynamic FSM extensions such as code hot-swapping or even dynamic structural modifications to running FSM. We have shown how the common problem of dealing with real-time memory allocation and garbage collection can be simplified by making use of characteristics of the target domain which for the Coordination domain is the feasibility to run single threaded and the tendency to produce little garbage. The evaluation shows, in our opinion, reasonable overhead and latencies which are quite acceptable for the robustness and flexibility gained.

Much further optimization is possible. Following the best practice of avoiding premature optimization the current implementation takes the simplest but not necessary most efficient approach. Thus profiling the implementation is likely to reveal many opportunities for improvement.

The weakest point in our implementation is currently the unpredictable memory consumption of user programs. In practice this has been solved by prior profiling the worst case memory usage in a test run covering all transitions. Due to the single threaded, deterministic implementation the result can then be used for pre-allocating the required amount of memory.

The current approach of triggering garbage collection after a run-to-completion step works well and yields deterministic behavior. However real-world state machines are rarely as homogeneous as those used in the evaluation. Therefore it might be beneficial for reducing jitter to dynamically adapt the collection points in the state machine graph according to the duration of the respective garbage collection. Future work will investigate this.

REFERENCES


