OROCOS RTT-Lua: an Execution Environment for building Real-time Robotic Domain Specific Languages

Markus Klotzbücher, Peter Soetens and Herman Bruyninckx

Abstract—From the beginning the OROCOS Real-Time Toolkit has offered support for dynamically loadable, hard real-time safe scripting and state machines. Users often consider these mechanism as key features because they provide the ability to perform dynamic and even interactive changes to the system by means of a language which is much simpler and more robust than the C++ language of the host system. In spite of this some shortcomings of the current implementation have become evident: Firstly there is a need for a more expressive state machine model which adheres to well-known semantics. Secondly the cost to maintain and evolve an own scripting and state machine language is high. At last portability of scripts and state machines is limited as the scripting implementation is inherently tied to the RTT framework. This paper describes our approach to overcome these issues by creating a Lua based execution environment for constructing real-time safe robotic domain specific languages. We illustrates its application by describing a hierarchical Statechart model used for coordination in the robotics domain.

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

I. INTRODUCTION

Scripting languages are commonly lightweight dynamic languages which are embedded in larger systems in order to extend these. These systems are often optimized for speed or low latency and thus developed in compiled languages such as C or C++. The extension language in contrast aims at offering a dynamic interface to the functionality offered by the base system. Extension languages have been in use for decades as for example Emacs Lisp [1], TCL [2] or variants of Scheme [3].

The OROCOS Real-Time Toolkit (RTT) is a C++ framework for constructing complex component based robotic systems [4]. From the beginning a scripting language and a state machine implementation were provided which allow to implement the behavior of a component without the need to recompile. More importantly these languages are hard real-time safe which means execution takes place in a temporally deterministic fashion. This is achieved by pre-allocating all necessary data such as variables and functions at load-time. Hence during execution the potentially non-deterministic operations of allocating and recuperating of memory can be avoided.

The RTT state machine language allows construction of simple non-hierarchical state machines which can be loaded into components at runtime. This is a major advantage compared to more static model driven engineering approaches like MDA [5] and xUML [6] which typically require a model transformation and/or compilation step before being executable.

With the growing complexity of robotic systems the need for a more expressive hierarchical state machine model has become clear. The obvious solution would be to extend the existing implementation of the RTT. This however would not solve the problem of portability of state machines and scripts. Additionally this approach would further increase the complexity and consequently cost of maintenance of the current implementation.

An alternative approach is to implement the required hierarchical state machines as an internal domain specific language (DSL). This technique is an efficient way to implement new programming languages by constructing these on top of an existing host language [7]. This has the advantage of avoiding the development of 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.

The current RTT scripting language does not easily permit construction of internal DSL. Reasons for this are the static typing model, the absence of a Lisp like eval function and the lack of a generic compound type. These language constructs are missing because of the complexity involved in implementing these in combination with the approach of rigorous pre-allocation, which in turn is needed for hard real-time safe execution.

In order to overcome these issues the Lua [8] extension language was chosen as a basis for a new RTT scripting language as it offers the following:

a) custom memory allocation: Lua provides a mechanism to use a custom memory allocator instead of the standard OS functions. As described later this is crucial for use in a hard real-time context.

b) real-time garbage collector: Lua is widely used in soft real-time environments such as game engines and to this end provides a precisely controllable incremental garbage collector. For instance it is possible to tune how aggressively memory is collected or to stop and restart the collector as desired.

c) suitable for building internal DSL: Lua is suitable for building internal domain specific languages as it is...
dynamically typed, has a Lisp like eval mechanism in form of the Lua loadstring function and provides the Lua table as a generic compound type.

As an example of internal DSL, based on RTT-Lua we describe rFSM (reduced Finite State Machines) which is a minimal variant of UML hierarchical Statecharts for the domain of Coordination in Robotics.

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 [9] [10] [11]. The aspect of Communication defines how system parts communicate with each other and what are 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 [12] [13] [14] and are well suited for modeling Coordination for the following 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 therewith generated UML Statecharts have good compositional properties which enable incremental development of large Coordination models by composition of simpler ones.

finite state machines have good compositional properties which enable incremental development of large Coordination models by composition of simpler ones.

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 actsuates 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 acceleration model will predict the ball to stop at the last estimated 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 different estimation components which do not make this application dependent assumption. A better solution is to introduce a Coordination component encapsulates this policy. The state machine for such a component is shown in figure 2.

Instead of making assumptions about the component layout out 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.

```lua
Listing 1 shows the equivalent rFSM syntax for the coordination state machine.

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 actsuates 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 acceleration model will predict the ball to stop at the last estimated 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 different estimation components which do not make this application dependent assumption. A better solution is to introduce a Coordination component encapsulates this policy. The state machine for such a component is shown in figure 2.

Instead of making assumptions about the component layout out 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.

```
it is assigned a generated function which returns the events from port \texttt{ip}. The next two lines define the following and the \texttt{pause} simple states (\texttt{sista}). The latter realizes the starting and stopping of the RobotController component in its \texttt{entry} and \texttt{exit} function respectively. The remaining code defines the transitions between the states and the events which trigger them.

B. Contributions

The contribution of this work is an approach for creating a dynamic execution environment for building robotic real-time domain specific languages based on the Lua extension language. We show how the problem of memory allocation and recuperation can be dealt with and illustrate this for a lightweight state machine implementation.

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 effectively unifies state machine simulation and execution, thus avoiding the problem of the two behaving inconsistently due to transformation artifacts. 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 [10].

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 RTT-Lua in general and specifically for the rFSM DSL. Section IV gives some experimental results on the timing behaviour of our state machine implementation. We conclude and summarize in chapter V.

II. RELATED WORK

The UML language has long been used for modeling real-time systems [15] [16], however mostly as a specification and modeling language without the intent to execute the model. More recently the UML MARTE profile [17] (Modeling and Analysis of Real-Time and Embedded systems) extends the UML modeling language with features specific to real-time and embedded systems. Peñil and al. shows in [18] how executable specifications can be generated from these models, yet additional transformations are necessary.

With respect to real-time garbage collection the work on real-time Java [19] [20] is relevant as it proves that allocation and collection of memory can effectively be performed in hard real-time. More specifically the work on time triggered garbage collection [21] is important as it describes an alternative approach to the classic allocation driven collection scheme. However unlike our approach the multi-threaded general purpose language of Java is targeted which introduces additional complexity due to parallel collection. In contrast our approach can assume the single threaded context of a component as explained in section III D.

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.

This section describes how we address these issues for RTT-Lua.

A. Avoiding run-time allocations

As mentioned earlier the simplest approach is to entirely avoid run-time memory allocations and collection as implemented for the OROCOS RTT scripting [4]. Interestingly this approach can be realized with Lua too. To achieve this a set of statements which result in memory allocations must be avoided. The most restricting statements are:

- concatenating strings
- creating tables and functions
- executing the dofile and loadstring functions
- deep recursion of non-tail calls

This way the garbage collector can be completely stopped and hence the overhead avoided. For validating this behavior we introduce a Lua command which allows to enable a hard real-time mode. If in this mode a memory allocation occurs a backtrace will be printed in order to identify the statement which caused it.

Nevertheless with this approach one is back at the level of limited expressiveness as with the original RTT scripting, although this time with a more common language and the option to loosen these constraints in exchange of more expressiveness. In order to use the full language which is desirable for the domain specific languages we have in mind one has to face both real-time allocation and garbage collection. This is described as follows.

B. Real-time memory allocation

The conventional memory allocation functions provided by modern general purpose operating systems are not real-time safe which means they might block unbounded time before returning the requested memory or failing.

Fortunately results in real-time allocation such as [22] and more recently [23] allow memory allocations with guarantees for worst case execution time. For the RTT-Lua implementation we use the TLSF memory allocator [23].

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 memory pool created at startup. Increasing the pool size at run-time 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 DSL the memory consumption depends on:

\footnote{If a function's last statement is to call a function and return its result, this statement is called a tail-call. Some languages (incl. Lua) can optimize tail-calls, so a tail-call does not consume additional stack space.}
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

The graph shows the static memory consumption (in KiB) for rFSM instances of n states and 2*n 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++ framework functions for Coordinating the actual computations and ii) no large data structures but instead mostly references to data are transferred between the scripting language and framework 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 robotic systems which are typically well equipped with memory this is in general not a problem.

C. 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 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 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.

D. Real-time garbage collection

Real-time garbage collection has been intensively researched over the last decades [24] [25] [26] 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 tightly coupled 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 sequentially execute these actions.

This does not imply that concurrency of scripts is not possible. Multiple scripts or DSLs can be executed concurrently in different components and communicate as needed by means of the underlying framework. By this a less tightly coupled form of parallelism is encouraged which has the benefit of avoiding the complexity and non-determinism of parallel garbage collection.

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 as this allows to keep the transition as short as possible.

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 rFSM DSL implementation after each run-to-completion step. As the overall memory consumption is monitored by...
the state machine 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. EXPERIMENTAL EVALUATION

This section gives some experimental results of evaluating the temporal behavior of different rFSM state machines. All tests were run using the Xenomai [27] real-time framework on a Pentium M 1600 MHz. The source code of the tests is available [28].

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 entering the target state of the transition.

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 worst case duration required for the rFSM core to transition between two states. Two things can be observed. Firstly, the transition time for deeper nested hierarchical states is higher as the rFSM core must exit and enter more states until the last transition target state 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 increase for hierarchical states containing more substates, so does the duration of the garbage collection.

V. CONCLUSIONS

We have presented an approach for using the Lua extension language for scripting the behavior of RTT Components at different levels of real-time requirements. For applications with very tight real-time constraints memory allocation and garbage collection can be avoided entirely. This follows the approach of the current RTT scripting and has the downside that only a subset of the language can be used. Using more expressive language features such as required by the rFSM DSL will result in memory allocations at run-time. We show how these can be dealt with deterministically by estimating the required memory and performing allocations using a real-time allocator.

Our approach has the following advantages. Most importantly it allows to avoid premature optimization by prototyping functionality in a high level language and to optimize later for performance or latency if necessary. Portability is
improved by use of an extension language which is easily integrated into other frameworks. At last, using a scripting language improves the overall robustness of the system: the impact of programming errors in scripts are limited to the respective interpreter and can not affect unrelated computations running in different threads of the same process.

The applicability of our approach was illustrated by the rFSM Statechart DSL. We have shown how the common problem of dealing with real-time memory allocation and garbage collection can be further simplified by making use of characteristics of the target domain. For the domain of Coordination this is the feasibility to run single threaded and the tendency to move mostly references to data structures between scripting and underlying framework. The evaluation shows, in our opinion, reasonable overhead and latencies which are quite acceptable for the robustness and flexibility gained.

The weakest point of the rFSM DSL is currently the unpredictable memory use 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


Intl. Conf. on SIMULATION, MODELING and PROGRAMMING for AUTONOMOUS ROBOTS

Darmstadt (Germany) November 15-16, 2010
ISBN 978-3-00-032863-3
pp. 284-289