Robots, Software, Mayhem?
Towards a Design Methodology for Robotic Software Systems

Dietmar Schreiner, Franz Puntigam
lnstitute of Computer Languages, Compilers and Languages Group
Vienna University of Technology
Vienna, Austria
{*dietmar.schreiner,franz.puntigam*}*@tuwien.ac.at*

***Abstract*—Current and upcoming robotic systems have to fulfill mission and safety critical tasks within real world environments. Robots like unmanned underwater vehicles and autonomous vehicles for space exploration require long-term autonomy, others like intelligent vehicles and mobile domestic robots have to closely collaborate with human individuals. Consequently, robotic system software has to be reliable and safe. Moreover, state-of-the-art robotic software is highly reactive, inherently parallel, distributed, and operating in real-time. Hence, it is complex, hard to develop, and even harder to certify. Our work aims at an improved software development methodology that on the one hand allows high level development of certifiable robotic software and on the other hand is capable of synthesizing optimized low-level code for robust concurrent real-time environments. To reach this goal we rely on methodologies based on timed automata and on static tokens to semi-automatically guarantee the absence of resource conflicts.**

***Keywords*-Concurrency control; Mechatronics; Real time systems; Robot programming;**

I. Motivation

The development of autonomous robotic systems has experienced a remarkable boost within the last years. Away from stationary manufacturing units, current robotic systems have grown up into autonomous, mobile systems that not only interact with real world environments, but also fulfill mission critical tasks in collaboration with human individuals on a reliable basis. Typical fields of application are unmanned vehicles for exploration but also for transportation, reconnaissance and search-and-rescue in hazardous environments, and ambient assisted living for elderly or disabled people.

However, the back-side of this boost is an even larger increase in complexity of those systems. Numerous actuators and sensors have to be controlled simultaneously. Complex actions have to be performed via timed parallel execution of multiple instruction streams on distinct electronic control units. Autonomy, especially long term autonomy as required by deep-sea or space exploration missions, necessitates features of fault-tolerance, error recovery, or at least well-defined fallbacks. Due to the physical interaction of robots with the real world, safety violations are extremely harmful, in the worst-case they might lead to severe damage and even to casualties. Consequently, robotic software typically exacts guaranteed safety in order to protect human life.

II. Problem Statement

State-of-the-art robotic software is highly concurrent at process level, but also at thread level within single processes. System resources range from shared memory up to bus systems, sensors, and actuators. As a consequence, development of robotic system software has become quite complex and thus error prone. Manual synchronization of access to shared resources as much as deriving complex timing constraints within the distributed robotic system put a major burden on software developers, and hence introduce a potential threat for software dependability. To overcome this issue, a new development methodology for robotic system software is urgently needed.

III. Solution

Our research aims at a semi-automatic methodology for the development of concurrent, reactive software within the robotics domain. The methodology should be able to detect race conditions on specific global resources under user guidance. By specifying execution models of the robotic software via timed automata, worst-case execution time estimates, and access protocols to shared resources, a compiler will be capable of injecting synchronization artifacts at truly affected locations only. In addition, expensive locks can automatically be avoided if execution timing leads to infeasible simultaneous accesses. Consequently, developers will no longer have to handle synchronization issues manually.

A. Timed Automata

Robotic software is typically developed by designing finite state machines which describe a system's behavior in terms of internal states and external events. An abstract state is characterized by a set of actions executed while within the state, and by a set of shared resources that are accessed by the state's actions. Events are abstract representations of well defined changes within shared resources or within the robot's perceptible environment. A robot resides within a

given state (so executes the state's actions) as long as no event induces a transition to another state. In addition, the actions within one state, but also multiple concurrent state machines have to be executed in predefined time frames, and have to be coordinated in a well-timed manner to reach complex goals. Hence, the described state machines have to be augmented by timing-related state transitions [1].

In a first step, the states of the modeled timed automata have to be mapped to the implementation's source code. Within the chosen methodology states are identified semi-automatically by manually identifying resources that are of relevance for the specific state, and by defining the transitions to subsequent ones. Based on this information a static analysis at source code level will be able to automatically identify all actions that are executed within one state, and will also be able to collect information on timing of usage of shared resources [2]. In that way, the program's flow of control is mapped to coarse grained timed automata that provide a rather formal vehicle for further use in token based synchronization. One benefit of this approach is the capability of handling synchronization at a coarse grained level. However, the level of detail at specific points within the system's model can be increased easily by introducing sub-states if required.

B. Constraining Access to Shared Resources

We need a mechanism to ensure that accesses to shared resources correspond to the related automata. Process types [3] and typestates [4] seem to be appropriate for this purpose: The type of each reference to a shared resource encodes information about the current state of the corresponding automaton as far as visible through the reference, supported kinds of access depending on the state, and state changes associated with each kind of access. Static type checking anticipates state changes, ensures that all accesses occur when automata are in appropriate states, and avoids conflicting accesses. State information is itself a critical resource. To keep the system tractable we represent each state as a set of abstract entities called tokens and distribute the tokens over all references to the corresponding shared resource (or more accurately over the types of the references). These tokens are not shared since each of them belongs to a single reference. We can access shared resources only through references having all tokens required for the access, and on access this set of tokens is replaced by another set to reflect state changes. Tokens help the static checker to anticipate states and state changes and to avoid race conditions. Usually, tokens do not exist at run-time. In a specific setting based on the Actor model [5], this rather static approach is all we need for proper synchronization. However, for the more complex software of robots – especially for dealing with time constraints – we have to extend the model.

As an example, let a robot be able to move and do experiments. Moving and experimenting are largely independent activities modeled by different automata. The moving automaton has states identified by tokens like *waiting* and *moving*, and the experimenting automaton like *phase1*, *phase2*, etc. Interdependences can be handled: If taking a soil sample in *phaseN* requires the robot not to move, this action requires two tokens, *phaseN* and *waiting*. Static checking ensures that previous experimentation phases have completed and the robot is not moving while taking the soil sample. Otherwise independent processes controlling movement and experiments have to cooperate to make *waiting* available for the experiment. Such cooperation can be expensive and shall be avoided if possible. Timed tokens (these are tokens valid only within specified time slots) are helpful in this respect: Let a token give us access to parameters controlling the movement. Then, several references in different processes can have this token associated with non-overlapping time slots; each process has exclusive access to the shared parameters at different periods of time. The process that controls experiments can inspect the parameters without cooperating with the movement process, but cannot prevent the robot from moving during an experiment.

IV. Conclusion

Robotic software is highly concurrent, reactive, and timing sensitive. Manual synchronization of access to shared resources is difficult and error prone. To overcome this issue, a development methodology is needed, which allows a compiler to detect and resolve resource conflicts under given execution models and timing bounds. The targeted approach will help us to increase software quality as much as improve its certifyability.

References

[1] R. Alur and D. L. Dill, "A theory of timed automata," *Theor. Comput. Sci.*, vol. 126, no. 2, pp. 183–235, 1994.

[2] D. Schreiner, M. Schordan, G. Barany, and K. M. Göschka, "Source Code Based Component Recognition in Software Stacks for Embedded Systems," in *Proceedings of MESA 2008*. IEEE, October 2008, pp. 463–468.

[3] F. Puntigam, "Coordination requirements expressed in types for uactive objects," in *Proceedings ECOOP'97*, ser. Lecture Notes in Computer Science, M. Aksit and S. Matsuoka, Eds., vol. 1241. Jyväskylä, Finland: Springer-Verlag, Jun. 1997, pp. 367–388.

[4] K. Bierhoff and J. Aldrich, "Lightweight object specification with typestates," in *ESEC/FSE-13*. Lisbon, Portugal: ACM Press, Sep. 2005, pp. 217–226.

[5] G. Agha, I. A. Mason, S. Smith, and C. Talcott, "Towards a theory of actor computation," in *Proceedings CONCUR'92*, ser. Lecture Notes in Computer Science, vol. 630. Springer-Verlag, 1992, pp. 565–579.