Control System Framework for Autonomous Robots Based on Extended State Machines

Torsten Merz[1] Piotr Rudol[1] Mariusz Wzorek[1]

Department of Computer and Information Science,
Linköping University, SE-58183 Linköping, Sweden

Abstract

We present a new framework optimized for the design, implementation, and testing of control systems for autonomous robots. It is based on a new visual specification language which specifies both control and data flow, and which is suited to be interpreted in real-time. The framework is divided into a comprehensive development and a lightweight run-time environment. The latter is fully integrated with a real-time operating system and permits to reconfigure a control system without compilation at run time. Moreover, communication in distributed systems is supported. An earlier version of the framework has been successfully applied in an autonomous helicopter and an autonomous ground vehicle project.

1. Introduction

Building control systems for highly autonomous robots for real-world applications is an open problem. Apart from finding ways to achieve autonomy in principle, we have learned from several robot projects that system integration and testing is a major problem. We argue that it is possible to achieve a relatively high level of autonomy using techniques available today in combination with a suitable framework.

A suitable framework should permit to develop and test system components independently according to a given specification, minimize their dependencies, enable their seamless integration, and make it possible to model, analyze, and control the system behavior easily. For testing an integrated system the framework should provide information about which component is active at what time and what are input and output data of system components. Moreover, we believe that a suitable framework should permit to reconfigure a system at run time. Further requirements are discussed in [9].

[1]Supported by the Wallenberg Foundation, Sweden

The main contribution of the work presented in this paper is the development of a framework which offers a unique combination of the following features. The framework (1) comprises the control, the data processing, and the communication part of a system, (2) supports reconfigurability at run time, (3) is based on a single, well-defined specification language, (4) supports all stages of development, (5) is optimized for control systems for autonomous robots, (6) has been successfully applied in several robot projects.

Related work can be found in many fields: robotics, control theory, embedded system design, and software engineering. We looked at existing methods and included them in the framework if they turned out to be useful in practice for building robotic systems. There are many specification languages, programming languages, software frameworks, and design tools which are used in control or embedded system design (Ptolemy II, Esterel, Stateflow, UML 2, among others) but none of them is optimized for building control systems for autonomous robots.

The specification language and the computation model we propose is mainly influenced by Harel's Statecharts formalism [6]. State machine based approaches have already been used successfully in many robotic systems. Brooks [3] for instance uses state machines to build reactive systems and Kleinehagenbrock et. al. [7] include them in a deliberative/reactive system. Albus et. al. [1] propose an architecture for intelligent hybrid control systems which has some similarities with our framework. It also includes state machines, defines a hierarchy of functional modules and includes a communication system, but it lacks some of the features mentioned above. Our framework supports the component-based design methodology. In [4] a component-based framework is proposed which aims for similar goals but it also does not provide some of the features mentioned above.

The remainder of this paper is structured as follows. Section 2 describes the basic concepts of the framework. The extended state machine language is defined and explained in Section 3. Section 4 describes the implementation of the framework and Section 5 concludes.

2. Basic concepts

The specification of the framework we propose is derived from the requirements for control systems for autonomous robots discussed in [9] and our experience building these systems. Although its focus is on control system synthesis, its underlying formalism permits to integrate system analysis and model checking tools.

For efficiency and robustness reasons the framework is divided into a run-time environment designed for embedded computers typically used in robotics and a development environment which can be installed on standard desktop computers. Every feature that is not required for the execution of the control system is provided by the development environment. The framework is based on a specification language which can be interpreted in real-time. This permits to reconfigure a control system at run time. In order to have full control over all essential system components, the run-time environment is implemented at a very low level of the software system. We avoid building critical parts of our framework on third-party software which is not exactly specified. In all other parts we try to take advantage of existing implementations as much as possible.

The specification language is the common basis of the framework. It has to meet orthogonal demands: on the one hand it has to be expressive enough for realizing control systems for autonomous robots in general and on the other hand it has to be interpreted in real-time. We use a state machine based approach. There are several reasons for this: state machines are well suited to model reactive systems, they can be used to coordinate all kinds of tasks including deliberative services (see [5]), they can be formally analyzed, they are traceable for debugging, and interpreters can be implemented efficiently.

To cope with the combinatorial state explosion and complexity problem of finite state machines (FSMs), to enable mixed synchronous/asynchronous systems, to include data flow, and to control task execution we introduce *extended state machines* (ESMs, see Section 3). We believe that specifying control and data flow explicitly is essential for structuring complex robotic systems. In the computational model we propose, the data flow and the control flow are separated i.e. the data flow does not influence the control flow. To enable efficient implementations, data is never copied between user tasks executed by the same processor. Instead, the framework provides pointers to data which is stored in memory only once.

ESMs support the component-based design methodology. This allows to develop and test new components individually and to reuse existing components. ESMs basically specify the interaction of components. In a hybrid control system a component contains continuous control laws while the state machine defines the switching mechanism. In our framework, components are tasks or state machine containers with specified data interfaces (see Section 4). They are associated with task states or superstates of the ESM language. In the literature, a component is sometimes defined as a *binary unit* (see [4]). Binary units can be started by system commands which are part of the source code of a task while source code units or libraries are compiled or linked with task code. Tasks and state machine containers are reusable in different parts of the ESM if they are designed for reuse.

We do not use explicit external events in our framework. Instead we represent the actual (sensed) qualitative state of the robot or its system components by flags which are used as guard conditions in ESMs (see Section 3). Asynchronous events of the physical world are modeled by a special internal periodic event in combination with guard conditions. That way we shift problems with sequentializing concurrent events and delays of event communication to the state machine level. The proposed framework permits to implement an appropriate solution for each application in the ESM language.

In a robotic system timing plays a crucial role as an autonomous robot has to act in a physical world which is usually not synchronized with the execution of tasks. To avoid unpredictable delays in the system the run-time environment uses non-preemptive periodic real-time tasks which are executed sequentially by one processor. This means that tasks have to terminate within a fixed time. Non-preemptive task execution has the additional advantage that it does not require sophisticated task communication mechanisms. In the proposed framework, the task schedule is computed prior to execution from the declaration of task durations and rates. The schedule can be changed at run time to achieve some flexibility (see Section 4).

Not all tasks must or can be deterministic in time. This is the case for most deliberative services such as task and motion planners, and system services such as file handling or standard network access. Non-real-time tasks are executed preemptively within a special periodic time slot. In case a real-time task terminates earlier the remaining time is used as additional time to execute non-real-time tasks. That way the idle time of the processor is minimized although we use static scheduling. Combining real-time and non-real-time tasks in one system permits to take advantage of both. We realized this concept with a real-time patch to a standard operating system which permits both to implement a run-time environment which has full control of the processor and to run standard applications (see Section 4).

Similar to the RCS system [1] the framework permits to build a hierarchy of time horizons. At the lower levels, periodic user tasks such as device drivers or controllers run at high rates and have short durations. At higher levels less time critical but usually more complex tasks are executed

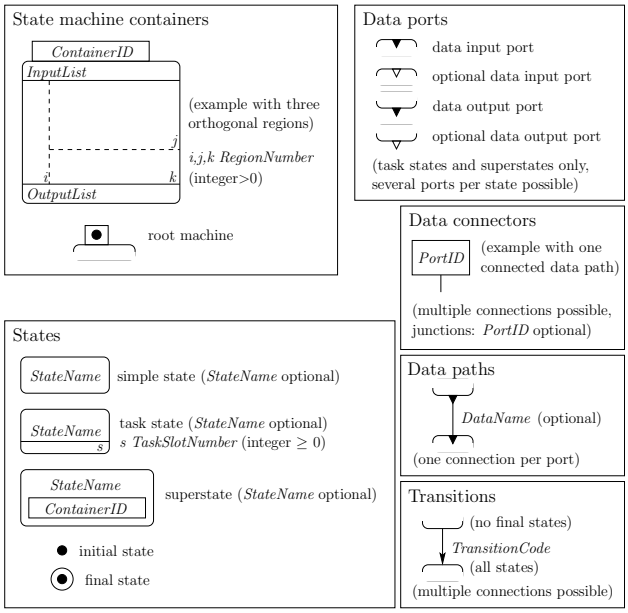

Figure 1. Visual syntax of the extended state machine language.

at lower rates. The ESM interpreter is a task which runs at a rate which makes the system responsive enough to react on events of the physical world. In multi-processor systems each processor has its own ESM interpreter.

In our framework several forms of communication exist: communication between states of the ESM language processed by the same interpreter, processors of the same computer system, computer systems of the same robot, and between different robots or robots and operators. Communication between states is synchronous or asynchronous and reliable, between processors asynchronous and reliable, between computer systems asynchronous and mostly reliable, and between robots or robots with operators asynchronous and mostly unreliable (in case of wireless transmission). The first form is realized by a common event queue and control/system flags held in ordinary memory (see Section 3), the second by passing flags and data through shared memory, and the remaining forms by transmitting flags and data periodically in packets with predefined size and rate. Similar to the task schedule the communication schedule is computed prior to execution from the declaration of packet size and rate. The time needed to transmit a packet is given by the average data rate of the link and the packet size.

The run-time environment coordinates devices for sending packages with a predefined number of bytes to remote machines, for receiving packages and for checking their integrity. It permits building real-time communication applications if the underlying network is suitable. We successfully built real-time communication links based on serial line standards (RS232C and similar) and Ethernet.

```
TransitionCode = Event [ "[" Guard "]" ] [/ ActionList] |
   *[/ ActionList];
* = "↑ [⊙]";
ActionList = Action {"," Action};
Action = "↑" EventID | ControlFlag | "¬" ControlFlag;
Event = EventID | "↑";
Guard = ["¬"] GuardFlag | Guard Op Guard | "(" Guard ")";
Op = "∧" | "∨";
GuardFlag = ControlFlag | "⊙" | "⊗" | "⊘";
ControlFlag = ("•" FlagID) | ("∘" FlagID);
InputList = Element {"," Element};
OutputList = Element {"," Element};
Element = PortID | ("↑" EventID) | ("∘" FlagID);
(* all ID symbols are ANSI C id-tokens *)
(* all Name symbols are ASCII characters *)
```

Figure 2. Syntax of symbols in Fig. 1 in EBNF notation.

The ESM interpreter logs all information needed to observe state transitions at run time or after mission execution with a state machine debugger which is provided by the framework. Log data is coded efficiently so that it can be transmitted through low bandwidth links. Beside state machine data it is possible to log any data produced by user tasks.

3. Extended state machines

We define an *extended state machine* (ESM) as a conventional deterministic transducer finite state machine with the following extensions: (1) guarded events as inputs, (2) action lists as outputs, (3) event broadcasting and filtering, (4) AND/OR decomposition of states, (5) states associated with tasks, (6) control and system flags, (7) states with timers, (8) data flow between states. This definition is different from the extended (finite) state machine definition commonly used in the literature which introduces variables for modeling quantitative aspects.

Feature (1), (2), event broadcasting, and (4) are adopted from the classical Statecharts formalism introduced by Harel [6]. A guarded event is an event with a Boolean expression (*Guard*) which has to evaluate to TRUE before triggering a transition. Flags represent the values of Boolean variables. An action list contains a sequence of actions which are executed in transitions (Mealy machine). We define two types of actions: send event (adding an event to the event queue of the ESM interpreter, see below) and set flag. There are no explicit external events in our ESM formalism and events do not carry data. We define a special periodic event (*pulse event*) which in combination with guard conditions models external events while internal events (events produced by the state machine itself) are used for broadcast communication. We introduce a filtering mechanism which limits the scope of internal events.

ESMs permit AND and (X)OR decomposition as the Statecharts formalism does. AND decomposition introduces concurrent states (orthogonal decomposition) while (X)OR decomposition enables different levels of abstraction by hierarchical structuring of states.

We distinguish, as Harel does, between actions and activities. Actions have no duration (zero-time assumption) and are executed in transitions, while activities take time and are associated with states. A state can represent any activity of a system. We define three types of states: *simple states*, *superstates*, and *task states*. Superstates represent nested states at a higher level of abstraction (AND/OR decomposition) while task states represent user defined program code being executed. The ESM formalism does not define a semantics for simple states. *Control flags* are introduced to be able to configure submachines or tasks and to react on their outcome. *System flags* are used by the state machine interpreter and the task dispatchers (see Section 4) to enable the execution of tasks and to indicate that a submachine reached a *final state*, a timeout occurred, a task is finished, or that it cannot be executed. All states have a timer. It is started when a state is entered and a system flag is set when a state specific time limit is reached. Finally, our formalism permits to specify data flow between task states, superstates, and both of them.

We use a visual language for representing ESMs based on state diagrams similar to Statecharts. Its syntax is defined in Fig. 1 and Fig. 2 (textual symbols of the language are written in italics in Fig. 1). We introduce several special characters as terminal symbols to shorten labels in state diagrams and to avoid confusion with names and identifiers. The symbols are described in the text of this section and in Fig. 3.

Symbol	Context	Description
EventID	*Event*	receive event *EventID*
↑	*Event*	receive pulse event
⊙	*Guard*	read exit flag of superstate or task state
⊘	*Guard*	read timeout flag of superstate or task state
⊗	*Guard*	read busy flag of superstate or task state
∘*FlagID*	*Guard*	read control flag *FlagID* of parent superstate
•*FlagID*	*Guard*	read control flag *FlagID* of superstate or task state
↑*EventID*	*Action*	send event *EventID*
∘*FlagID*	*Action*	set control flag *FlagID* of parent superstate
•*FlagID*	*Action*	set control flag *FlagID* of next superstate or task state
↑*EventID*	*InputList*	forward event *EventID* from upper machine level to all regions of lower machine level
PortID	*InputList*	input port of superstate
∘*FlagID*	*InputList*	declaration of input control flag *FlagID*
↑*EventID*	*OutputList*	forward event *EventID* to all regions of upper machine level
PortID	*OutputList*	output port of superstate
∘*FlagID*	*OutputList*	declaration of output control flag *FlagID*

Figure 3. Description of symbols.

User defined tasks are associated with task states. Each task state represents an instance of a task. All real-time instances of tasks are executed in non-preemptive task slots identified by their *TaskSlotNumber* while non-real-time instances are executed in preemptive threads (TaskSlotNumber=0). A special task attribute declares execution restrictions (e.g. mutual exclusive execution). All control flags of a task have to be declared. Control flags can be set in a transition to a task state (• operator, input flags) and used in a guard of a transition from a task state (• operator, output flags). Each task state has its own set of flags. Beside control flags, system flags can be used in guards of transitions from task states. Data inputs and outputs are represented by triangles attached to the task state symbol (*data ports*). Each port has a task unique identifier (*PortID*) and the following attributes: data type, memory allocation information, and optional port flag. A user task can be associated with several states. This has to be considered in the implementation of such a task. Task states are described more in detail in Section 4.

A *state machine container* encloses one or more submachines (AND/OR decomposition). It contains a list of inputs and outputs and one or more regions. *Regions* encapsulate orthogonal decomposed state machines. They are ordered by a consecutive number (*RegionNumber*). Different regions inside a state machine container are separated by dashed lines. A superstate represents an instance of a state machine container at a higher levels of abstraction. Input and output data are declared in *InputList* and *OutputList* respectively. They are represented by triangles attached to the superstate symbol (data ports). The lists also declare control flags of the superstate and which events are forwarded. Control flags are set and used in guards in the same way as described above for task states. Each superstate has its own set of control flags. Their scope is limited to the regions of the state machine container. Beside control flags, system flags can be used in guards of transitions from superstates. The framework permits to reuse state machine containers, i.e. it provides all means necessary to execute multiple instances of a state machine. The reuse of state machine containers has to be considered in the implementation of included user tasks.

Data paths are used to connect data ports. Data transfer is realized by passing pointers to data structures in shared or ordinary memory along data paths. Each data port can be connected to one data path only. In order to connect several data paths with each other, *data connectors* are introduced. Data connectors are also used to associate data ports in different regions with each other and ports inside a state machine container with ports of the corresponding superstate. Ports can only be connected or associated if their properties match. Input and output ports belonging to the same state containing pointers to the same data structure in

memory can be combined (double triangle) and connected to a single path.

The semantics of the ESM execution is defined by the algorithm given in Fig. 4. In the following we define the terms

```
ExecuteStateMachine
1 lock memory with control and system flags
2 create empty event queue
3 append pulse event with global scope to event queue
4 while event queue not empty
5   remove first event Event from event queue
6   call MakeTransitions(1,Event)
7 unlock memory
8 return
```

```
MakeTransitions(MachineLevel, Event)
1  for RegionNumber = 1 to number of regions
2    at MachineLevel
3    if t − t_s > t_timeout of current local state
4      set timeout flag
5    if Event is visible and received by a transition of
6      the current local state and Guard is TRUE
7      if current local state is task state
8        and task is real-time
9        disable task execution (reset enable flag)
10     if next local state is task state or superstate
11       reset control and system flags
12     ExecuteActions(ActionList)
13     if next local state is task state
14       enable task execution (set enable flag)
15     else if next local state is final state
16       set exit flag
17     save state start time t_s = t
18     enter next local state
19   else if current local state is superstate
20     call MakeTransitions(MachineLevel+1, Event)
21 return
```

```
ExecuteActions(ActionList)
1 for each action Action in ActionList
2   if Action is send event action
3     put event with source information in event queue
4   else set control flag
5 return
```

Figure 4. Extended state machine algorithm.

used in the algorithm. States of ESMs can be represented as AND/OR-trees with superstates as AND and regions as OR nodes. The root node of the tree represents the root machine superstate and leaves symbolize atomic states. *Atomic states* are simple states and task states. We define the *current state* of the ESM as the Sub-AND/OR-tree defined by all paths from the root to all leaves representing the current atomic states. The *machine level* of an ESM is calculated by dividing the level of the AND/OR-tree containing AND or leaf nodes by two. The *current local state* is defined as the current state of the ESM at a given machine level in a given region. *Next local state* is the state which can be reached from a current local state by making one transition. An event is *visible* if it is broadcasted or forwarded along the path from the creation node to the node representing the current state in the AND/OR-tree (event scope). The creation node is the node that represents the state from which the transition with the send event action was made. Events are broadcasted to all regions of a state machine container or forwarded to other regions if they are declared in OutputList and InputList respectively.

The main procedure of the algorithm is ExecuteStateMachine which is supposed to be called periodically. The period time determines the lower bound of the time the robot needs to react to external events of the physical world or switch control modes using the ESM. During its execution, the control and system flags cannot be changed from outside. All events are stored in a single event queue and processed in the order they arrive. In the beginning there is only the pulse event in the queue which is visible in all states. *MakeTransition* processes all current local states of the current state recursively in a depth-first manner for a given event (micro step). When entering a new machine level the current local state is the *initial state* of the given region. The algorithm terminates when the event queue is empty (macro step). In practice it is sufficient, if the period time of the algorithm is about one order of magnitude higher than the shortest time between two external events which have to be processed by the ESM for the proper operation of the robot. Considering this and assuming that the algorithm terminates within the period time, the synchrony hypothesis [2] can be applied to the ESM. Note, that the effect of the time needed to compute a guard condition from sensor readings and the time it takes from enabling a task or set a control flag until an actuator moves must be considered additionally.

4. Implementation of the framework

This section describes some details of our implementation of the proposed framework. As motivated above it is divided into a development and a run-time environment. The two parts exchange information by configuration files, ANSI C source code files, and log files. Source code is only used in parts that do not require a reconfiguration at run time, if it reduces execution time significantly, or if it can not be avoided.

4.1. Development Environment

The major components of the development environment are depicted in the left-hand part of Fig. 5. They are implemented on a standard PC as an integrated application with a common GUI (Graphical User Interface). All operations involving manipulation of visual elements of the ESM language (see Fig. 1) are performed in a graphical manner by

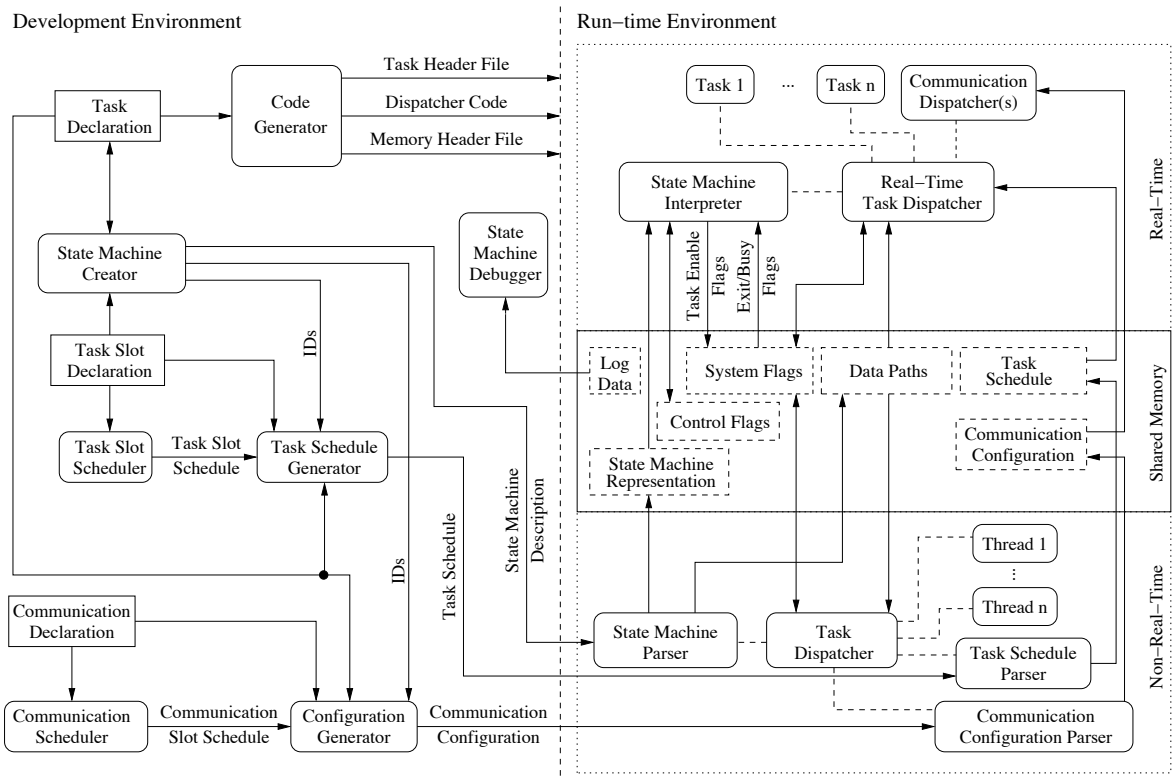

Figure 5. Framework structure.

the *State Machine Creator* using drag-and-drop technique. All textual manipulation is done using dialog boxes. Fig. 6 shows a screenshot of the *State Machine Creator*. It contains an ESM for the hovering flight mode of an autonomous helicopter developed in the WITAS project [5].

The output of the development environment is twofold: (1) in form of source code files generated by the *Code Generator* (code changes require recompilation of the run-time environment) and (2) in form of files produced by the *Task Schedule Generator*, the *State Machine Creator*, and the *Configuration Generator* which are interpreted at run time. The *Code Generator* generates C-code for the task dispatchers which contains function calls to user task functions (*Dispatcher Code*), creates a header file declaring those functions (*Task Header File*), and produces the *Memory Header File* which defines control flags for user tasks.

The task scheduling is performed in two steps. Firstly, the *Task Slot Scheduler* generates a slot schedule based on the *Task Slot Declaration*. It contains an identifier number (*TaskSlotNumber*), a duration, and a rate for each time slot. This information is used by the *Task Slot Scheduler* to find a suitable start time for every slot. A simple exhaustive search is fast enough to find the start times for the number of user tasks we require in our current systems. Secondly, the *Task Schedule Generator* creates a configuration file for the run-time environment which contains the schedule for all user tasks of an ESM based on the previously scheduled slots and the slots assigned to the tasks. One slot can be assigned to several mutual exclusive tasks with similar timing requirements (e.g. computationally expensive filters which are not required at the same time) or by concurrent tasks as long as they do not exceed the slot duration.

The communication scheduling is done in a similar way as the task scheduling. The *Communication Scheduler* generates the communication slot schedule based on information given in the *Communication Declaration*. It contains the declaration of all data links served by one processor and all outgoing and incoming frames. The declaration of each data link consists of an identifier for the associated communication dispatcher (see Section 4.2), the transfer rate, and the frame header size. For each outgoing frame the following must be declared: the frame identifier (included in the header), the communication dispatcher identifier, the data port (*PortID*), the payload size, and the slot rate. The declaration for an incoming frame consists of the identifiers of the frame, the communication dispatcher, and the data port. The *Configuration Generator* creates a configuration file for the run-time environment which contains the communication slot schedule and the mapping between frames and function arguments of communication dispatcher functions. Both the communication configuration and the task schedule can be changed at run time.

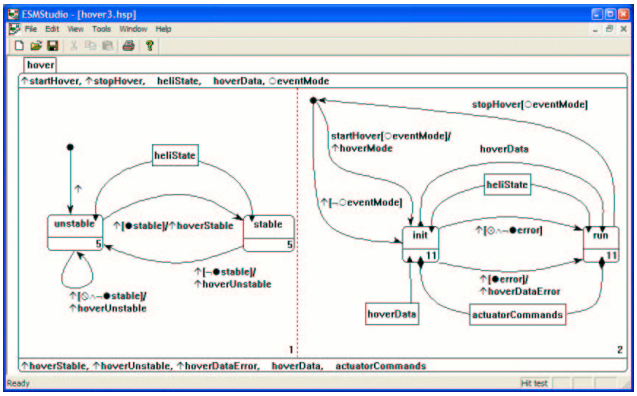

Figure 6. Screenshot of the State Machine Creator showing a state machine container.

The development environment includes a *State Machine Debugger* which is a very useful tool for investigating the behavior of a robotic system. For each point in time the debugger enables to visualize graphically the current state of an ESM and evaluate all guard conditions of connected transitions. The required information is either received directly from the run-time environment while the robotic system is operating or read afterwards from a log file which was created during a mission. The debugger offers a step function which permits to browse through every micro step.

Thanks to the well defined semantics of the ESM language it is possible to perform numerous verification checks for a designed system. There exist many tools for the verification of FSMs that can be applied. Additionally, errors specifically related to ESMs such as attempting to run two exclusive tasks at the same time or connecting ports with mismatching types can be found automatically. The framework also supports simulations. It is possible to simulate state transitioning by providing models which describe system and control flags of user tasks and to simulate data flow by providing models for the generation or transformation of data.

4.2. Run-time Environment

The major components of the run-time environment are shown on the right hand side of Fig. 5. We have implemented it for PC/104 embedded computers which are commonly used in robotics. Nevertheless, porting it to different architectures is easy. As operating system we use Linux with a RTAI (Real-Time Application Interface) kernel patch. RTAI is a hard real-time extension to the Linux kernel developed at the Department of Aerospace Engineering of Politecnico di Milano (DIAPM) [8]. It provides the features of an industrial-grade real-time operating system (RTOS) together with the possibility to use standard Linux OS services and standard Linux applications. In the run-time environment described here those and non-real-time user tasks are executed preemptively within a special periodic time slot (*TaskSlotNumber* = 0). User tasks are implemented as ANSI C functions. Binary applications can be started from non-real-time user tasks.

The core of the run-time environment consists of a real-time module (RTM) and a user space application (UA). Shared memory is used for communication between the module and the application. The RTM includes all components from the real-time part of the framework and the UA those of the non-real-time part. The execution of tasks, both in the real- and non-real-time environments is coordinated by two task dispatchers that use the task schedule generated in the development stage.

The *State Machine Interpreter* (SMI) is implemented as a periodic real-time task and interprets ESM code according to the algorithm presented in Fig. 4. Each period corresponds to one macro step. The SMI uses *System Flags* to coordinate the task execution in both, real-time and non-real-time parts. A global event queue is used internally. For each event it holds the identifier for the event and the identifier for the state from which the corresponding transition is made. In order to propagate events efficiently it is necessary to find the scope of the event quickly. The scope is defined by the ESM structure and the event filters (see Section 3). In our implementation we use an event lookup table, which holds a list of states for each event/state identifier pair. The table is created during the initialization of the SMI.

As mentioned in Section 3 data flow is pointer based. Pointers are passed to user task functions as function arguments which correspond to data ports. The connections are stored in shared memory (*Data Paths*) according to the ESM. Memory for the data must be allocated and released by user tasks. Because of the different address spaces used in the real- and the non-real-time parts it is necessary to provide tasks to correct data pointers with an offset when using both types of memory.

The startup procedure of the run-time environment is as follows: The UA is started and the RTM is inserted. Then they synchronize via shared memory. The UA runs the *State Machine Parser*, the *Communication Configuration Parser*, and the *Task Schedule Parser* which initialize the data structures in shared memory. All parsers use the configuration files created in the development environment. The RTM sleeps until the UA initialization process is finished. Then it executes the SMI once. All further initialization steps such as starting the SMI task have to be specified by the ESM. At that point the startup procedure is finished and the regular schedule is executed.

The communication between different machines, robots, or robots and operators is realized by communication dispatchers (CDs) which are associated with task states in ESMs. For each data link exists a separate CD. A phys-

ical data link contains one or more virtual channels which are mapped to the ports of a task state. CDs for transmission and reception are separate tasks. Each input of a transmission CD contains a pointer to a data packet which is sent in a single frame during a communication time slot defined in the communication configuration (parsed file). A frame is not copied to the transmission buffer of the communication device driver if it does not fit. In this case the transmission CD sets a control flag to be able to react to this problem. Communication time slots are usually periodic. There is no acknowledgment of received frames.

The reception CD allocates payload buffers for each output. Each received frame carries an identifier which assigns the frame to a task port according to the communication configuration. A frame is only copied to a buffer after a checksum validation. The status of the reception is reported by the following control flags: frame available, frame delayed and frame errors. The communication mechanism is independent of underlying communication device drivers.

During the ESM execution data containing state information and guard conditions is directly sent to the development environment and written to shared memory (*Log Data*). The data is coded in a way that it can be sent through a low bandwidth link and that it consumes minimal memory space. Recorded log data is written to a log file after a robot mission is finished.

5. Conclusion

We presented a framework which has the following characteristics: (1) it is based on a concise formalism with a well-defined semantics which specifies control and data flow, (2) it is optimized for robotic systems but not limited to a specific robot architecture, (3) it permits to build deterministic real-time control systems, (4) it includes an ESM controlled communication mechanism, (5) it can be distributed on several computer systems or processors, (6) it offers possibilities for debugging, simulation, and model checking, (7) it supports the component-based design methodology and the reuse of components, (8) it is suitable for rapid prototyping, (9) it supports reconfigurability at run time, (10) it enables lightweight, efficient and robust implementations of control systems. To our knowledge there is no other framework proposed in literature with comparable characteristics.

The framework design mainly evolved from our experience building control systems for autonomous robots. We successfully applied an earlier version of the framework in an autonomous helicopter and an autonomous ground vehicle project. The new version includes all features described in this paper and is about to be finished.

The ESM language we propose provides a well-defined formalism to describe a system. This is essential for building model checkers and execution monitoring mechanisms. In contrast to FSMs, the use of ESMs prevents combinatory state explosion even for large robotic systems, permits to realize mixed synchronous/asynchronous state machines, and incorporates data flow and task execution control mechanisms. Hierarchical decomposition enables abstraction of state machines, their reuse, as well as inheritance.

Thanks to the separation of development and run-time environment, building new systems (including systems with non-PC compatible hardware) only requires design of an ESM, implementation or porting of associated user tasks, and porting of the run-time environment. We assume that the ESM language is abstract enough to reuse existing ESM designs for new applications without or with only little modification.

Future work includes integrating model checking tools, extending our existing robotic systems, and confirm the versatility of the framework. Thanks to the component-based design methodology supported by the framework and the inclusion of data flow into ESMs, we are expecting a seamless incorporation of more deliberative services such as task planners and a straightforward integration of new sensors and effectors.

References

[1] J. Albus and F. Proctor. A reference model architecture for intelligent hybrid control systems. In *Proc. of the 13. World Congress, IFAC*, June (30-5) 2000.

[2] A. Benveniste and G. Berry. The synchronous approach to reactive and real-time systems. *Proc. of the IEEE*, 79(9):1270–1282, 1991.

[3] R. Brooks. A robot that walks; emergent behaviors from a carefully evolved network. In *Proc. of the 1989 IEEE Int'l Conf. on Robotics and Automation (Vol. 2)*, pages 692–696, May (14-19) 1989.

[4] A. Brooks *et. al*. Towards component-based robotics. In *Proc. of the Int'l Conf. on Intelligent Robots and Systems*, August (2-6) 2005.

[5] P. Doherty. Knowledge representation and unmanned aerial vehicles. In *Proc. of the Int'l Conf. on Intelligent Agent Technology*, pages 9–16, 2005.

[6] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[7] M. Kleinehagenbrock *et. al*. Supporting advanced interaction capabilities on a mobile robot with a flexible control system. In *Proc. of the Int'l Conf. on Intelligent Robots and Systems*, September 28 – October 2 2004.

[8] P. Mantegazza *et. al*. RTAI: Real time application interface. *Linux Journal*, 72, April 2000.

[9] T. Merz. Building a system for autonomous aerial robotics research. In *Proc. of the 5th IFAC Symp. on Intelligent Autonomous Vehicles*, July (5-7) 2004.