Tabular Notations for State Machine-Based Specifications

Markus Herrmannsdörfer, Dr. Sascha Konrad, and Brian Berenbach
Siemens Corporate Research

Finite state machines are a widely used concept for specifying the behavior of reactive systems. Numerous graphical notations based on finite state machines have been developed and are commonly used today, such as state transition diagrams, Harel statecharts, and Unified Modeling Language (UML) state machine diagrams. While not as widely used, tabular notations for state machine-based specifications offer complementary advantages to diagrammatic notations. In this article, we describe five approaches using tabular notations for state machine-based specifications and evaluate these approaches for use in software development.

The term *reactive system* describes a system that needs to continuously react to inputs coming from the environment. Finite state machines are a widely used concept for specifying the behavior of such systems. Since finite state machines allow the rigorous capture of functional aspects of system behavior[1], they offer several advantages over informal specifications. For example, they provide the ability to automatically generate code or test cases, and they enable formal verification and validation (V&V). Generally, a finite state machine is an appropriate representation when a problem or solution has the following characteristics:

- Finite and discrete set of states (e.g., on, off, and standby).
- Discrete and manageable set of inputs.
- Change of state is only performed in response to an input (e.g., if a button is pressed, then the machine transitions from state off to state on).

State machines[2] are used for specifying functional properties for a wide variety of systems, such as control systems and user interfaces. For example, Siemens uses state machines to precisely specify the circuitry in mail sorting systems and the controls in car radios. They are also the paradigm of choice for software compiler design and programmatic interpretation of natural language. Numerous graphical notations for state machines have been developed and are commonly used today, such as state transition diagrams, Harel statecharts [1], and UML state machine diagrams [2]. Graphical notations are often preferred by developers, analysts, and testers over textual information, since diagrams allow the visualization of complex relationships.

Tabular notations for state machines (commonly also referred to as *state tables* or *state transition tables*) offer complementary advantages to these graphical notations. For example, the incompleteness of a specification, i.e., the actions of the system in a specific state in response to a specific event that are not addressed by the specification, can easily be identified as empty cells in the table. In addition, tabular notations are relatively compact and have shown to scale well to practical systems [3]. Due to these reasons, tabular notations for state machines are preferred in some domains over graphical notations for the rigorous specification of system behavior. For instance, Siemens Automotive commonly receives system requirements in the form of state tables, captured in either Excel sheets or proprietary databases.

While a tabular representation is relatively compact and the completeness of the requirements specification can easily be determined, it has been shown to cause numerous difficulties. For instance, the requirements specification for a system of realistic size is often quite large and of considerable complexity, consisting of numerous large tables. As a result, precisely understanding the required behavior solely through visual inspection is difficult. Moreover, requirements captured in simple Excel sheets are difficult to analyze for consistency and adherence to critical properties.

This article presents and evaluates several state machine-based tabular notations that can address some of the aforementioned problems. For instance, some notations enhance the understandability of the specification by offering a complementary graphical representation. In addition, hierarchical composition is used by several notations to keep the specification tractable and some provide tool support for V&V. The remainder of this article is organized as follows: the Background section provides an overview of finite state machines and Harel statecharts. The Tabular Notations for State Machines section describes five approaches using tabular notations for state machine-based specifications. We conclude by evaluating these notations for use in software development with respect to several factors.

Background

This section introduces finite state machines, including a common graphical and tabular notation, and briefly describes the advanced features of Harel statecharts.

Finite State Machine

The term *finite state machine* describes a class of computational models that consist of a finite set of states, a start state, a set of inputs (events), and a transition function that determines the next state of the finite state machine based on the current state and input [4]. The finite state machine starts computation in the start state; transitions between states are performed based on the transition function. Numerous variants of this basic type of state machine exist. For example, Moore machines extend finite state machines with outputs (actions) associated with states, while Mealy machines associate outputs with transitions [5]. For the remainder of this article, we use Mealy machines as the computational model. Finite state machines may be deterministic or non-deterministic. In deterministic finite state machines for a given input, one transition can be taken from the current state, at most. In non-deterministic finite state machines, however, one input may enable several transitions of which one is then taken.

A common way of representing finite state machines is the use of *state transition diagrams* (commonly also referred to as *state diagrams*). State transition diagrams are directed graphs in which states are depicted as nodes and transitions are represented by directed edges. Transitions are commonly labeled with the triggering events and actions, using the following general syntax: *trigger/action(s)*. Figure 1 contains a sample state transition diagram showing the simple behavior of a door: The door can be opened or closed.

If the door is closed, then it can be locked or unlocked. The door can only be opened when it is unlocked. If the door is closed (irrespective of being locked or unlocked), then it can be pushed in. Because of this event, an alarm will sound and the door will then be permanently in the state *Broken*.

In addition to the graphical representation, finite state machines may be specified using state transition tables. A state transition table denotes the action performed by the automaton and the next state based on the current state (row) and event that occurred (column). A dash denotes that no such transition exists. A state transition table representing the automaton specified in Figure 1 can be seen in Table 1. Using this tabular notation, completeness of the specification can be readily established. Since a cell needs to be labeled explicitly with a dash if no such transition exists, a cell that does not contain a destination state or a dash renders the specification incomplete. Using a graphical notation, determining the completeness of the specification is more difficult, since a missing arrow in a diagram could potentially be the result of an omission, but could also mean that no such transition exists.

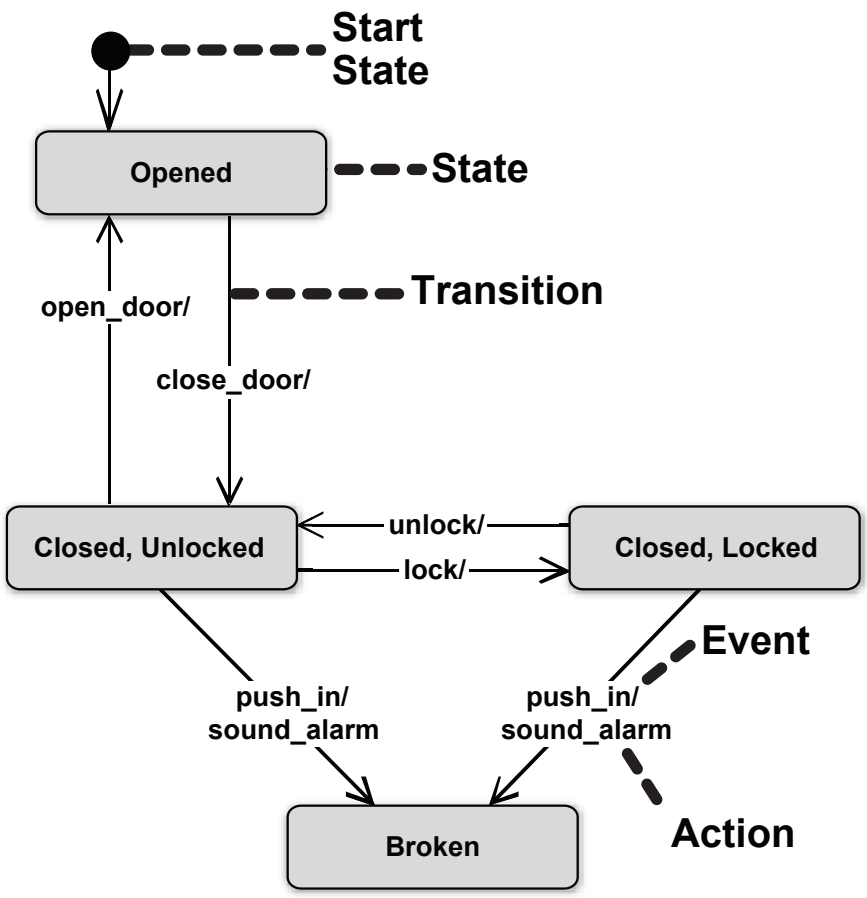


Figure 1: *Sample State Transition Diagram*

Harel Statecharts

While finite state machines have shown to be useful for modeling reactive systems, their representation as state transition diagrams does not scale well to large-scale systems and may become *unstructured*, *unrealistic*, and *chaotic*. To address this problem, David Harel developed statecharts that extend state transition diagrams with the following concepts [1]:

1. **Depth.** Commonly also referred to as *XOR (eXclusive OR) decomposition* or *state nesting*. Using state nesting, a state may be a composite state that contains exactly one region serving as a container for sub states. To be in the composite state, the system must be in exactly one of its sub states (which itself may be composite states again).
2. **Orthogonality.** Also known as AND decomposition. Using *AND decomposition*, a state may be a composite state comprising two or more orthogonal regions executing independently and concurrently. Therefore, to be in the composite state, the system must be in a state of all of its orthogonal regions at the same time. Each orthogonal region may itself contain additional composite states.
3. **Broadcast communication.** Since orthogonal regions are independent and execute concurrently, the computational model of statecharts uses broadcast communication. As a result, each orthogonal region receives occurring events and may take transitions that had become enabled.

In addition to these basic extensions, Harel statecharts provide additional constructs such as entry and exit actions for states, conditionals, and history states (see [1] for more details). The UML notation for state machines is based on Harel statecharts and uses a number of these extensions. (For a detailed comparison of the syntax and semantics of UML state machine diagrams and Harel statecharts, please refer to [6].) Figure 2 (see page 20) shows how statecharts provide more structure and reduce the perceived complexity of the diagrammatic representation of the door example in comparison to the state transition diagram in Figure 1. For instance, after the introduction of a composite state *Closed* in Figure 2, describing the behavior of the door when it is pushed in requires only one transition, which makes the diagram appear cleaner and less cluttered. In general, the extensions provided by Harel statecharts and UML state machine diagrams have shown to be an effective means to reduce the perceived complexity of state machine representations for reactive systems. For example, the authors in [7] performed some studies with university students and concluded that the use of composite states in UML state machine diagrams improves understandability.

Table 1: *State Transition Table Corresponding to Figure 1*

Event / State	open_door	close_door	lock	unlock	push_in
Opened	-	/Closed, Unlocked	-	-	-
Closed, Unlocked	/Opened	-	/Closed, Locked	-	sound_alarm/ Broken
Closed, Locked	-	-	-	/Closed, Unlocked	sound_alarm/ Broken
Broken	-	-	-	-	-

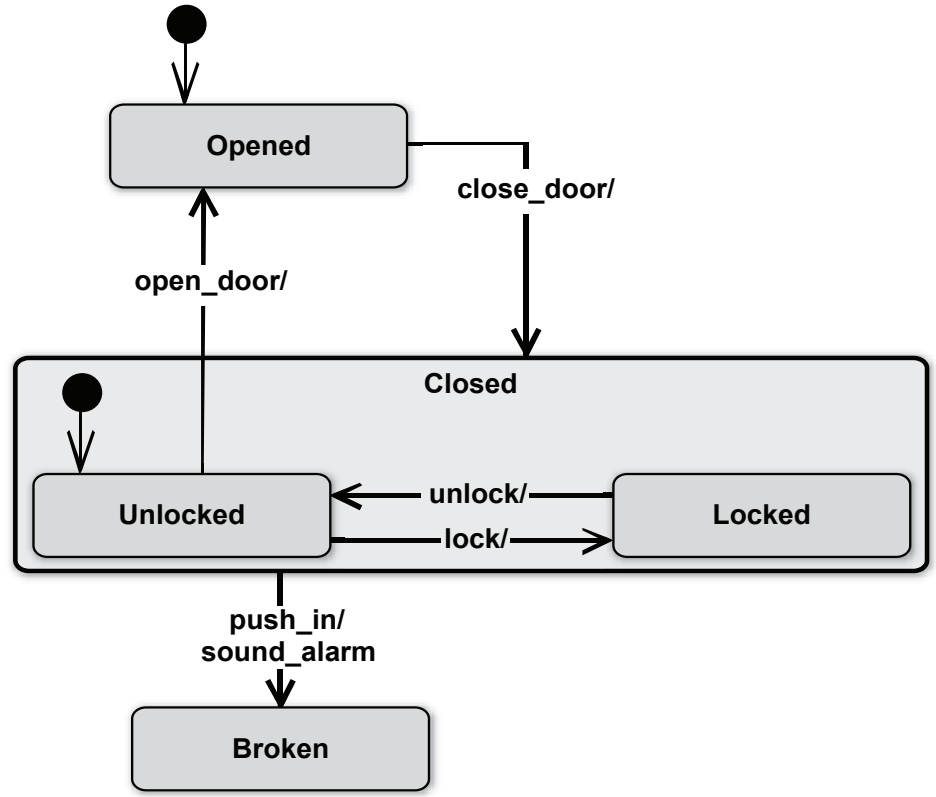


Figure 2: *Sample Statechart*

Tabular Notations for State Machines

This section describes five approaches that use tabular notations for state machine-based models, namely Virtual Finite State Machines (VFSM) [8], Software through Pictures (StP) [9], Parnas Tables [10], Software Cost Reduction (SCR) [3], and the Requirements State Machine Language (RSML) [11].

VFSMs

The VFSM is a concept for the specification of control systems in a virtual environment. The environment is termed virtual since events and actions of the environment are represented by abstract names for inputs and outputs in the state machine [8]. The behavior of the system may be specified as a *state table* that shows actions and transitions performed in a certain state based on specific conditions. Table 2 shows a sample state table for $State_1$. Upon entering the state, $Output_1$ is always produced and upon exiting the state, $Output_2$ is always produced. If $Condition_1$ is satisfied, then $Output_3$ is produced without causing a state change (internal transition). However, if $Condition_2$ is satisfied, then $Output_4$ is produced and the state machine transitions to $State_2$ (external transition).

While VFSMs support entry and exit actions, they do not support state nesting or orthogonality. However, different sets of concurrent high-level and low-level finite state machines can be created and connected to achieve structuring through hierarchical decomposition [12].

The application of VFSMs is facilitated by StateWORKS Studio, a tool suite for creating specifications using VFSMs [13]. The tool suite offers an editor that combines and synchronizes diagrammatic and tabular views of VFSMs. In addition, a simulator and an executor are provided that can be used to validate and execute VFSM specifications.

Table 2: *VFSM State Table*

State name	Condition(s)	Action(s)	
$State_1$	Entry action	$Output_1$	
	Exit action	$Output_2$	
	$Condition_1$	$Output_3$	Internal transitions
	…	…	
$State_2$	$Condition_2$	$Output_4$	External transitions
	…	…	

StP

StP Structured Environment (SE) is a tool-supported approach for specifying a system using diagrammatic notation complemented with tabular notation [9]. The behavior of a system is specified in terms of control flow diagrams and state transition diagrams. Complementary to state transition diagrams, two tabular notations are provided: *state event matrix* and *state transition table*.

A state event matrix shows all transitions of the state machine in a grid of source states and triggering events. Similar to the state transition table shown in Table 1, a transition is entered into the cell at the intersection of its source state (row) and its triggering event (column). The cell contains the list of actions to perform and the target state of the transition. Table 3 shows an example state event matrix, in which the state machine transitions from $State_1$ to $State_2$ upon occurrence of $Event_1$, producing $Action_1$, and it transitions back to $State_1$ upon occurrence of $Event_2$, producing $Action_2$. If $Event_3$ occurs in $State_1$, then the state machine performs $Action_3$ but remains in the current state.

A state transition table shows all transitions of a state machine in a list (refer to Table 4). The tabular layout provides a column for the source state, the triggering event, the action, and the target state.

Table 3: *StP SE State Event Matrix*

State	Event		
	$Event_1$	$Event_2$	$Event_3$
$State_1$	$\frac{Action_1}{State_2}$		$\frac{Action_3}{State_1}$
$State_2$		$\frac{Action_2}{State_1}$	

The tabular notations provided by StP SE are compact and readable. However, diagrams and tables of StP SE provide neither state nesting nor orthogonal regions. Similar to VFSMs, structuring is possible using hierarchical decomposition. In order to facilitate the implementation phase, the StP SE tool suite provides code generation and reverse engineering capabilities for the C programming language.

Current State	Event	Action	Next State
$State_1$	$Event_1$	$Action_1$	$State_2$
$State_1$	$Event_3$	$Action_3$	$State_1$
$State_2$	$Event_2$	$Action_2$	$State_1$

Table 4: *StP SE State Transition Table*

Parnas Tables

Parnas and Madey developed the four-variable model as an underlying state machine model to formally specify system requirements [14]. The name of the model arises from the fact that a specification contains four distinct sets of variables:

- Variables monitored by the system (MON).
- Variables controlled by the system (CON).
- Variables that the input devices of the system read from (INPUT).
- Variables that the output devices of the system write to (OUTPUT).

The relations between the variable sets of the four-variable model are illustrated in Figure 3.

Specifically, the variables are linked by the following five relations:

- Natural constraints on the monitored and controlled variables (NAT).
- Expected change of controlled variables in response to changes in monitored variables, i.e., the actual system requirements (REQ).
- Relation of monitored variables to input variables (IN).
- Relation of controlled variables to output variables (OUT).
- Relation between input and output variables, realized by software (SOFT).

A possible notation for expressing these relations are Parnas Tables [10]. Parnas Tables are a collection of 10 table types for capturing functional and relational expressions, each having a distinct syntax and semantics. A developer should choose the table format that produces a simple, compact representation for expressing the relation at hand. For each table type, rules exist for identifying incompleteness and inconsistency.

Table 5 (see page 22) contains a sample Parnas Table of type decision table. A *decision table* can represent a function or relation where the domain is an ordered set of potentially distinct types. One dimension of the table itemizes the elements of the domain. Table 5 shows the syntax of a decision table representing the relation between two variables, A and B, and a decision that is made based on the values of these variables. For instance, Table 5 states that **if** $A = A_2$ **and** $B = B_2$ **then make** $Decision_2$.

Parnas Tables do not support nesting or orthogonality, but allow the developer to reference a function that is defined in a different table. Since Parnas Tables have completely formal semantics, tool support can be developed to check the tables automatically. However, to the best of our knowledge, such tool support is not currently available.

SCR

SCR is a set of formal methods for the design of software systems [3]. Similar to Parnas tables, SCR also uses the four-variable model as its underlying abstraction, and the relationships between monitored and controlled variables are captured in tables [10, 14].

In order to capture the relations concisely, SCR defines modes. A mode describes a set of system states in which the system exhibits equivalent behavior in response to events and conditions. Mode classes then describe the relationships between these modes and are modeled in terms of mode transition tables. In order to model complex systems with independent components, several mode classes may be constructed to capture concurrency. The occurrence of an event is denoted by a value change of a condition. $@T$ is used to specify that a condition becomes *true*, while $@F$ specifies that a condition becomes *false*.

SCR uses three different types of tables to specify a system: *condition tables*, *event tables,* and *mode transition tables*.

A condition table defines the value of a variable depending on the mode and a condition. For example, in Table 6, the variable var_3 is assigned the value *greater*, *equal*, or *less* in the modes M_1 and M_2 of mode class MC_1, depending on the values of variables var_1 and var_2.

An event table defines the value of a variable as a function of the mode and a (possibly conditioned) event. For example, Table 7 (see page 22) assigns variable var_4 a *true* or *false* value in the modes M_1 and M_2 of mode class MC_1 based on a change in value of the variable var_3.

Finally, the mode transition table defines how the mode of a mode class changes in response to events. A sample mode transition table for the mode class MC_1 is given in Table 8. Specifically, the system changes from mode M_1 to mode M_2 upon variable var_4 becoming *true*, and switches back to mode M_1 if the variable becomes *false*. Commonly, a mode transition table contains only events that change the mode; events that do not cause mode changes are omitted to increase readability.

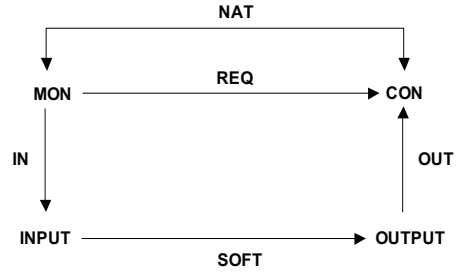
The SCR notation is rigorous and compact, but purely tabular. Nesting and orthogonality are not supported by the notation, but hierarchical decomposition can be used to structure complex systems. Tools to support various V&V approaches have been developed [3]. Once the system model is complete, the model can be checked for different types of errors, such as incompleteness or ambiguities. In addition, a simulator can be used to run scenarios and inspect whether the results are as expected.

RSML

The RSML was originally developed to write requirements specifications for process-control systems such as a collision avoidance system for a commercial airliner [11]. RSML combines a graphical notation based on Harel statecharts with a tabular notation for specifying transition conditions. As such, RSML retains most of the advanced features of statecharts, such as depth and orthogonality, while using tables to facilitate the readability of conditions associated with transitions.

RSML specifications generally consist of state diagrams with unlabeled transitions. Transitions are not labeled in order to increase the readability of a state diagram when enabling conditions of transitions are complex. Instead of using labels,

Figure 3: *Four-Variable Model* [14]



	Decision$_1$	Decision$_2$	Decision$_3$
A	A_1	A_2	A_3
B	B_1	B_2	B_3

Table 5: *Parnas Decision Table*

Mode Class MC$_1$	Conditions		
M_1, M_2	$var_1 < var_2$	$var_1 = var_2$	$var_1 > var_2$
var_3	greater	equal	less

Table 6: *SCR Condition Table for Variable Var$_3$*

Mode Class MC$_1$	Events	
M_1, M_2	@T(var_3 = equal)	@T(var_3 = greater) OR @T(var_3 = less)
var_4	true	false

Table 7: *SCR Event Table for Variable Var$_4$*

Old Mode	Event	New Mode
M_1	@T(var_4)	M_2
M_2	@F(var_4)	M_1

Table 8: *SCR Mode Transition Table for Mode Class MC$_1$*

properties of transitions are defined separately from the diagrams in transition definitions. A transition definition contains the source and destination of the transition, the state machine where the transition is located, the triggering event, the guarding condition, and the output action. The guarding condition of a transition is defined in terms of AND/OR tables. A sample AND/OR table can be seen in Table 9, which describes that the associated transition is enabled (after the triggering event has occurred) when *Expression$_1$* is *true* AND *Expression$_2$* is *false* at the same time, OR *Expression$_3$* is found to be *true*. The period denotes that the truth value of the expression is irrelevant for the current evaluation.

The final RSML specification can then be checked for consistency and completeness. In addition, techniques have been developed that allow the analysis of RSML specifications using theorem proving and model checking techniques [15]. As such, the correctness of an RSML specification can be rigorously established. Similar to SCR, tools supporting the simulation of RSML specifications also exist.

Table 9: *RSML AND/OR Table*

		OR	
AND	Expression$_1$	T	·
	Expression$_2$	F	·
	Expression$_3$	·	T

Conclusions

Due to advanced syntactical and semantical features, Harel statecharts and UML state machine diagrams are better suited than the basic state transition diagram notation to handle complex systems. Similarly, the five presented approaches use advanced features that make them better suited for complex systems than basic state transition tables. Analysts or developers that need to determine which approach to use in their development processes should consider several factors. If the main goal is to facilitate the implementation phase and reduce coding efforts, then the *VFSMs* and *StP* approaches seem preferable, since they offer commercially available tool support that allows executing or generating code from the specifications.

However, if the focus is the formal analysis of the system specification for various completeness and correctness properties, Parnas Tables, SCR, and RSML are better suited. Since Parnas Tables do not offer tool support and require the user to understand the syntax and semantics of each possible table type, they can only be recommended to developers with a solid understanding of formal methods that do not need automated support for formal analysis. In contrast to Parnas Tables, SCR and RSML offer mature tool support for V&V. When deciding between SCR and RSML, an important factor may be the availability of a graphical representation. While SCR is purely tabular, RSML uses the tabular representation only for capturing the guarding conditions of transitions, while states and unlabeled transitions are captured in terms of diagrams. We believe that such a combination of diagrammatic and tabular views combines the specific advantages each of these views offer. In addition to combining graphical and tabular views, RSML also supports the concepts of nesting and orthogonality of Harel statecharts. These concepts have shown to be effective means to reduce the perceived complexity of models. Dutertre and Stavridou have examined the use of SCR and RSML for an avionic storage management system specification and concluded that RSML specifications are commonly easier to understand than SCR specifications due to these structuring features [16].

In conclusion, we believe that while tabular notations for state machines are not a *silver bullet* solution, they may greatly facilitate the specification and analysis of systems in specific domains. Tabular notations seem to be explicitly useful for systems with a large number of transitions between states or rather complex enabling conditions of transitions. In order to retain the advantage of having a graphical view, the presented VFSM, StP SE, and RSML approaches use tables and diagrams. As such, they attempt to combine the complementary advantages of diagrammatic and tabular notations. In addition, if the system under design is rather complex (i.e., having a large number of states and transitions), notations supporting nesting and orthogonality may provide a significant reduction in specification complexity. The mature V&V tool support offered by some of the presented approaches also offers significant advantages over specifications using Excel tables or proprietary databases, where often the only available means of analysis is visual inspection.◆

Notes

1. Non-functional aspects, such as performance and reliability, are usually captured by different means.
2. For the remainder of this article, we assume that the system being specified has a finite set of states and we use the terms finite state machine and state machine interchangeably.

References

1. Harel, D. "Statecharts: A Visual Formalism for Complex Systems." Science of Computer Programming 8.3 (1987).
2. Object Management Group. "UML 2.0 Superstructure Specification." 2004 <www.omg.org/cgi-bin/doc?formal/05-07-04>.
3. Heitmeyer, D. "Tools for Constructing Requirements Specifications: The SCR Toolset at the Age of Ten." International Journal of Computer Systems Science and Engineering 20.1 (2005) <http://chacs.nrl.navy.mil/publications/CHACS/2005/2005heitmeyer-finalJCSSE.pdf>.
4. National Institute of Standards and Technology (NIST). "Finite State Machine." Dictionary of Algorithms and Data Structures. NIST 2006 <www.nist.gov/dads/HTML/finiteStateMachine.html>.
5. Hopcroft, J., and J.D. Ullman. Introduction to Automata Theory, Language, and Computation. Reading, MA: Addison-Wesley, 1979.
6. Crane, M., and J. Dingel. "UML vs. Classical vs. Rhapsody Statecharts: Not All Models are Created Equal." Lecture Notes in Computer Science 2005 <www.cs.queensu.ca/~stl/papers/MoDELS2005.pdf>.
7. Cruz-Lemus, J.A., M. Genero, M. Esperanza Manso, and M. Piattini. "Evaluating the Effect of Composite States on the Understandability of UML Statechart Diagrams." Lecture Notes in Computer Science 3713 (2005) <www.giro.infor.uva.es/Publications/2005/CGMP05/CruzLemusGeneroMansoPiattini-Models05.pdf>.
8. Wagner, F. VFSM Executable Specification. Proc. of the International Conference on Computer Systems and Software Engineering. The Hague, Netherlands, 1992 <www.stateworks.com/active/download/wagf92-software-engineer ing.pdf>.
9. Aonix. "Software Through Pictures." 2006 <www.aonix.com/stp.html>.
10. Parnas, D.L. "Tabular Representation of Relations." CRL Report 260. Ontario, Canada: McMaster University, 1992.
11. Leveson, N., M.P. Heimdahl, H. Hildreth, and J. Reese. "Requirements Specification for Process-Control Systems." IEEE Transactions on Software Engineering 20.9 (1994) <sunnyday.mit.edu/papers/tcas-tse.pdf>.
12. Wagner, F., and P. Wolstenholme. "Modeling and Building Reliable, Re-Useable Software." Workshop on Model-Based Development of Computer Based Systems. Huntsville, AL, 2003 <www.stateworks.com/active/download/wagf03-2-modeling-reliable-software.pdf>.
13. StateWORKS Software. "StateWORKS Studio." 2006 <www.stateworks.com>.
14. Parnas, D.L., and J. Madey. "Functional Documentation for Computer Systems Engineering." Science of Computer Programming 25.1 (1995).
15. Choi, Y., and M.P. Heimdahl. "Model Checking RSML-e Requirements." Proc. of the 7th IEEE international Symposium on High Assurance Systems Engineering (HASE '02). 2002 <www.umsec.umn.edu/files/47.73.model-checking-rsml.pdf>.
16. Dutertre, B., and V. Stavridou. "Avionics Systems Requirements: A Comparison of RSML and SCR." Proc. of the 16th International System Safety Conference, Seattle, WA, 2002 <www.csl.sri.com/papers/issc98/issc98.ps>.

About the Authors

Markus Herrmannsdörfer is a Ph.D. student at the Chair of Software and System Engineering at Technische Universität Munchen, Germany. During his master-level studies, he worked on the subject of this publication as an intern at Siemens Corporate Research in Princeton, New Jersey. His current focus is on metamodeling, especially on the problem of metamodel evolution.

Technische Universität München
Institut für Informatik
Boltzmannstr. 3
85748 Garching bei München
Germany
Phone: +49 (89) 289-17336
E-mail: herrmama@in.tum.de

Sascha Konrad, Ph.D., is a consultant at Siemens Corporate Research. He received his intermediate diploma from the University of Kaiserslautern, Germany, and his master's and doctorate degrees in computer science and engineering from Michigan State University. Konrad's research interests include requirements engineering and automated analysis of software specifications, including software patterns, the Unified Modeling Language, agile and model-driven software development, formal methods and computer-aided verification, and distributed and embedded systems.

Siemens Corporate Research
755 College RD East
Princeton, NJ 08540
Phone: (609) 734-6500
E-mail: sascha.konrad@siemens.com

Brian Berenbach is the technical program manager for requirements engineering at Siemens Corporate Research. He has been working in the field of requirements engineering for more than 15 years, first as a consultant, and then as a senior member of the technical staff at Siemens. Recently, his program has been involved with requirements definition for such diverse products as medical systems, baggage handling, mail sorting, automated warehouses, and embedded automotive systems.

Siemens Corporate Research
755 College RD East
Princeton, NJ 08540
Phone: (609) 734-6500
E-mail: brian.berenbach@siemens.com