UC Davis

UC Davis Previously Published Works

Title

Model-driven design and implementation of discrete event control for a machine tool control system

Permalink

<https://escholarship.org/uc/item/4tk7b72b>

Journal

International Journal of Computer Integrated Manufacturing, 20(6)

ISSN

0951-192X

Authors

Liu, Y.

Li, W.

Yamazaki, K.

et al.

Publication Date

2007-09-01

DOI

10.1080/09511920600939849

Peer reviewed

Model-Driven Design and Implementation of Discrete Event Control for a Machine Tool Control System

Yadong Liu¹, Wei Li¹, Kazuo Yamazaki¹, Makoto Fujishima²

¹IMS-Mechatronics Lab, Mechanical & Aeronautical Engineering Department,
University of California, Davis, CA 95616, U.S.A, ydlu@ucdavis.edu

²Mori Seiki Co. Ltd., 2-35-16 Meieki, Nakamura-ku, Nagoya City, Japan

Abstract

Design and implementation of discrete event control for machine tool control system is extremely complicated. In current industrial practice, designers tend to derive implementations from a rough system design in terms of system specification analysis. Such an implementation-based method leads to ad hoc system design and implementation, with system performance which relies highly on the designers' experiences. Usually a long "cycle and debug" stage is needed to fix errors after a prototype system has been built. In addition, it is always difficult to build a new system by modifying an existing one when the specification is changed. In this paper, the authors propose a Model-Driven method to enhance the design and implementation of discrete event control for a machine tool control system. Based on the system specification, an executable model is first built. This model is then evaluated by simulation to eliminate the design errors before implementation. Finally for system implementation, a separate process engine with operation rules is obtained from the model. A key module of machine tool control system is used to illustrate the proposed method.

Keywords: Model-driven, Discrete Event Control, Machine Tool, StateChart

1 INTRODUCTION

Machine tool control systems have been playing an important role in the manufacturing industry over a number of decades. A typical machine tool control system usually consists of a mixture of continuous variable and discrete event control components. Continuous control components represent the real-time control laws of the system within each system work mode, such as motor position control, velocity control and torque control, etc, while the decision-making and switching between different work modes is represented by discrete event control. This paper mainly focuses on the latter.

Nowadays, rapidly changing technology, complicated applications and consumer preference are driving manufacturers to develop machine tool control systems with significantly reduced costs and shorter time-to-market. The traditional system development process suffers from either the difficulty in a relatively long “cycle and debug” stage to fix errors or modifying the original design to satisfy the new specification. The reasons are summarized as follows:

First, the traditional development method is typically case by case and strongly depends on designers’ experience. Whether the final design results reflect the original intention it depends on the designers’ understanding of the primitive specification in the textual format. The textual specification tends to be ambiguous, incomplete, and inconsistent [1]. It is almost impossible to show that system specification is accurate, complete, and internally consistent because there are no “natural language” compilers that can check the semantic content of the specification. Furthermore, with the functionality of modern machine tool systems becoming more complex, it is much harder for the designers to figure out the internal relation of the specification. Thus there could be more opportunities to introduce the errors. These errors cannot be found until a prototype system is built.

Secondly, the control system is always mixed with data and data processing related codes. It's not easy to find out which portion of the codes is related to the corresponding part of the system specification. If some modifications are required in the specification, the designers need to go through the entire code to locate the corresponding portion. Even a small modification may lead to a lot of bugs because the modified part is not consistent with the rest of the code, which makes the system debugging harder and more time-consuming. In a sense, for a new design, even if it has a quite similar specification compared to an existing one; most designers prefer starting from scratch to modifying the existing code [2].

In order to overcome these disadvantages, a new Model-Driven design and implementation method is proposed in this paper to enhance the development process of machine tool control system, in which: designers capture the system specification as much as possible and build an executable system model with some Computer-Aided Control System Design (CACSD) tools. The model has formal syntax and semantics, so that it can be simulated to eliminate the design errors before implementation. The control system derived from the model is divided into two portions: a generic process engine which implements the control algorithm and operation rules which are related to individual specification. The process engine is used to interpret the operation rules. Using this implementation scheme, only the operation rules need to be re-generated every time when the specification is modified while the engine remains unchanged.

The rest of this paper is organized as follows. Section 2 describes a typical machine tool control system structure. Section 3 discusses the StateChart-based Model-Driven discrete event control system design and implementation. Section 4 presents a case of a Mode

Control Supervisor design and implementation by using the proposed method. The paper concludes with section 5.

2 TYPICAL STRUCTURE OF MACHINE TOOL CONTROL SYSTEM

Figure 1 shows the typical structure of a machine tool control system which applies the foreground/background frame pattern mostly used in industry. The foreground is called real-time system (RTS), while the background is called free-time system (FTS). RTS involves two tasks: TIS and EIS which take care of time-based periodical interrupt service and event-based interrupt service respectively. Usually the PID algorithm of motor control, for example, is handled by TIS. Emergency stop is typically handled by EIS, etc. In the FTS, the system runs repeatedly along the sequence from DIG to OCS task. The external operation request and the internal system status will influence the task execution order. The complete execution sequence of the FTS is shown as follows:

1. The system starts from either ST (normal start-up) or RST (error recovery).
2. INZ task initializes the system, such as default parameter setup, work mode selection, etc.
3. DIG task checks system working status to judge whether the system has fault or not.
4. If the system has no fault, the control goes to SSC task, otherwise to ERH task.
5. If there is no fault after ERR, SSC task collects all the sensors and system status flags at this moment.
6. MCS determines at the current moment which task should be activated.
7. The control enters one of four kinds of work mode related tasks: ACS/MOS/PPS/PGS.

8. OCS task refreshes the entire system indicators and system status flags.
9. The control goes back to DIG task.

Figure 1: Typical structure of machine tool control system

Mode Control Supervisor (MCS) is a key module within the entire FTS. It accepts the operator's request such as the push of a button, or the variation of a signal from a sensor such as the triggering of a switch, and then based on current system status, decides in which mode the system should work and what operations should be performed. The most important aspect of MCS is that the same input may cause different outputs based on the current system status. So the MCS cannot be described in terms of a function, which simply maps input request into output. This aspect makes the design and implementation a tough job.

As mentioned earlier in section 1, the traditional MCS design starts with analyzing the functionality of a machine tool to get the textual specifications. Then the designers directly implement these specifications based on their personal experiences, which results

in software codes with a large number of convoluted conditional branches (like IF-ELSE or SWITCH-CASE statements in C language). The disadvantages of this method have been discussed before.

3 PROPOSED MODEL-DRIVEN DESIGN AND IMPLEMENTATION METHOD

Figure 2 shows the flowchart of the proposed Model-Driven design and implementation method. The first step is requirement analysis. Then an executable model is built. Several modeling methods are available. StateChart is chosen in this paper. The StateChart model is then checked by CACSD tools to get the error-free model. Next, the error-free model is translated into operation rules, which will be interpreted by an invariant process engine as implementation. Finally a prototype system is built to test the system performance. Within this flowchart, modeling, model check, and implementation are three key steps, and will be explained in detail in this section.

Figure 2: Flowchart of Model-Driven design and implementation method

StateChart modeling

The behavior of a discrete event control system depends entirely on the occurrence of asynchronous discrete events over time. For example, when the machine is working in the

automatic mode (state) and the emergency stop button is pushed (event), the machine will exit from the automatic mode (state) and enter emergency stop mode (state).

Finite State Machine (FSM) is the most popular modeling method for describing the behavior of discrete event systems because of its essentially event-driven nature [3]. A FSM model consists of a set of states, a set of transitions between states, and a set of actions associated with these states or transitions.

It should be pointed out that the FSM tends to be unmanageable as the modeling system becomes complex. The reason is that it doesn't support hierarchy and concurrency. Without support for hierarchy, a complex system will precipitate an explosion in the number of transition paths. At the same time, the lack of concurrency would cause an increase in the number of states. So the problem with such kind of FSM models is that once they reach several hundred transitions or states, they become incomprehensible to designers.

To overcome these shortcomings with FSM, Davis Harel from Weizmann Institute of Science, invented StateChart in 1987 [4]. Harel's StateChart is an extension of FSM by adding the following mechanisms: hierarchy and concurrency.

Hierarchy is represented by the XOR (exclusive OR) decomposition of states. A hierarchical superstate consists of a number of substates: being in a hierarchical superstate means being exactly in one of its substates. Figure 3 shows a comparison between FSM and StateChart in terms of hierarchy. The FSM on the left has three states *A*, *B*, *C*. The same transition *d* connects *AC* and *BC*. Two transition paths are needed to represent the transition *d*. The StateChart on the right introduces a hierarchical superstate *D* that contains *A* and *B* so that only one transition is needed.

Figure 3: FSM and StateChart hierarchy comparison

Concurrency is implemented by the AND decomposition of states. A concurrent superstate comprises a number of substates: being in a concurrent superstate means being in all its substates simultaneously. These substates are orthogonal since no transition is allowed between the substates of a concurrent superstate. On the right side of Figure 4, a StateChart concurrent superstate contains substate *A* and substate *D*. Meanwhile state *A* is a hierarchical superstate that is composed of substate *B* and substate *C*. Also state *D* is a hierarchical superstate has substates *E*, *F*, and *G*. There are totally $N=2+3=5$ atomic states in StateChart representation. However, the corresponding FSM on the left has to have $N=2 \times 3=6$ states to represent all the possible states of the system. As the states number increases, the difference will be dramatic.

Figure 4: FSM and StateChart concurrency comparison

In addition to hierarchy and concurrency, StateChart also introduces several features to facilitate the modeling of a complex discrete event system.

History condition allows a hierarchical state to remember its last active substate. This substate will be activated when the superstate becomes active again, overriding the default substate.

Compared to an action, an **activity** is an operation that takes time to complete. It is associated with a state. An activity may be a continuous operation or a sequential operation that terminates by itself after an interval of time.

Broadcasting communication means that every occurrence of any event is assumed to be broadcast throughout the system instantaneously. In particular, events generated internally during transitions are broadcast throughout the system, possibly triggering a new transition in other components, and giving rise to a whole chain of transitions.

Model check

The StateChart model check contains two steps: verification and validation. The term verification and validation (V &V) comes from software engineering [5]. During and after the implementation process, the software codes must be checked to ensure that it meets its specification and delivers the functionality expected by the users. Verification and validation is the name given to these checking processes.

3.2.1 Verification: Are we building the model correctly?

Textual specification tends to be ambiguous, incomplete and inconsistent. The model built directly from the textual specification probably has some kinds of logic flaws because of the misunderstanding of the specification. These flaws include conflicting transitions, unreachable transitions, unreachable states, and state dead ends as shown in Figure 5. For example, state *B* is a dead end which means that as transition *c* happens, the system will stay in *B* with no way to come out. The verification can be done with the aid of CACSD tools and it is quite straightforward.

Figure 5: Logic flaws of a model

3.2.2 Validation: Are we building the correct model?

Even if the model is logically correct; it may not behave as intended since the specification does not always reflect the real needs of the users. One way of the validation is through simulation. As mentioned before, the StateChart model has strict syntax and semantics so that a StateChart model is an executable model that can be simulated in many CACSD tools [6]. Simulation is to send designed events into the model and see whether the transitions, actions and activities resulting from the events accord with the system's desired functionality. For simple systems, the simulation is enough. However, as the systems become more and more complex, it is hard to generate event test vectors to go through all the transition paths in the model to make sure that the model satisfies some properties of the system such as safeness and activity.

Recently, a lot of research is being conducted to apply the formal methods in model check of Statechart model [7], [8], [9]. These methods are based on exhaustive state space exploration of the model of a FSM. Given an input in the FSM model and a property in temporal logic, a model checker (such as SMV and SPIN) determines whether the property holds in the model, and returns with a counterexample trace in case the property fails. All the methods translate the StateChart model into another modeling language which is the input of the model checker. These modeling languages do not exhibit the high

level features like hierarchy and history of StateChart. In the process of translation, these high level features are removed. For example, hierarchical StateChart is flattened leading to large state space. Also, there is no traceability between the front-end Statechart model and back-end input languages of the model checker, which is an important requirement, if one wants to debug the Statechart model based upon the checking report. Thus, it is concluded that currently simulation is more practical for StateChart model check; but the formal method is promising and should be the direction of future research.

Implementation

Several methods have been proposed to implement StateChart: Conditional Statements [10], StateTable [10], State pattern [11] and HSM pattern [12]. These methods have critical problems with the efficiency and readability of codes.

In our implementation scheme, the implemented software is divided into two parts: a generic processing engine and specific operation rules. Operation rules are converted from the StateChart model by a generator. The processing engine is an invariant program that can interpret these operation rules. By separating the engine and operation rules, complexity of the implementation doesn't simply disappear; instead, most of the complexity is taken care of by the processing engine. When a new specification is given, only the operation rules need to be re-generated.

3.3.1 Operation rules

All the information contained in a StateChart model can be represented by two data structures: one is a StateChart topology data structure; the other is a StateChart transition data structure.

Topology data structure

Figure 6 illustrates a simple StateChart example and its corresponding tree structure that reflects the hierarchy and concurrence defined by its topology. The correspondence between states and nodes of the tree is a one-to-one relation. In the tree, each state has its identifier which is a numerical ID of the state. Also each state has its direct descendant,

Figure 6: A simple StateChart example and its corresponding topology tree

```

Struct BSTATE{
    unsigned short    state,

    descendant,

    sibling,

    ancestor;

    char              status;

    ptrFunc            Entry, Exit;
};

```

Figure 7: Data structure of StateChart topology

```

Struct BTRANS{
    unsigned short from,

    event,

    to;

    int              (*guard)(void);

    ptrFunc            action;
};

```

Figure 8: Data structure of StateChart transition

sibling, and ancestor. For example, as shown in Figure 6, State *B*'s direct descendant, sibling, and direct ancestor are state *BA*, state *C*, and State *A*, respectively. According to this tree structure, the definition of data structure, which describes the topology of a StateChart model, is given in Figure 7. It stores the following information for each state: its identifier (state), direct descendant (descendant), next sibling (sibling), direct ancestor (ancestor), status information (status) and two pointers to functions to be called whenever the corresponding state is entered (Entry) or exited (Exit), respectively. The status information indicates the existence or nonexistence of a history condition defined at the corresponding state, and tells if the state is a concurrent or a hierarchical state.

Transition data structure

Similarly, the data structure of transition defines all the information of a transition as shown in Figure 8. It has two identifiers specified as the origin (from) and the destination state (to) of each transition, one trigger event identifier (event), one pointer to a function which returns a Boolean value indicating whether the transition can take place (guard) and one pointer to a function which represents an action associated with the transition (action).

3.3.2 Processing engine

The processing engine interprets the data structures that describe the topology and transition of a particular StateChart model. StateChart extends FSM by hierarchy and concurrency to make the modeling easier. However, it also makes the implementation of the engine more difficult. Consider, for example, the origin state of a transition being one of the concurrent states. Because the concurrent states need to be active at the same time, once the origin state is deactivated, so are its concurrent states.

Figure 9: Processing flow of the engine

The flowchart in Figure 9 illustrates the processing flow of the engine. There are five steps taken in sequence in order to finish one transition:

1. The event is sent to the engine.
2. The engine looks up the transition data structure to find out all the possible transitions triggered by the event. Then the status of each state in the topology data structure is checked to determine the current active states, only the active state can be the origin state of the fired transition. Once the origin state of the transition and the active state are matched, the transition is determined.
3. Once the transition is determined, the origin state is determined. However, if the origin state is a hierarchical or concurrent superstate, the active atomic state reachable from the origin state needs to be identified by the engine as the origin state first. Then the engine can also identify the nearest ancestor of the origin and destination state. After that, the engine deactivates all the states along the path from the active atomic state to the nearest common ancestor (excluding the latter) as well as all concurrent components along this path. During the process of deactivation, the engine calls the exit action functions.

4. The engine performs the actions associated with the transition.
5. All the states along the path from the nearest common ancestor to the destination state as well as the concurrent components are activated. Once the destination state is reached, the engine searches the history condition until an atomic state is finally reached. During the process of activation, the engine calls the entry action functions.

4 CASE STUDY

In order to evaluate the proposed method, the development of a MCS for a Mori Seiki 5-axis machining center GV503 control system has been carried out as a case study. Stateflow® of Matlab®, as a popular CACSD tool is chosen for the MCS StateChart modeling and model check.

MCS StateChart modeling

The work modes of the GV503 are generally classified into four groups as shown in Table 1.

Work Mode Groups			
Auto	Manual	Program Edit	Parameter Setup
MEM MDI TAPE	Handle JOG RPD ZRN	Edit	Setup

Table 1: Work Mode Groups of the GV503

In each mode, the machine performs different operations. MCS is used to determine the mode in which the system should operate based on the system status and the operator's requests.

The MCS StateChart model consists of four levels that correspond to the hierarchy of MCS. Level 1 has only one state: MCS. In level 2, MCS is composed of three substates: Off, On, Emergency stop. In level 3, On state is further decomposed into four exclusive substates according to the work mode groups which are Automatic, Manual, Program Edit, and Parameter Setup. In level 4, Automatic has three exclusive substates: MEM (default automatic mode), MDI (manual data input), and TAPE (DNC mode) substates. In the same manner, Manual state has Handle, JOG (jogging motion), RPD (rapid traverse), and ZRN (home return) substates.

There are 13 events defined in this case. Each event represents a push of a button. For example, MEM_P means that the MEM mode button is pressed, JOG_P means the JOG mode button is pressed, etc. The transition between different modes is triggered by the push of a button. For example, when the machine is working in MEM mode and the MDI mode button is pushed, the machine will be changed to MDI mode. The model is built in Matlab® Stateflow® as shown in Figure 10.

Figure 10: MCS StateChart model

MCS model check

The verification of the model can be done with the aid of the debugging function of Stateflow®. For validation, a test bench is built in Matlab by constructing a virtual Human-Machine Interface (HMI). By pushing buttons on the virtual HMI, events are sent to the model as a simulation stimulus. The communication between the Stateflow® model and HMI is achieved through the Matlab® Simulink®. The simulation environment based on Matlab® is shown in Figure 11. The simulation results are then compared with the MCS design specifications. If there are some errors, the StateChart model can be modified and simulated again until the model is error-free.

design. The process starts with modifying the MCS StateChart model, followed by another simulation and finally generating an operation rules description file. Results show that much less design time, debugging and testing effort are required to finish this new design compared to the traditional method.

5 CONCLUSION AND FUTURE WORK

A Model-Driven design and implementation method is proposed for discrete event control of a machine tool control system in order to overcome the drawbacks of the traditional method. The flowchart of the proposed method has been given and illustrated in detail by using the development of a key module of a machine tool control system (Mode Control Supervisor) as an example.

```

BSTATE bInfo[ ] =
{/* State, Descend, Sibling, Ancestor, Status, Entry, Exit */}
{ MCS, Off, MCS, 0, _noHist, Show, Show }
{ Off, 0, Estop, MCS, _noHist, Show, Show }
{ Estop, 0, On, MCS, _noHist, Show, Show }
{ On, Auto, Off, MCS, _Hist, Show, Show }
{ Auto, MEM, Manual, On, _noHist, Show, Show }
.....
}

```

Figure 12: Topology table

```

BTRANS bTrans[ ] =
{/*Origin State, Event, Destination State, Guard, Action */}
{ Off, On_P, On, 0, 0 }
{ EStop, Reset_P, Estop, 0, Set_Error0 }
{ Estop, Off_P, Off, cond, 0 }
{ On, Off_P, Off, 0, 0 }
{ On, EStop_P, EStop, 0, Set_Error1 }
.....
};

```

Figure 13: Transition table

Following the flowchart, at the first stage, StateChart is chosen for the discrete event control system modeling. After verification and validation of the built model, an error-free StateChart model is obtained. The topology and transition data structures which contain all the information in the StateChart model have been defined, so that at the implementation stage, the operation rules can be generated from the StateChart model by applying the data structures. In the control software, a separated and invariant engine is used to interpret these operation rules of which structure has been presented. With the separation of the engine and operation rules in the control software, only the operation rules need to be re-generated as the design specification changes in the future. Also this modular implementation makes the system modification and debugging much easier, which has been indicated in the case study.

In the future, the continued research will focus on building a more complex and realistic application of discrete event control for machine tool control system. The automatic mapping from verified Statechart model to operation rules related data is also need to be studied. In addition, the possibility of applying this method to complex logic sequences in machine tool environment or manufacturing system will also be explored.

ACKNOWLEDGEMENT

The authors wish to express their sincere appreciation for the generous support from Mori Seiki, which makes this research possible.

REFERENCES

- [1] David Lightfoot, 1991, Formal specification using Z, Macmillan.

- [2] Georg Frey, Formal methods in PLC control demonstrated at a flexible manufacture line, American Control Conference 2002.
- [3] C. G. Cassandras and S. L. Lafortune, 1999, Introduction to Discrete Event Systems, Kluwer Academic Publishers.
- [4] David Harel, 1987, StateCharts: A Visual Formalism for Complex Systems, Science of Computer Programming, No.8.
- [5] Ian Sommerville, 2004, Software Engineering, Seventh Edition, Pearson Education
- [6] Shin-ichiro Ie, et al., 1996, An automated control logic generation method for plant sequential-control, Proceedings of the 1996 IEEE International Symposium on Computer-Aided Control System Design.
- [7] E. M. Clarke and W. Heinle, Modular translation of statecharts to SMV, Technical Report CMU-CS-00-XXX, Carnegie-Mellon University School of Computer Science.
- [8] Diego Latella, et al., 1999, Automatic verification of a behavioural subset of UML statechart diagrams using the SPIN model-checker. Formal Aspects of Computing, 11(6):637–664.
- [9] S. Seshia, et al., 1999, A translation of Statecharts to Esterel. In *Proceedings of the 1st World Congress on Formal Methods (FM'99)*.
- [10] B.P. Douglass, 1998, Real Time UML – Developing efficient objects for embedded systems, Massachusetts: Addison- Wesley.
- [11] E. Gamma, R. Helm, R. Johnson and J. Vlissides, 1995, Design patterns: elements of reusable object-oriented software, Massachusetts: Addison Wesley.
- [12] M. Samek, P.Montgomery, 2000, State-oriented programming, Embedded Systems Programming, (8): 22-43.