Combining Statecharts and Z for the Design of Safety-Critical Control Systems

Matthias Weber

Technische Universität Berlin
we@cs.tu-berlin.de

Abstract. In this report, we describe an approach that integrates a mathematical specification language with more traditional software design techniques to yield a practicable methodology for the specification of safety-critical control systems. To manage complexity and to foster separation of concerns, the system design model is divided into three views: the architectural view, specified with object and class diagrams; the reactive view, specified with statecharts; and the functional view, specified with Z. A systematic relationship between the reactive and the functional view entails proof obligations to guarantee semantic compatibility. We illustrate this approach with a case study on controlling a heavy hydraulic press.

1 Introduction

Formal methods have been seriously applied during the past years in various industrial and academic pilot projects as reported, for instance, in [3]. However, the breakthrough has not yet been achieved. Many companies involved in such projects are scaling down their use of formal methods to a level that is in accordance with their current industrial relevance. For instance, they have only small teams of highly trained research staff working on selected critical aspects of systems.

What are the reasons for the failure of formal methods to achieve broader acceptance? From our own experience and from our analysis of experience reports ([3, 8, 12], for instance), we believe that one major reason is that presently formal methods come with too broad a goal.. Often, they aim at a superior and uncompromising methodological framework for the development of perfectly correct systems. They often presuppose idealized circumstances, and they have usually been developed in academic environments where such circumstances can be guaranteed. Also, such a monolithic approach does not leave much room for coexistence and interaction with other methodologies that are in standard use within an industrial development context. Still, research on such methods is necessary and has provided us with many useful techniques and results, but it is highly unlikely to lead to methods that will be quickly accepted in practice.

We believe that a more modest approach to the integration of formal techniques into the system design process will lead to a more immediate application of such techniques [7]. Starting out from existing and accepted conventional design methods which are amenable to the integration of mathematical techniques,

one should investigate at which points during the design process mathematical techniques can be smoothly and usefully integrated. The rationale for the use of formal techniques at these points should be convincing to the experienced engineer. Once experiments and case studies have provided evidence that the formal elements introduced are accepted, one can start to investigate further possible anchor points for mathematical techniques. This investigation can then be based on the experience gained during the first phase and on the evolving formal literacy of the design team. Hence, in principle, by iterating this process, one obtains a method that has more and more formal elements. It is important to note, that we do not attempt to embed conventional techniques into a formal method but rather the other way around.

In this report, we sketch an approach that sets out to integrate a mathematical specification technique into a well-known engineering technique in order to yield a practicable methodology for the specification of safety-critical control systems. Our starting point is the statechart notation, which is currently gaining acceptance in industry for the specification of embedded systems. To cope with the growing complexity and the safety requirements of these systems, we propose an integration of the specification language Z into statecharts, Z being used to model the data structures and data transformations within the system.

The idea of combining statecharts and Z is certainly not new; for example [1] uses a combination of Z and timed statecharts in the context of an application from avionics. The next section explains key ideas of our approach. The remaining sections illustrate the approach by developing a control system for a heavy hydraulic press.

2 Specification Methodology

A widely used technique in modern software engineering is to model a system by a combination of different – but semantically compatible – "views" of that system. The primary benefit of such an approach is to keep very complex systems manageable and to detect misconceptions or inconsistencies at an early stage. In the approach presented here, we divide the modeling into three views: the architectural model of the system, the reactive model of the system, and the functional model of the system (Figure 1).

The *architectural model* of a system describes the relationships between the types of components used in the system as well as the actual configuration of the system components itself. For the description of this model, we adopt the object-oriented modeling paradigm [2, for instance]: We understand an embedded control system as a hierarchically structured collection of objects that change state and interact with each other throughout their lifetime. The relationships between object classes are described using well-known elements of class diagrams, i.e. diagrams displaying classes and their structural relationships, such as aggregation and inheritance.

The two other views are primarily concerned with the specification of the behavior of single components of the embedded control system. We make a

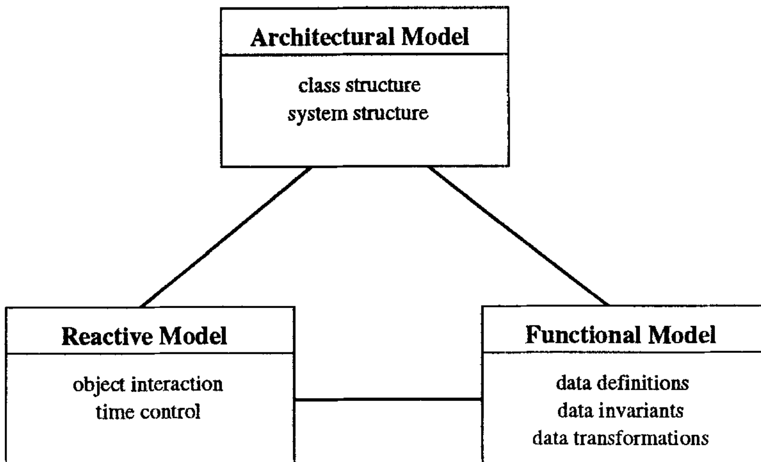

Fig. 1. The three modeling views of an embedded system

fundamental distinction with respect to the behavior of system components. The *functional model* of a component comprises data definitions, data invariants, and data transformation relations, in particular, for any component, its encompasses its local state and the input/output relation of its operations. Constraints, e.g. related to safety properties, about the components states can be derived based on these descriptions. The *reactive model* comprises the life-cycle of components, i.e. interactions with other components and the control of time during these interactions. Reactive behavior is modeled by specifying how, and under which timing constraints, operations from external objects are requested or supplied (or both) in the state changes of objects.

We specify reactive behavior using an appropriate variant of timed hierarchical state transition diagrams, i.e. with a variant of statecharts [5]. There are two reasons for this choice: firstly, statecharts have proven to be sufficiently expressive for modeling complex component interactions and time control, and secondly, the use of statecharts, or close variants of statecharts, is currently spreading in industry. This also enables us to use existing analysis and simulation tools for this notation.

Often, functional behavior in state-based systems is specified by textual or formal descriptions of pre- and postconditions and of data invariants. In our approach, we specify the functional behavior of objects using the state-based formal specification language Z [13]. There are two main reasons for using Z: firstly, in our view, Z has proven to be particularly useful for modeling complex functional data transformations; and secondly, both in academia and industry, Z has become one of the most widely used formal specification notations. Since we aim at a practical approach when modeling functionality, we try to stick to a constructive subset of Z, i.e. a subset that can be compiled into efficient code, whenever this is reasonable in a particular application. The use of a math-

ematical notation for modeling functional behavior enables us to prove abstract safety properties about the control system, such as provisions that the system may never enter certain hazardous states. Safety conditions imposed on data structures and data relationships should, of course, be specified using the full expressive power of the Z language.

Note that we are *not* arguing in favor of a monolithically formal approach. Rather, our goal is to systematically embed mathematical elements into industrially used engineering techniques. As will be seen, this leads to an approach some parts of which are "hard", i.e. fully precise, while others remain "softer", i.e. allow for a certain range of interpretations. In our view, such an approach leaves more room to be adapted to the actual circumstances in particular industrial application contexts.

3 The Case Study: Control of a Hydraulic Press

We consider a simple embedded control system, a controller for a heavy hydraulic press that is operated manually. Hydraulic presses are devices for pressing workpieces into a certain shape. The human operator, at the press, places the workpiece in the press and initiates the closing of the press. The plunger of the press moves down, presses the workpiece and subsequently moves up again. The workpiece can then be removed from the press and the entire process may be repeated.

Hydraulic presses are dangerous, since the worker operating the press may hurt himself by accidentally trapping his hand in the press. A typical safety device to prevent hand injuries are *two-hand controllers*, i.e. control units with two buttons, located about 1 meter apart, that must both be kept pressed while a potentially dangerous action is performed [4]. In addition, both buttons must be pressed within a small period of time (in our example 0.5 sec) in order to successfully initiate the closing of the press. The obvious intention behind two-hand controllers is to keep both of the worker's hands out of the danger area. If a button is released while the press is closing, the press will immediately stop and reopen. However, after a certain point is reached, which we call the *critical point*, the closing press can no longer be stopped physically, and hence cannot react to the release of a button. Finally, for reasons of reliability, the system should be capable of detecting sensor readings that are incompatible with the physical properties of the press. In such a situation, which might be due to a broken sensor or a failure in message transmission, the system should immediately stop the press.

This very simple embedded system is a good example to introduce and explain our approach, since it comes with interesting safety and real-time constraints, but is simple enough to not clutter the presentation with technical details. It should be obvious that the above informal specification is far too sketchy to adequately specify the required system behavior.

4 Architectural View

In the previous section, we have presented the informal requirements of the hydraulic press control case study. Since this is a very small example, the analysis and architectural design is straightforward. The results are summarized in the diagrams presented in Figures 2 and 4. For this example, we mostly use notations inspired from OMT [10] and Booch [2]. However, choice of notations is by no means essential and it should not be difficult for the experienced to adapt the information content of the following diagrams to his favorite notation.

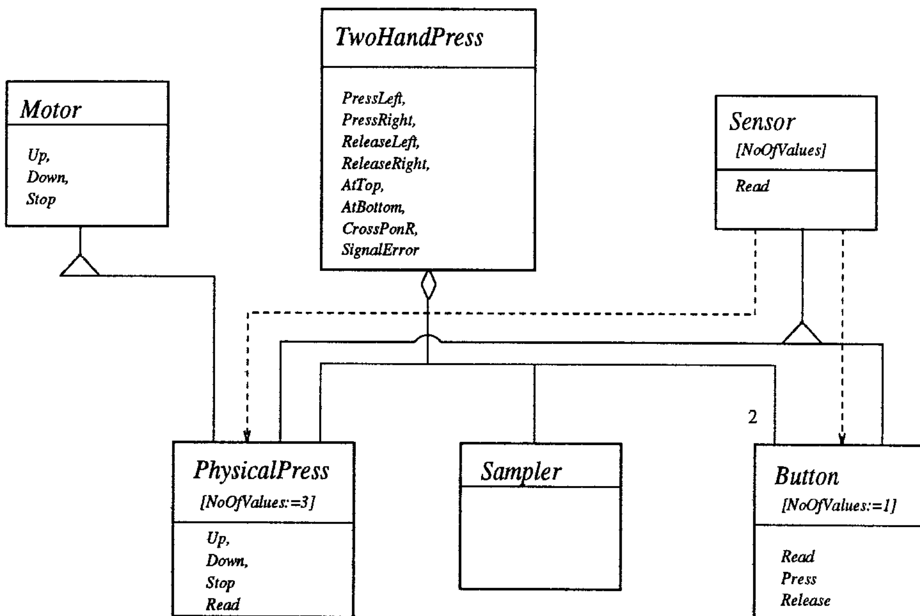

Fig. 2. Class diagram of the press system

The class diagram describes a two-hand press-object as consisting of four subobjects: two buttons to control the press, the physical press, and a data sampler. Aggregation is denoted by links adorned with a rhomb. Multiplicities can be specified explicitly along aggregation links. A button is a particular instance of a sensor. It offers an operation to read its (only) measured value. This value indicates whether the button is currently pressed. The parameter instantiation relationship is denoted by dashed arrows. In addition to being an instantiation, a button is also a specialization of a sensor, because it incorporates additional operations for pressing and releasing a button. The specialization (or inheritance) relationship is denoted by links adorned with a triangle. In the context of this case study, the physical press is modeled as an entity specializing both a sensor

and a motor. In particular, besides an operation to read the current state of the press, it includes operations to move up, down and to stop. The physical press is also an instantiation of a sensor measuring three values. These values are further described below.

In Figure 2, the sampler and the control of the subobjects of the two-hand press together constitute the software part of the system. The press and the buttons model physical objects, connected to the control by communication lines. From a more traditional, software-centered point of view they would be represented as the environment of the software control component.

Since this is a rather simple system and it has a severe real-time requirement, our main architectural decision is to adopt a time-frame approach to specify its behavior (see Figure 3).

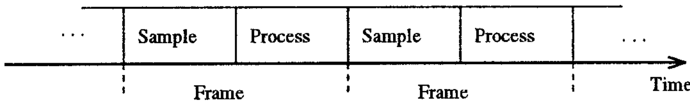

Fig. 3. Time-Frame Processing

More specifically, the idea is to let the sampler periodically read the current values measured by the physical press and the buttons and then, based on these values, to send control messages to the press control itself. The sending of these messages can be interpreted like events affecting control. The press control processes these messages and converts them into motor commands to move the press. In this sense the purpose of the sampler is to abstract from the low-level details of communication with the external devices and to offer an appropriate interface to the logical view of the controller. Of course, we must be concerned that the control does not miss a relevant input, i.e. the maximum time for the control to react to an input must be less than the length of sampling interval. The controller requests the operation of individual buttons using natural number indices, e.g. *Button*[1].*Read* reads values from the first button.

The communication relationships between objects of this system are displayed in Figure 4.

Of course, there are many alternative approaches to this specific one, for example the two sensors could themselves be active processes interrupting the control by signaling events to it. However, the advantage of time-frame based processing is that we can more rigorously control the order of events. Furthermore, given the small number of sensors in this case study a concurrent solution would not be very realistic. In this example, all communications links denote synchronous communication. Asynchronous communication can be indicated by appropriate adornment of communication links.

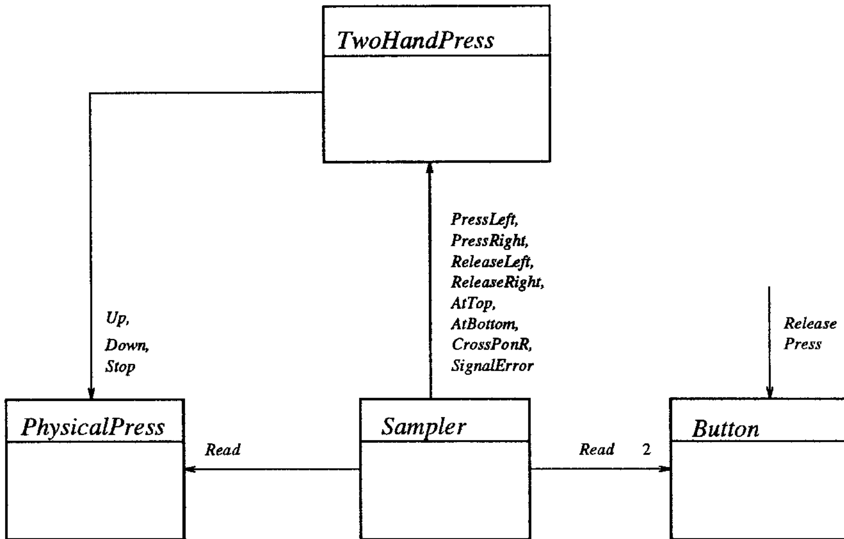

Fig. 4. Communication links of the press system

5 Reactive View

The top-level reactive behavior of the press control is described by the statechart in Figure 5. Initially, the control remains idle until the sampler signals that the

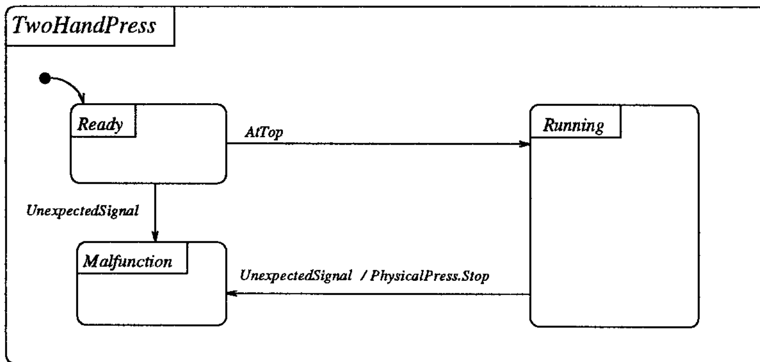

Fig. 5. Top-level reactive behavior of the controller

press is in default position, i.e. at the top. The control then enters the running mode. In case of a malfunction, the motor is stopped and a special error state

is entered. A malfunction is recognized if the sensors deliver values that are not expected at any point of operation. *UnexpectedSignals* is an abbreviation for a group of transitions. We return to its definition below.

Following common conventions, we denote states by rounded boxes and indicate their names on the upper left corner. As usual, we use a dot-anchored kind of arrow to point to default substates to be entered when entering a complex state. In general we use two kinds of transitions, *operation transitions* and *timeout transitions*.

The arrows for operation transitions are in general adorned as follows:

ProvidedOperations[Condition]/RequestedOperations

If the object is in the source state and one of the indicated provided operations, separated by **or**, is requested from an external object, then, if the condition is satisfied, the indicated operations are requested from the indicated external objects and the object changes into the target state of the arrow. The condition is optional, an omitted condition acts as a condition that is always true. Requested operations are optional too: if no requested operations are indicated, the object just performs a change of the internal object state. The other form of transitions, the timeout transition, is explained below.

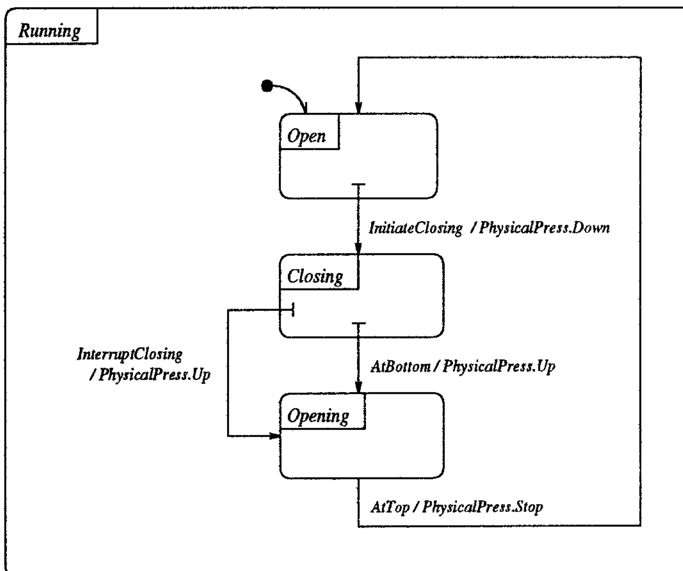

Fig. 6. Refinement of the running state

The running state is further refined in Figure 6. The press is operated in a continuous cycle of closing and opening. Entering the state *Closing* is associated

to a motor command to move down. The open state and the label *InitiateClosing* are further refined below. The closing state may be left by either releasing one of the buttons, or by reaching the bottom of the press. Both cases lead to a motor command to move up. Opening then continues until the sampler signals the press being again at the top. Following a common convention about state diagrams, we use stubbed arrows to indicate transitions originating from substates of not yet sufficiently refined states.

The behavior of the control in the opening and closing states has not yet been refined to sufficient detail. First, we have to distinguish between those states in which the closing press above or below the *critical point*, i.e. the point below which the press can no longer be reopened before closing. This is clarified in the state diagram in Figure 7. The two arrows leaving the refined closing state

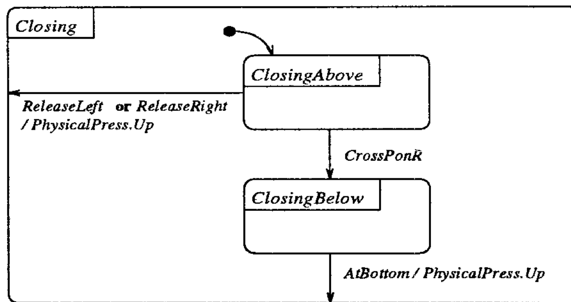

Fig. 7. Refinement of the closing state

correspond to the two arrows leaving the respective unrefined state in Figure 6. Identification of such arrows should be unambiguous by graphical position and by label.

At this point, we have sufficiently exposed the state structure of the two-hand press, to define precisely the transition group labeled *UnexpectedSignals* in Figure 5.

$$\begin{aligned} \mathit{UnexpectedSignals} \equiv\ & \mathit{AtTop}[\mathit{ClosingBelow}] \\ & \textbf{or } \mathit{AtBottom}[\mathit{Ready} \lor \mathit{Open} \lor \mathit{ClosingAbove}] \\ & \textbf{or } \mathit{CrossPonR}[\mathit{Ready} \lor \mathit{Open}] \\ & \textbf{or } \mathit{SignalError} \end{aligned}$$

The most complex aspect of the press behavior is obviously the transition from the open to the closing state. This is described in detail in the state diagram in Figure 8 According to the logic of the two-hand press, in order to initiate the closing of the press, the two buttons have both to be released and subsequently both to be pressed within a specific time interval (*MaxDelay* milliseconds). Therefore, the safety state, which the system enters initially, can be left only when both buttons are released. Now when, e.g. the left button

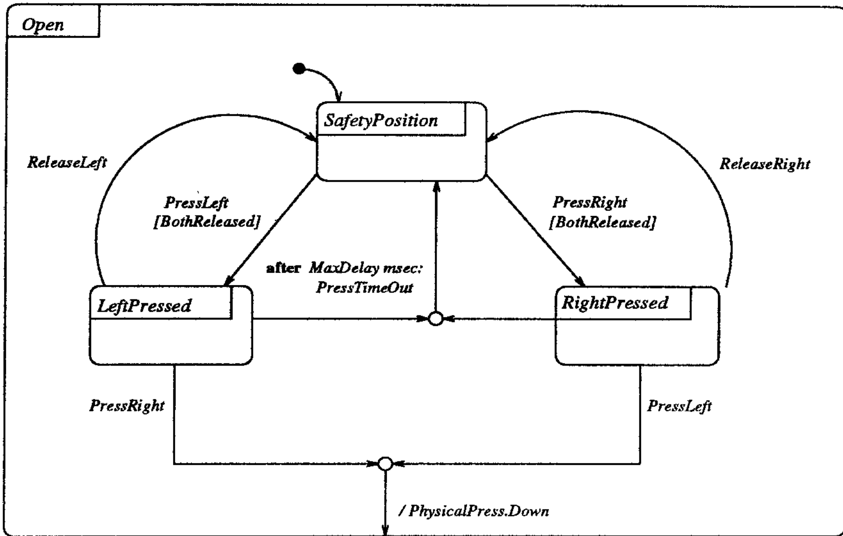

Fig. 8. Refinement of the open state

is pressed after both buttons were released, the right button must be pressed within a certain time interval, *MaxDelay* milliseconds, otherwise a timeout occurs and the system re-enters the safety state. If the right button is pressed soon enough, the system requests the motor to move the press down and enters the closing state.

The transition groups labeled *InitiateClosing* and *InterruptClosing* in Figure 6 can now be defined as follows:

$$\begin{aligned} \mathit{InitiateClosing} &\equiv \mathit{PressLeft}\,\textbf{or}\ \mathit{PressRight} \\ \mathit{InterruptClosing} &\equiv \mathit{ReleaseLeft}\ \textbf{or}\ \mathit{ReleaseRight} \end{aligned}$$

This example has made use of the second kind of transition, the timeout transition. The respective arrows are adorned as follows:

$$\textbf{after}\ \mathit{TimeExpression} : \mathit{InternalEvent} / \mathit{RequestedOperations}$$

If the system has been in the source state of such an arrow for the time specified in the time expression, then it requests operations from other objects and changes into the target state of the arrow. As for operation arrows, the condition and requested operations may be omitted. Timeout transitions are a very simple, but often sufficient, means to deal with time constraints. If necessary, they could be generalized to timed transitions [9].

After modeling the reactive view of the global control of the hydraulic press system, we have yet to describe the reactive behavior of the sampler: After initialization, the sampler periodically samples the two button sensors and the press sensor. For each sensor, the current values of its signals are read, and,

depending on the values of these signals, a certain operation from the controller is requested. We do not detail the corresponding state diagrams at this point. The state diagrams for the motor, the buttons, and the physical press are not part of the controller but part of its environment. Since their discussion does not add anything interesting at this point, their treatment is not further detailed.

A variety of formal semantics for statecharts have been developed [14]. The present paper is more in the line of current work on embedding statecharts into an object-oriented setting [11] [6]. Therefore, we would like to add two remarks about basic semantic concepts of the statechart notation as used in this report:

- The basic communication mechanism is point-to-point communication rather than broadcasting. Requesting an operation from an object can be interpreted as sending a message to an object, and providing an operation to an object can be interpreted as receiving a message from an object. As specified in the architectural view, communications can be synchronous or asynchronous. Following the approach in [11], operation transitions are thus based on the concepts of request and provision of operations rather than the concept of event.
- The execution of a transition is not timeless and external messages may arrive at any time. As a consequence, the system may not be able to immediately react to a message. Therefore, incoming messages must be queued and then worked off individually. By convention, if there is no transition for a particular message, then the system does not change state.

Further experience with case studies should guide the evolution of the notations and the semantics assumed here.

6 Functional View

Following common practice when presenting Z specifications, we first specify the state space of the hydraulic press control and then the effect of its operations on this space. The internal state space is essentially made up of appropriate *internal models* of the physical components. These models contain all information necessary for the control to decide on which action to take. In order to avoid naming confusion, we introduce a systematic naming convention: The internal model of a physical unit U is named *UModel*.

Press controller: State

First, we define the states of the button control. A button is an object that can be pressed or released.

$Button ::= pressed \mid released$

Remember, that the requirements of the press control described situations in which both buttons must be released first before they may be pressed again to initiate closing of the press. To model this information, we use the following set:

$DoubleRelease ::= required \mid notrequired$

We do not explicitly mirror the full substate structure of the press control from the reactive view, e.g. the various substates of open. Rather, in this functional view, we find it more convenient to model the buttons explicitly and later define in terms of our Z model the states of the state diagrams.

ButtonModel

$left, right : Button$

$release : DoubleRelease$

$(left = released \land right = released) \Rightarrow release = notrequired$

This schema describes the button model as consisting of the two buttons and a flag that indicates whether a release of both buttons is required. A logical constraint allows a release to be required only if at least one of the buttons is pressed.

We introduce an auxiliary schema for describing those situations in which the press is correctly triggered to start moving, i.e. both buttons have been pressed within the permitted delay after both have been previously released.

PressTriggered

$ButtonModel$

$left = pressed$

$right = pressed$

$release = notrequired$

Note, that in the functional view, we do not model real-time aspects, rather, these aspects are delegated to the reactive view.

Next, we define the press states. The press, without the buttons, may be ready, open, closing above or below the point of no return, opening, or in state of error.

$PressState ::= ready \mid open \mid closingabove \mid closingbelow \mid opening \mid error$

The internal model of the press is defined by:

PressModel

$press : PressState$

By means of the notions introduced so far, we can now specify the state of the press control as follows:

```
┌─ TwoHandPress ─────────────────────────────
│ PressModel
│ ButtonModel
├──────────
│ PressTriggered ⇒ press ≠ open
│ press = closingabove ⇒ PressTriggered
└────────────────────────────────────────────
```

This schema describes the press control as consisting of the press state and the button control all being subject to two constraining conditions related to functionality and safety: The first condition states that the press can be open only if it has not been triggered. The second condition states that above the critical point the press can be closing only if it has been triggered. These conditions must be satisfied for any state of the system

Note that the functional specification of the state space reexpresses information that is present in the state structure of the reactive view. For example, the definition of *PressState* is closely related, but not quite identical, to the states used in the reactive view. for example the state *Ready* can be defined by the following schema[1]

```
┌─ Ready ────────────────────────────────────
│ TwoHandPress
├──────────
│ press = ready
└────────────────────────────────────────────
```

In general, our primary intention is to specify each view, so that it makes maximal sense by itself, e.g., in case of the functional view, we are interested in specifying clear and crisp data invariants. As in this example, this may well lead to redundancies. If desired, redundancy can be avoided by allowing, within the functional model, the use of states and operations *derived* from the reactive model. The development of a notation for such *derived functional models* is subject of current work.

Press controller: Operations

We now turn to the specification of the operations of the press controller. First, we specify the effect of pressing the left button. Local to the button model, the effect of this operation can be specified as follows.

[1] The relation between the reactive and the functional view are discussed in detail the next section.

PressLeftLocal

$$
\begin{array}{|l}
\Delta ButtonModel \\
\hline
left = released \\
left' = pressed \\
right' = right \\
release' = release
\end{array}
$$

This operation can be extended to the two-hand press state by specifying how the press state is affected by the pressing of the left button. There are two cases. If the right button has already been pressed and no release is required yet, then the press begins to close. If this is not the case, the press remains open.

PressLeft

$$
\begin{array}{|l}
\Delta PressModel \\
PressLeftLocal \\
\hline
(press = open \land right = pressed \land release = notrequired) \\
\quad \Rightarrow press' = closingabove \\
(press \neq open \lor right = released \lor release = required) \\
\quad \Rightarrow press' = press
\end{array}
$$

This specification captures very succinctly the normal behavior of the operation to press the left button. The effect of pressing the right button can be specified analogously.

Next, we turn to the release operations. Again, we begin by specifying the effect of releasing the left button local to the button control.

ReleaseLeftLocal

$$
\begin{array}{|l}
\Delta ButtonModel \\
\hline
left = pressed \\
left' = released \\
right' = right \\
right = released \Rightarrow release' = notrequired \\
right = pressed \Rightarrow release' = required
\end{array}
$$

Note, that the release of a button may affect the release flag. Next, we extend this operation to the state of the two-hand press. The interesting case here is to capture the effect of releasing a button at a time when the press is closing and still above the point of no return.

$$
\begin{array}{|l}
\hline
\textit{ReleaseLeft} \\
\Delta \textit{PressModel} \\
\textit{ReleaseLeftLocal} \\
\hline
\textit{press} = \textit{closingabove} \Rightarrow \textit{press}' = \textit{opening} \\
\textit{press} \neq \textit{closingabove} \Rightarrow \textit{press}' = \textit{press} \\
\hline
\end{array}
$$

Analogously, we can specify the operation to release the right button.

After specifying the button operations, we now turn to the operations describing state changes resulting from signals received from the physical press. For example, the effect of the press indicating arrival at the top of the press can be specified as follows:

$$
\begin{array}{|l}
\hline
\textit{AtTop} \\
\Delta \textit{TwoHandPress} \\
\hline
\textit{press} \in \{\textit{opening}, \textit{ready}\} \\
\qquad \Rightarrow \textit{press}' = \textit{open} \\
\textit{press} \in \{\textit{opening}, \textit{ready}\} \land (\textit{left} = \textit{pressed} \lor \textit{right} = \textit{pressed}) \\
\qquad \Rightarrow \textit{release}' = \textit{required} \\
\textit{press} \in \{\textit{closingabove}, \textit{closingbelow}\} \\
\qquad \Rightarrow (\textit{press}' = \textit{error} \land \textit{release}' = \textit{release}) \\
\textit{left}' = \textit{left} \\
\textit{right}' = \textit{right} \\
\hline
\end{array}
$$

The first implication specifies the normal behavior, i.e. the signal is arriving during initialization or opening of the press. Note, in this case, the change of the release flag, i.e. after a arriving at the top, a full release of both buttons is required. The second implication specifies the abnormal behavior, i.e. the signal is arriving during closing of the press, in which case the press stops the motor and goes into the error state. The remaining operations *CrossPonR*, *AtBottom*, and *SignalError*. can be specified in a similar style.

Press controller: conditions

The condition that both buttons are released can be defined as follows:

$$
\begin{array}{|l}
\hline
\textit{BothReleased} \\
\textit{TwoHandPress} \\
\hline
\textit{left} = \textit{released} \\
\textit{right} = \textit{released} \\
\hline
\end{array}
$$

Press controller: internal events

Finally, we specify the sole internal event that arises in case the press is open, either one of the buttons was pressed, but the delay for pressing the other button has been exceeded. In this case, the event changes the system back into its safety position.

PressTimeOut

$\Delta TwoHandPress$

$press' = press = open$

$left = pressed \Leftrightarrow right = released$

$release = notrequired$

$release' = required$

$left' = left$

$right' = right$

This completes the functional view of the control. At this point, the reader may argue that this functional view of the system is redundant, since all behavioral aspects of this finite state system could have been adequately specified using statecharts alone. We would argue here that the functional view is useful in its own since it shows in a very explicit way that the internal models of the physical components satisfy important safety conditions. Admittedly, one could have expressed all details of the "button logic" with statecharts, but this would have definitely obscured the specification and the proof of its properties. Furthermore, this is a very small example, and, in our experience, the data space and the amount of data transformation tends to grow quickly in more complex control systems.

7 Consistency

The reactive and functional view of an embedded system can be checked against each other in many interesting ways: The basic idea is to systematically and consistently relate the state hierarchy and the transitions introduced in the statecharts with the state spaces and operations as defined by the Z schemas.

7.1 Relating states

A straightforward way to relate states between the two different views is to map every state diagram state S into an appropriate Z schema S_z describing this state, and then to formulate various proof obligations for this mapping to be adequate.

Assuming as given such a mapping for a particular component, the consistency conditions can be presented in three steps. For an arbitrary state S from

the reactive model of this component, we distinguish between the following two cases:

- S is an elementary state, i.e. there is no decomposition of S in the reactive model. In this case, one has to verify that the associated Z state S_z is nonempty, i.e.
  Consistency: $\vdash \exists S_z$.
- S is a hierarchically composed state, i.e. in the reactive model S is decomposed into exclusive sub-states S_1, S_2, $\cdots$, and S_n $(n > 0)$ with associated Z-schemas S_z, S_{1z}, S_{2z}, $\cdots$, and S_{nz}. In this case, one has to check sufficiency, necessity, and disjointness of the decomposition.

  Sufficiency: $S_{1z} \vee S_{2z} \vee \cdots \vee S_{nz} \vdash S_z$.
  Necessity: $S_z \vdash S_{1z} \vee S_{2z} \vee \cdots \vee S_{nz}$
  Disjointness: $S_z \vdash \neg (S_{iz} \wedge S_{jz})$ for all $i, j \in \{1, \cdots, n\}$, where $i \neq j$.

Of course, the top-level statechart of a component must be related to the Z schema defining the full state space of the component.

7.2 Hydraulic press example

In case of the hydraulic press, the states from the state interaction diagrams can be defined in terms of the Z-model quite easily. We illustrate this for the substates of the open press (see Figure 8).

```
┌─ SafetyPosition ─────────────────────────────────
│ TwoHandPress
├──────────────
│ press = open
│ left = pressed ∨ right = pressed ⇒ release = required
└──────────────────────────────────────────────────
```

The second condition states that in the safety position, if any button is pressed, a release is required before the press may begin to close.

The substate *RightPressed* can be defined as follows.

```
┌─ RightPressed ───────────────────────────────────
│ TwoHandPress
├──────────────
│ press = open
│ left = released
│ right = pressed
│ release = notrequired
└──────────────────────────────────────────────────
```

The substate *LeftPressed* can be defined analogously. The composed state *Open* can be defined as follows.

Open
TwoHandPress
---
press = *open*

To ensure consistency between these definitions, we have to prove necessity of the OR-composition:

$Open \vdash (SafetyPosition \vee LeftPressed \vee RightPressed)$

Sufficiency and disjointness can be shown in a similar way. Similar definitions and consistency proofs can be given for the other states. The reader might object at this point, that one may always define composed states in such a way as to automatically satisfy the completeness proof obligation. While we admit that this is possible, we want to stress at this point, that our methodological guideline is to define composed states as naturally as possible from different points of view. In some cases, consistency between views may follow by construction, in others, e.g. the *Open* state, consistency must be ensured by a separate nontrivial reasoning.

7.3 Relating operations

In the functional view, we have defined a Z schema for each service, internal event, or guard in the statechart. Based on the association of a Z schema to each statechart box one can verify conformance between the statechart transitions and the Z definitions.

The idea is to consider an arbitrary state and an arbitrary operation and then to check for consistency with respect to the transitions leaving that state. More precisely, given an arbitrary operation *Op* and state S, we have to prove that each transition leaving S and labeled with *Op*, and possibly some condition, behaves as expected, i.e. results in the desired state. We furthermore have to prove, that if the operation or event *Op* occurs and neither one of the conditions of those transitions are true, the application of *Op* preserves this state.

First, we distinguish the case that no transitions labeled with *Op* are leaving S. In such a case, we have to show that application of S preserves this state.

Preservation: $S_z \wedge Op_z \vdash S'_z$.

S_z and Op_z are the Z schemata associated to S and *Op*.

It remains to deal with the case that the transitions $t_1, \cdots, t_n$ $(n > 0)$ are labeled with *Op* and guards $C_1, \cdots, C_n$ and move from S to states $S_1, \cdots, S_n$. We check for consistency of these transitions as follows:

Applicability: $S_z \vdash \text{pre } Op_z$.
Explicit Correctness: $S_z \wedge Op_z \wedge C_{iz} \vdash S'_{iz}$, for $1 \leq i \leq n$.

Implicit Correctness: $S_z \wedge Op_z \wedge \neg (C_{1z} \vee \cdots \vee C_{nz}) \Rightarrow S'_z$, if S_z is primitive.

C_{iz} and S_{iz} are the Z schemata associated with C_i and S_i. Note the applicability check, i.e. any state from which a transition labeled with Op is leaving must imply the precondition of Op. Note also, that implicit correctness has to be checked only for primitive states, as it induces implicit correctness for composed states.

Note that implicit correctness is trivial in those cases in which the disjunction of the guards is complete, for example in the frequent number of cases where $n = 1$ and $C_1 \Leftrightarrow true$.

7.4 Hydraulic press example

First, we consider the operation *PressLeft*. Apparently there are only two relevant transitions, giving rise to the obligations:

$SafetyPosition \wedge PressLeft \wedge BothReleased \vdash LeftPressed'$

$SafetyPosition \wedge PressLeft \wedge \neg\ BothReleased \vdash SafetyPosition'$

$RightPressed \wedge PressLeft \vdash ClosingAbove'$

Furthermore, the operation is inapplicable in two states only, namely:

$LeftPressed \vdash \neg\ \mathrm{pre}\ PressLeft$

$ClosingAbove \vdash \neg\ \mathrm{pre}\ PressLeft$

For the other primitive states, we have to prove preservation, e.g.:

$Opening \wedge PressLeft \vdash Opening'$

An orthogonal analysis can be done with the other press and release operations. Next, we turn to the control event *AtTop*. The transitions to be verified are:

$Ready \wedge AtTop \vdash SafetyPosition'$

$Opening \wedge AtTop \vdash SafetyPosition'$

$Running \wedge ClosingBelow \wedge AtTop \vdash Malfunction'$

Inapplicability is given in the states *Open* and *Error*. The other control events can be analyzed in a similar fashion.

Finally, there is one internal event *PressTimeOut*. The following transitions must be checked.

$LeftPressed \wedge PressTimeOut \vdash SafetyPosition'$

$RightPressed \wedge PressTimeOut \vdash SafetyPosition'$

Inapplicability is given in the the remaining states. All these properties amount to very simple checks of the given definitions. Nevertheless, checking these conditions is very helpful for debugging a specification.

8 Conclusions

The proposed combination of statecharts and Z for modeling embedded control systems proved to be both semantically and pragmatically interesting. It is important at this point, to conduct more experiments with the aim of identifying useful recommendations, guidelines, and heuristics for the process of developing such combined specifications. Parallel to that, tools for translating specifications into code should be developed or adapted. For statecharts, such tools are available. Concerning Z specifications, we would argue to stick to an operational modeling style, from which efficient code can be generated. This was straightforward in the hydraulic press example. The degree to which such a style can be reasonably adopted seems to depend on the particular application context.

References

1. L. M. Barroca, J. S. Fitzgerald, and L. Spencer. The architectural specification of an avionics subsystem. In *IEEE Workshop on Industrial-strength Formal Specification Techniques*, pages 17–29. IEEE Press, 1995.
2. G. Booch. *Object-Oriented Analysis and Design with Applications.* Benjamin Cummings, second edition, 1994.
3. D. Craigen, S. Gerhart, and T. Ralston. An international survey of industrial applications of formal methods. Technical Report NISTGCR 93/626, National Institute of Standards and Technology, Gaithersburg, MD 20899, 1993.
4. Zentralstelle für Unfallverhütung und Arbeitsmedizin. *Pressen – Sicherheitsregeln für Zweihandschaltungen an kraftbetriebenen Pressen der Metallbearbeitung.* Hauptverband der gewerblichen Berufsgenossenschaften, Langwartweg 103, 5300 Bonn 1, 2nd edition, 1978.
5. D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.
6. D. Harel and E. Gery. Executable Object-Modeling with Statecharts. In to appear, editor, *Proc. ICSE 18*, 1996.
7. M. Heisel, S. Jähnichen, M. Simons, and M. Weber. Embedding mathematical techniques into system engineering. In M. Wirsing, editor, *ICSE-17 Workshop on Formal Methods Application in Software Engineering Practice*, pages 53–60, 1995.
8. I. Houston and S. King. CICS Project Report: Experiences and Results from the Use of Z in IBM. In S.Prehn and W.J.Toetenel, editors, *VDM'91 Formal Software Development Methods*, volume 551 of *LNCS*, pages 588–596. Springer-Verlag, 1991.
9. Y. Kestens and A. Pnueli. *Timed and Hybrid Statecharts and their Textual Representation*, volume 299 of *LNCS*, pages 591 – 620. Springer-Verlag, 1992.
10. J. Rumbaugh et al. *Object-Oriented Modeling and Design.* Prentice-Hall, 1991.
11. B. Selic, G. Gullekson, and P. T. Ward. *Real-Time Object-Oriented Modeling.* John Wiley & Sons, 1994.
12. IEEE Software. *Safety-Critical Systems.* IEEE, January 1994.
13. M. Spivey. *The Z Notation, A Reference Manual.* Prentice Hall, 2nd edition, 1992.
14. M. von der Beeck. A comparison of statecharts variants. In *Symposium on Fault-Tolerant Computing*, LNCS. Springer, 1994.