A Pragmatic, Rigorous Integration of Structural and Behavioral Modeling Notations

Daniel M. Berry*
GMD-FIRST
12489 Berlin, Germany
dberry@cs.technion.ac.il

Matthias Weber†
Research and Technology
Daimler-Benz AG
10559 Berlin, Germany
weber@dbresearch-berlin.de

Abstract

This paper describes a pragmatic, rigorous integration of the mathematical specification language Z with well-known object modeling notations and an object-oriented variant of statecharts. The goal is to preserve the abstraction and flexibility of widely-used design notations while being able to embed the precision and rigor of mathematical specification at selected places. The integration between the notations is based on a mapping between entities of the three models.

1. Introduction

Formal methods have been seriously applied in recent years in various industrial and academic pilot projects as reported, for instance, in [4]. However, the breakthrough, by which formal methods are routinely applied to software developments, 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 ([4, 12], for instance), we believe that one major reason is that presently formal methods come with too broad a goal, of complete formalization.

We believe that a more modest approach of integrating formal techniques into the current system design process will lead to a more immediate application of such techniques. Starting out from existing and accepted conventional design methods, one should investigate at which points during the design process mathematical techniques can be smoothly and usefully integrated.

Due to space limitations, many formal details have been omitted. The interested reader can find them in a technical report of the same title as this paper available from either author [1].

2. Modeling and Integration

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. In the approach presented here, as is shown in Figure 1, 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, and we provide a mapping between those views.
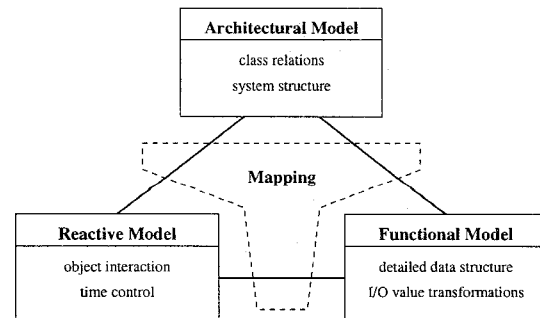

Figure 1. The three modeling views of a system

---

*regularly at Computer Science Department, Technion, Haifa 32000, Israel
†work done while at Fachbereich Informatik, Technische Universität Berlin, 10587 Berlin, Germany

38

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 a system as a hierarchically structured collection of objects that change state and interact with each other throughout their lifetime. In this setting, we use notation from OMT [11] and the Unified Modeling Language [14].

The two other views are primarily concerned with the behavioral specification. We make a 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. The *reactive model* comprises the life-cycle of components, i.e. interactions with other components and the control of time during these interactions.

We specify reactive behavior using an object-oriented variant of statecharts [7, 8], for two reasons: statecharts have proved to be sufficiently expressive for modeling complex component interactions and time control, and the use of statecharts is currently spreading in industry.

We specify the functional behavior of objects using the state-based formal specification language, Z [13], for two main reasons: Z has proved to be particularly useful for modeling complex functional data transformations, and both in academia and industry, Z has become a most widely used formal specification notation. Since we aim at a practical approach, we try to stick to a constructive subset of Z, i.e. a subset that can be compiled into efficient code. The use of a mathematical notation, such as Z, for modeling functional behavior enables us to prove many safety properties about the control system, such as provisions that the system never enters certain hazardous states. Safety conditions imposed on data structures and data relationships could, of course, be specified using the full expressive power of the Z language.

The *mapping* serves to explicitly relate the elements in each view that are intended to be descriptions of the same entities of the system. For the benefit of the humans reading the different views, it is useful to have a convention that reinforces the mapping, namely that of giving identical names to each view's manifestation of a single entity. Regardless of whether this convention is followed, the mapping establishes what are supposed to be describing the same entities. If in some view, there is no correspondent to an entity defined in another view, then in the first view, the semantics of the entity is that implied by its definition in the other view and that definition's projection, by the mapping, on to the first view. All of this depends on having demonstrated that the models are consistent under the mapping.[1]

Mappings are not a new idea. They are used extensively in automata theory to show different automata equivalent in power [9]; they are used in programming language semantics to show equivalence of definining interpreters [10]; they are used in multi-level formal development methods to show that a refinement to be a correct realization of its abstraction [17]; and they are used in multi-view system design environments to establish the relation between the views to enable analyses by tools in the environment [5].

3. A Small Case Study

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 [6]. 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. It is hoped that in this case, the press is closed too much for a hand to even be inserted.

This system is a good example to introduce and explain our approach, since it comes with interesting safety and real-time constraints, but is simple

---

[1]Related work aims at developing a formal mathematical foundation, i.e. an integrated model comprising all aspects of system behavior [16].

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, and thus a more complete specification is needed.

3.1. Architectural Model

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 object model presented in Figures 2 and 3.

A two-hand press-object is an aggregation of objects of classes *Button*, *PhysicalPress*, and *Sampler*. Aggregation is denoted by links adorned with a rhombus. Multiplicities may be specified explicitly by giving numbers along aggregation links; any missing number is assumed to be "1". The multiplicities in the class diagram and the object diagram both show that, in fact, the two-hand press object, *THP* has two button objects, *B1* on the left and *B2* on the right, one physical press object, *PP*, and one sampler object, *S*. A button is a specialization of a sensor. It offers an operation to read its only measured value, indicating whether the button is currently pressed. A button also incorporates additional operations for pressing and releasing a button. The specialization, i.e. 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, to move down and to stop. The physical press measures three values. These values are further described below.

The communication relationships between objects of this system are summarized in Figure 3: 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 as 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.

3.2. Reactive Model

The top-level reactive behavior of the Two-HandPress class is described by the statechart in Figure 4. Initially, the control remains idle until the sampler signals that the 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 *Malfunction* 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 sub-states 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.

The *Running* state is further refined in Figure 5. 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 transition diagrams, we use stubbed arrows to indicate transitions originating from substates of not yet sufficiently refined states.
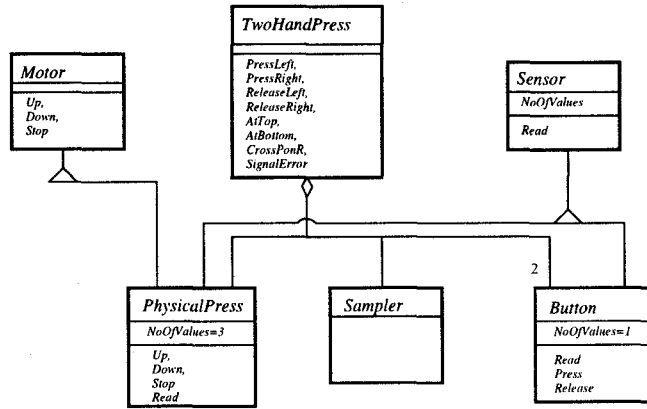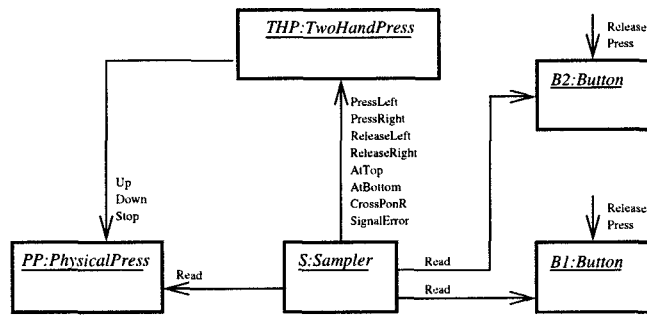
TwoHandPress

PressLeft,
PressRight,
ReleaseLeft,
ReleaseRight,
AtTop,
AtBottom,
CrossPonR,
SignalError

Motor

Up,
Down,
Stop

Sensor

NoOfValues

Read

PhysicalPress

NoOfValues=3

Up,
Down,
Stop
Read

Sampler

Button

NoOfValues=1

Read
Press
Release

2

Figure 2: Object model of the press system

THP:TwoHandPress

PressLeft
PressRight
ReleaseLeft
ReleaseRight
AtTop
AtBottom
CrossPonR
SignalError

Up
Down
Stop

Release
Press

B2:Button

Release
Press

PP:PhysicalPress   Read   **S:Sampler**   Read   **B1:Button**

Read

Figure 3: Communication relationships of the press system

TwoHandPressState

Ready   AtTop   Running

UnexpectedSignal

Malfunction   UnexpectedSignal PhysicalPress.Stop
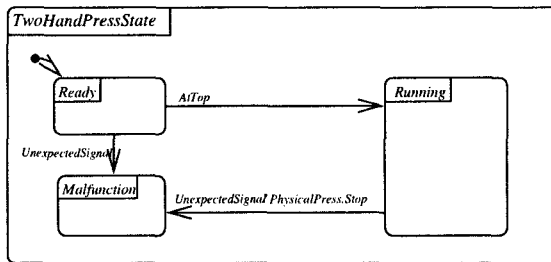
Figure 4: Top-level reactive behavior of the controller

Closing

ClosingAbove

ReleaseLeft or ReleaseRight
/ PhysicalPress.Up

CrossPonR

ClosingBelow

AtBottom / PhysicalPress.Up
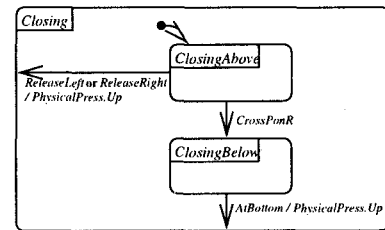
Figure 6: Refinement of the *Closing* state

Running

Open

InitiateClosing / PhysicalPress.Down

Closing

InterruptClosing
/ PhysicalPress.Up   AtBottom / PhysicalPress.Up

Opening

AtTop / PhysicalPress.Stop
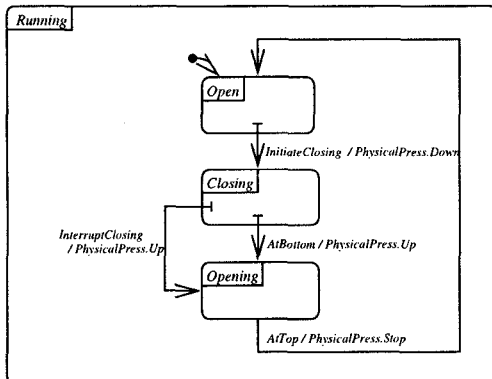
Figure 5: Refinement of the *Running* state

Open

SafetyPosition

ReleaseLeft   ReleaseRight

PressLeft
[BothReleased]   PressRight
[BothReleased]

after MaxDelay msec:
PressTimeOut

LeftPressed   RightPressed

PressRight   PressLeft

InitiateClosing / PhysicalPress.Down
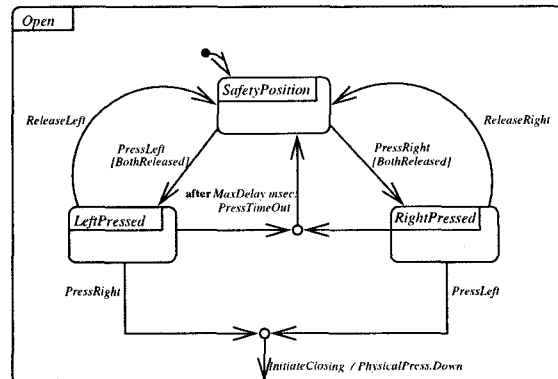
Figure 7: Refinement of the *Open* state

41

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 transition diagram in Figure 6.

The two arrows leaving the refined *Closing* state correspond to the two arrows leaving the respective unrefined state in Figure 5. 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 4.

> *UnexpectedSignals* ≡ *AtTop*[*ClosingBelow*]
> **or** *AtBottom*[*Ready* ∨ *Open* ∨ *ClosingAbove*]
> **or** *CrossPonR*[*Ready* ∨ *Open*]
> **or** *SignalError*

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 transition diagram in Figure 7. 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 have to be pressed within a specific time interval (*MaxDelay* milliseconds). Therefore, the *SafetyPosition* state, which the system enters initially, can be left only when both buttons are released. Now when, e.g. the left button 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 *SafetyPosition* 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 5 can now be defined as follows:

> *InitiateClosing* ≡ *PressLeft* **or** *PressRight*
> *InterruptClosing* ≡ *ReleaseLeft* **or** *ReleaseRight*

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

> **after** *TimeExpression* : *InternalEvent*/
> *RequestedOperations*

If the system has been in the source state of such an arrow for the time specified in the time expres-sion, then it requests operations from other objects and changes into the target state of the arrow. As for operation transitions, the condition and requested operations may be omitted.

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 machines at this point. The state machines for the motor, the buttons, and the physical press are not part of the controller but part of its environment.

3.3. Functional Model

Following common practice when presenting Z specifications, we first specify the state space of the hydraulic press and then the effect of the control operations on this space. The state space is made up of appropriate *data models* of the relevant system components. These models contain the information necessary for the control to decide which action to take. In order to highlight semantic dependencies, we will stick a systematic naming convention: The data model of a unit or a collection of units *U* is named *UModel*. Also, entities in the Z specification that are intended to give semantics to entities in the architectural and reactive specifications have the same name as the corresponding entities, right down to capitalization of the letters.

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

Button ::= *pressed* | *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* | *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 the states of the state machines in terms of our Z model.

```
┌─ ButtonModel ──────────────────────────
│ B1, B2 : Button
│ double_release_required : DoubleRelease
├────────────────────────────────────────
│ (B1 = released ∧ B2 = released) ⇒
│     double_release_required = 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. The logical constraint dictates that a double release is not required if both buttons are released.

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
├────────────────────────────────────────
│ B1 = pressed
│ B2 = pressed
│ double_release_required = 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 a malfunction.

```
TwoHandPress ::=
    ready | open | closingabove | closingbelow |
    opening | malfunction
```

The internal model of the press is defined by:

```
┌─ PressModel ───────────────────────────
│ THP : TwoHandPress
└────────────────────────────────────────
```

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

```
┌─ TwoHandPressState ────────────────────
│ PressModel
│ ButtonModel
├────────────────────────────────────────
│ PressTriggered ⇒ THP ≠ open
│ THP = closingabove ⇒ PressTriggered
└────────────────────────────────────────
```

```
┌─ Create ───────────────────────────────
│ TwoHandPressState'
├────────────────────────────────────────
│ THP' = ready
└────────────────────────────────────────
```

The state schema describes the two-hand press control as consisting of the press model and the button model all being subject to two constraining conditions related to functionality and safety. The first condition states that the press, THP, can be open only if it has not been triggered. The second condition states that above the critical point, THP 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 TwoHandPress is closely related, but not quite identical, to the states used in the reactive view. For example the states Ready and Open can be defined by the following schemas.

```
┌─ Ready ────────────────────────────────
│ TwoHandPressState
├────────────────────────────────────────
│ THP = ready
└────────────────────────────────────────
```

```
┌─ Open ─────────────────────────────────
│ TwoHandPressState
├────────────────────────────────────────
│ THP = open
└────────────────────────────────────────
```

A state can be defined by this sort of schema, for each state in TwoHandPress, and we assume that such states have been defined. Thus, we now have definitions for the Ready, Open, Closing-Above, ClosingBelow, Opening, and Malfunction states. We therefore need definitions only for the substates of Open, namely SafetyPosition, Left-Pressed, and RightPressed.

```
┌─ SafetyPosition ───────────────────────
│ TwoHandPressState
├────────────────────────────────────────
│ THP = open
│ B1 = pressed ∨ B2 = pressed
│     ⇒ double_release_required = required
└────────────────────────────────────────
```

```
__RightPressed_____
  TwoHandPressState
_____
  THP = open
  B1 = released
  B2 = pressed
  double_release_required = notrequired
```

The second condition of *SafetyPosition* states that in the safety position, if any button is pressed, a double release is required before the press may begin to close. The substate *LeftPressed* can be defined analogously, and we assume that it has been.

We do not need definitions for *Closing* and *Running* because these are composed uniquely from their substates *ClosingAbove* and *ClosingBelow* on one hand and *Opening*, *Closing*, and *Open* on the other hand. Strictly speaking, we do not need, in the functional model, a definition of *Open* and *TwoHandPressState*, because these are also composed uniquely from their substates, but in writing the functional model independently of the other models, it felt right to define them.

Operations: We now turn to the specification of the operations of the two-hand press controller. First, we specify the effect of pressing the left button, $B1$. Local to the button model, the effect of this operation can be specified as follows.

```
__PressLeftLocal_____
  ΔButtonModel
_____
  B1 = released
  B1' = pressed
  B2' = B2
  double_release_required' =
      double_release_required
```

This operation can be extended to the two-hand press state by specifying how the press state is affected by the pressing of $B1$. There are two cases. If the right button, $B2$, has already been pressed and no double release is required yet, then the press begins to close. If this is not the case, the press remains open.

```
__PressLeft_____
  ΔPressModel
  PressLeftLocal
_____
  (THP = open ∧ B2 = pressed ∧
      double_release_required = notrequired)
          ⇒ THP' = closingabove
  (THP ≠ open ∨ B2 = released ∨
      double_release_required = required)
          ⇒ THP' = THP
```

This specification captures very succinctly the normal behavior of the operation to press $B1$. The effect of pressing the right button, $B2$, can be specified analogously, and we assume that it has.

Next, we turn to the release operations. Again, we begin by specifying the effect of releasing the left button, $B1$, local to the button control.

```
__ReleaseLeftLocal_____
  ΔButtonModel
_____
  B1 = pressed
  B1' = released
  B2' = B2
  B2 = released ⇒ double_release_required'
      = notrequired
  B2 = pressed ⇒ double_release_required'
      = required
```

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.

```
__ReleaseLeft_____
  ΔPressModel
  ReleaseLeftLocal
_____
  THP = closingabove ⇒ THP' = opening
  THP ≠ closingabove ⇒ THP' = THP
```

Analogously, we can specify the operation to release $B2$, and we assume that it has been done.

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:

```
┌─ AtTop ─────────────────────────────────────
│ ΔTwoHandPressState
├─────────────────────────────────────────────
│ THP ∈ {opening, ready}
│     ⇒ THP' = open
│ THP ∈ {opening, ready} ∧ (B1 = pressed
│          ∨ B2 = pressed)
│     ⇒ double_release_required' = required
│ THP ∈ {closingabove, closingbelow}
│     ⇒ (THP' = malfunction ∧
│        double_release_required' =
│        double_release_required)
│ B1' = B1
│ B2' = B2
└─────────────────────────────────────────────
```

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 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 malfunction state. The remaining operations *CrossPonR*, *AtBottom*, and *SignalError* can be specified in similar styles, and we assume that they have been.

Conditions and Internal events: The condition that both buttons are released can be defined with the schema for *BothReleased*. Finally, we specify, in the schema for *PressTimeOut*, the sole internal event that arises in the case that 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.

```
┌─ BothReleased ──────────────────────────────
│ TwoHandPressState
├─────────────────────────────────────────────
│ B1 = released
│ B2 = released
└─────────────────────────────────────────────
```

```
┌─ PressTimeOut ──────────────────────────────
│ ΔTwoHandPressState
├─────────────────────────────────────────────
│ THP' = THP = open
│ B1 = pressed ⇔ B2 = released
│ double_release_required = notrequired
│ double_release_required' = required
│ B1' = B1
│ B2' = B2
└─────────────────────────────────────────────
```

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 very explicitly 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.

4. Integration of Models

The integration of the models is made rigorous by establishing a mapping between the entities defined and used in the three models. In the preceding sections, we have implicitly assumed such a mapping by using, within the models, identical names for those entities to be mapped to each other.

At the end of the paper, after the bibliography, Figure 9 gives, in tabular form, the mapping between the three models, and Figure 8 gives a legend explaining the notation used in the tables. When a given model does not have a correspondent to an entity in another model, then "—" shows up in the first model's entry for the entity. The indentation of the reactive model state names is intended to illustrate the hierarchical structure of the states in the reactive model.

The reactive and functional view can be checked against each other. 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.

4.1. Relating State Structures

Due to space limitations, we argue the state-structure consistency of only the specific Hydraulic press example. It is hoped that the reader can generalize to any situation involving a Z and a statechart specification. The assertions that must be proved in the general case are given in the technical report [1].

In the case of the hydraulic press, to ensure consistency between these definitions, we have to prove nonemptyness of primitive states,

∃ *Ready*, ∃ *Malfunction*, ∃ *SafetyPosition*,
∃ *LeftPressed*, ∃ *RightPressed*, ∃ *Opening*,
∃ *ClosingAbove*, ∃ *ClosingBelow*

we have to prove disjointness of substates,

disjoint(*ClosingAbove*, *ClosingBelow*)
disjoint(*SafetyPosition*, *LeftPressed*, *RightPressed*)
disjoint(*Opening*, *Closing*, *Open*)
disjoint(*Ready*, *Malfunction*, *Running*)

and we have to prove the necessity and sufficiency
of the **or**-compositions:

Open ⊢
  (*SafetyPosition* ∨ *LeftPressed* ∨ *RightPressed*)
(*SafetyPosition* ∨ *LeftPressed* ∨ *RightPressed*) ⊢
  Open

TwoHandPressState ⊢
  (*Ready* ∨ *Malfunction* ∨ *Running*)
(*Ready* ∨ *Malfunction* ∨ *Running*) ⊢
  TwoHandPressState

4.2. Relating Computational Structures

We have to demonstrate that the set of computations, i.e., sequences of states starting with an initial state and obtained from the previous by application of a transition or operation, in the two models are consistent with each other under the mapping. The standard approach is to use computational induction. That is, show first that the initial states in the two models are consistent under the mapping; this is the basis of the induction. Then show that if states S and S_z are consistent under the mapping, then the states S' and S'_z obtained by applying corresponding transitions and operations to S and S_z, respectively, are also consistent under the mapping; this is the inductive step.

Again, due to space limititations, we argue the induction for the specific hydraulic press example, expecting the reader to generalize. See the technical report [1] for the assertions that must be proved in the general case.

Basis: In the hydraulic press example, in the reactive model, the initial state is *Ready* which maps to the functional model *Ready* state which is defined as the same as the *ready* state into which the *Create* operation puts *THP*.

Inductive Step: The inductive step has to be shown for each operation applied to all possible states to which the operation is applicable by its

pre-conditions. We consider a few example operations.

First, we consider the operation *PressLeft*. It is necessary to find all transitions in the reactive model in which *PressLeft* occurs as the provided operation. These are all in Figure 7, giving the refinement of the *Open* state. In this diagram, there are only two relevant transitions, giving rise to the obligations:

SafetyPosition ∧ *PressLeft* ∧ *BothReleased* ⊢
  LeftPressed'
SafetyPosition ∧ *PressLeft* ∧ ¬ *BothReleased* ⊢
  SafetyPosition'
RightPressed ∧ *PressLeft* ⊢ *ClosingAbove*'

The precondition of *PressLeft* can be calculated in the functional model as

  THP ⊢ pre *PressLeft* ⇔ $B1$ = *released*.

This means that an after-state of *PressLeft* is specified only if $B1$ is released in the pre-state. There are two states in which we know for sure that the left button, $B1$ is not released, namely (1) *Left-Pressed*, following from its own definition, and (2) *ClosingAbove*, following from the second condition of *TwoHandPressState* and the definition of *PressTriggered*. Therefore, the *PressLeft* is inapplicable in two states only, namely:

LeftPressed ⊢ ¬ pre *PressLeft*
ClosingAbove ⊢ ¬ pre *PressLeft*

For the other primitive states, we have to prove preservation, i.e.: for all states S,

S ∧ *PressLeft* ⊢ S'

A similar analysis can be done with the other press and release operations. Next, we turn to the control event *AtTop*. By examination of the statecharts for the top-level behavior and the refinement of the *Running* state in Figures 4 and 5 and the definition of the transition group *Unexpected-Signal*, it is clear that the the transitions to be verified are:

Ready ∧ *AtTop* ⊢ *SafetyPosition*'
Opening ∧ *AtTop* ⊢ *SafetyPosition*'
Running ∧ *ClosingBelow* ∧ *AtTop* ⊢ *Malfunction*'

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

Finally, there is one internal event *PressTime-Out*. From the statechart for the *Open* state, it is clear that the following transitions must be checked.

LeftPressed $\wedge$ *PressTimeOut* $\vdash$ *SafetyPosition'*
RightPressed $\wedge$ *PressTimeOut* $\vdash$ *SafetyPosition'*

Inapplicability is given in 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.

5. Conclusions

This paper has shown a small example of using three different specification languages to specify different aspects of a system under design. Each specification language is used for its own strengths to specify what it can, leaving it to the other languages to deal with what it cannot specify. Of course, it is hoped that every aspect of the system under design that is in need of specification can be specified by at least one of the specification languages used. Each specification is carried out somewhat independently in its own best method; although, it is necessary to keep in mind the other specifications, if only to make sure that all specifications are specifying the same system under design.

Once the specifications are completed it is necessary to define a mapping that relates the two specifications and shows both when an entity is defined in more than one specification and when an entity is defined in only one specification. In the former case, it is necessary to prove that the multiple definitions are consistent with each other under the mapping, i.e., that the different views of that entity do not imply different behavior. In the latter case, if all the proofs have been carried out, we get the right to assume the meaning of the single definition as it projects to all other specifications, i.e., to inherit the single definition as applicable to all specifications.

Indeed, one can consider the models complementary only to the extent that there is a mapping explaining how they complement each other and from which the meanings of missing aspects in one model can be deduced by mapping from their meanings in the others.

In the example of this paper, the complementary models all have a similar conception so it was quite straightforward to find a very simple, identifier-to-identifier mapping between any two of the models. The referees of this paper wondered what would happen if the models were not of similar conception. As suggested by the earlier experience with mappings, cited in Section 2, the mapping can be any collection of functions on any

collection of state variables in one model to individual variables in another. Of course, in this case, it will not be so easy and systematic to find the mappings. However, one thing is clear; complete failure to find a mapping is a sign that the models are not complementary and that more work needs to be done.

Alternatively, one can insist that all people designing one system work closly enough together that the models do have similar enough conceptions that the mapping will be one-to-one.

The reader might also have objected at various points in the presentation of the example, that one can often easily define entities in one model in such a way as to automatically satisfy the proof obligations with respect to their definition in another model. While we admit that this is possible, we want to stress at this point, that our methodological guideline is to define each entity as naturally and as independently as possible from different points of view, perhaps even by different people. In some cases, the consistency between views may follow by construction, in others, consistency must be ensured by a separate nontrivial reasoning.

References

[1] D. Berry and M. Weber. A pragmatic, rigorous integration of structural and behavioral modeling notations. Technical report, GMD-FIRST and Computer Science Dept., Technion, 1997.

[2] G. Booch. *Object-Oriented Analysis and Design with Applications*. Benjamin Cummings, second edition, 1994.

[3] R. Büssow and M. Weber. A steam-boiler control specification with Statecharts and Z. In J. Abrial, H. Langmaack, and E. Börger, editors, *Formal Methods for Industrial Applications: Specifying and Programming the Steam-Boiler Control*, volume 1165 of *LNCS*, pages 109–128. Springer, 1996.

[4] 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, 1993.

[5] G. Estrin. A methodology for design of digital systems — supported by SARA at the age of one. In *Proc. of NCC*, pages 313–336. AFIPS, 1978.

[6] Z. für Unfallverhütung und Arbeitsmedizin. *Pressen – Sicherheitsregeln für Zweihandschaltungen an kraftbetriebenen Pressen der Metallbearbeitung*. Hauptverband der gewerblichen Berufsgenossenschaften, Langwartweg 103, 5300 Bonn 1, zweite edition, 1978.

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

[8] D. Harel and E. Gery. Executable object-modeling with Statecharts. In *Proc. of ICSE 18*, 1996.

[9] J. Hopcroft and J. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison Wesley, Reading, MA, 1979.

[10] C. McGowan. An inductive proof technique for interpreter correctness. In R. Rustin, editor, *Formal Semantics of Computer Languages*, Englewood Cliffs, NJ, 1972. Prentice-Hall.

[11] J. Rumbaugh et al. *Object-Oriented Modeling and Design.* Prentice-Hall, 1991.

[12] I. Software. *Safety-Critical Systems.* IEEE, January 1994.

[13] M. Spivey. *The Z Notation, A Reference Manual.* Prentice Hall, second edition, 1992.

[14] UML Partners Consortium. Version 1.0 of the Unified Modeling Language. Technical report, RATIONAL Software Corporation, 1997.

[15] M. Weber. Combining Statecharts and Z for the design of safety-critical control systems. In M.-C. Gaudel and J. Woodcock, editors, *FME'96: Industrial Benefits and Advances in Formal Methods*, volume 1051 of *LNCS*, pages 307–326. Springer, 1996.

[16] M. Weber. Abstract object systems. Rote Reihe 97-12, TU Berlin, 1997.

[17] W. Wulf, R. London, and M. Shaw. An introduction to the construction and verification of Alphard programs. *IEEE Trans. on SE*, SE-2(4):253–265, December 1976.

Meaning	Code
Object	O
Class/Type	T
Procedure/Operation	P
State	S
Condition/Property	C
Grouped Transition	G
Internal Event	E
Number	N
Model	M
Primitive	p
Defined by Structure	s
Defined by Formula	f
Formula renaming primitive	r
Undefined	u

Figure 8. Legend of mapping between entities in the models

Identifier	Architectural Model	Reactive Model	Functional Model
THP	O	—	Of
TwoHandPress	T	—	Tf
TwoHandPressState	—	Ss	Sf
Ready	—	Sp	Sr
Malfunction	—	Sp	Sr
Running	—	Ss	—
Opening	—	Sp	Sr
Closing	—	Ss	—
ClosingAbove	—	Sp	Sr
ClosingBelow	—	Sp	Sr
Open	—	Ss	Sr
SafetyPosition	—	Sp	Sf
LeftPressed	—	Sp	Sf
RightPressed	—	Sp	Sf
~/Create	—	Pu	Pf
PressLeft	P	Pu	Pf
PressRight	P	Pu	Pf
ReleaseLeft	P	Pu	Pf
ReleaseRight	P	Pu	Pf
AtTop	P	Pu	Pf
AtBottom	P	Pu	Pf
CrossPonR	P	Pu	Pf
SignalError	P	Pu	Pf
UnexpectedSignal	—	GF	—
InterruptClosing	—	GF	—
InitiateClosing	—	GF	—
PressTimeOut	—	Eu	Cf
BothReleased	—	Cu	Cf
MaxDelay	—	Nu	—
ButtonModel	—	—	Mf
PressModel	—	—	Mf
Motor	T	—	—
Up	P	—	—
Down	P	—	—
Stop	P	—	—
PP	O	—	—
PhysicalPress	T	Tu	—
NoOfValues	N	—	—
Up	P	u	—
Down	P	u	—
Stop	P	u	—
Read	P	—	—
S	O	—	—
Sampler	T	—	—
Sensor	T	—	—
NoOfValues	N	—	—
Read	P	—	—
B1	O	—	Of
B2	O	—	Of
Button	T	—	Sf
NoOfValues	N	—	—
Read	P	—	—
Press	P	—	—
Release	P	—	—
pressed	—	—	Sp
released	—	—	Sp
DoubleRelease	—	—	Sf
required	—	—	Sp
notrequired	—	—	Sp
double_release_required	—	—	Of
PressTriggered	—	—	Cf
ButtonModel	—	—	Mf

Figure 9. Mapping between entities in the models