A Pragmatic, Rigorous Integration of Structural and Behavioral Modeling Notations*

Daniel M. Berry
GMD-FIRST
Rudower Chausee 5
12489 Berlin, Germany
dberry@first.gmd.de

normally at: Computer Science Department
Technion
Technion City
Haifa 32000, Israel
dberry@cs.technion.ac.il

AND
Matthias Weber
Research and Technology
Daimler-Benz AG
Alt-Moabit 96a
10559 Berlin, Germany
weber@DBresearch-berlin.de

work done while at: Fachbereich Informatik
Technische Universität Berlin
Franklinstraße 28/29 10587 Berlin, Germany
we@cs.tu-berlin.de

August, 1997

*This paper is an expanded version of a paper of the same title published in the *Proceedings of the International Conference on Formal Engineering Methods*, Hiroshima, Japan, 12-14 November, 1997

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 [6]. 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 ([6, 14, 19], 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 are thought to be guaranteed, but actually is not! Also, such a monolithic approach does not leave much room for coexistence and interaction with other methods that are in standard use in industry.

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 [12]. 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. 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 work, and they are preceived as a net plus, 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, we proceed the other way around.

Indeed, Clarke and Wing, in their survey of the state of the art and future directions of *formal* methods state, "Given that no one formal method is likely to be suitable for describing and analyzing every aspect of a complex system, a practical approach is to use

different methods in combination." [5] It is a case of the whole being greater than the sum of its parts.

1.1 The Modeling Notations

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 a very complex systems manageable and to detect misconceptions or inconsistencies at an early stage. 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.

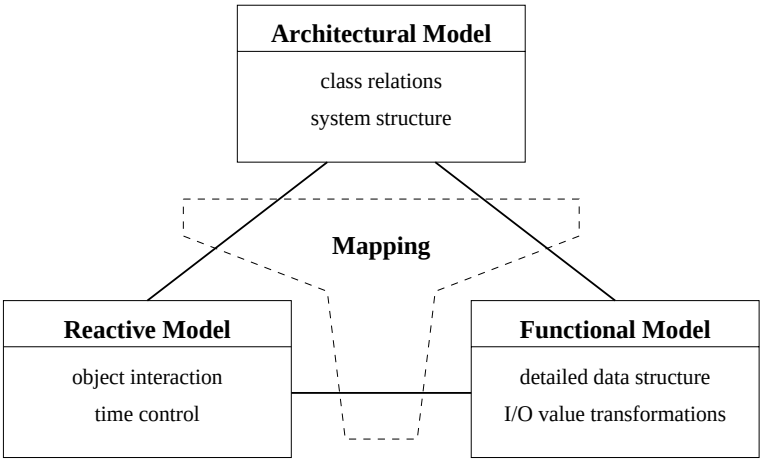

Figure 1: The three modeling views of an embedded system

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 [3, 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. In this setting, we use notation from OMT [17] and the Unified Modeling Language [21]. However, this choice of notation is by no means essential and it should not be difficult for the experienced to express the information content of the object diagrams in his or her favorite notation.

The two other views are primarily concerned with the behavioral specification of the embedded control system. 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. In particular, for any component, it 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 extended state machines specifying how, and under which timing constraints, operations from external objects are requested, 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. a variant of statecharts [10]. There are two reasons for this choice. First, statecharts have proved to be sufficiently expressive for modeling complex component interactions and time control. Second, the use of statecharts, or close variants of statecharts, is currently spreading in industry.

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 [20]. There are two main reasons for using Z. First, in our view, Z has proved to be particularly useful for modeling complex functional data transformations, and second, 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 mathematical notation for modeling functional behavior enables us to prove many abstract 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.

Mappings are not a new idea. They are used extensively in automata theory to show different automata equivalent in power [13]; they are used in programming language semantics to show equivalence of defininig interpreters [16, 1]; they are used in multi-level formal development methods to show that a refinement to be a correct realization of its abstraction [25, 2]; 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 [7, 8].

Early experimentation with a three-fold division of modeling using object modeling, state machines and model-oriented specification has already been reported in previous

work [23, 4]. The present paper extends this work by including a mapping between the models and illustrating how it can serve as a basis for a systematic validation of consistency.

1.2 The Integration Approach

A major technical and methodological goal of the approach presented here is to rigorously ensure consistent modeling of structure and behavior in the three different system models. This problem is not trivial, as different models may deliberately include overlapping information, and one has to check that no inconsistencies are introduced. Conversely, attempting to ensure consistency in the presence of overlapping information usually reveals a number of errors and misconceptions and, ultimately when these are corrected, considerably boosts confidence about the adequacy and robustness of the specification.

As already mentioned, we study here a pragmatic but rigorous approach for ensuring consistency between models by using a mapping to explicitly relate the elements in each view that are intended to be descriptions of the same entities of the system. Based on this mapping, the meaning of an entity in one view is then faithfully projected into its corresponding meaning in a second view. However, as this second view may have its own way of assigning meaning to entities, the projected meaning from the first view corresponds to a set of *properties* about the second view. The consistency of this projected meaning in the second view and the entities' own definitions in that view then amounts to the verification of proof obligations asserting these properties.

This rigorous, pragmatic approach for ensuring consistency between models should ultimately be complemented by a formal mathematical foundation, i.e. a single integrated model comprising all aspects of system behavior. Although the focus of this work is not on mathematical foundations, we would like to refer here to recent work on such an integrated mathematical model, called *abstract object systems*, which might, in the future, serve as a mathematical foundation for our approach [24].

2 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 [9]. 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. 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, if it is not past the critical point.

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 behaviori, and thus a more complete specification is needed.

2.1 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 class and object diagrams presented in Figures 2 and 4.

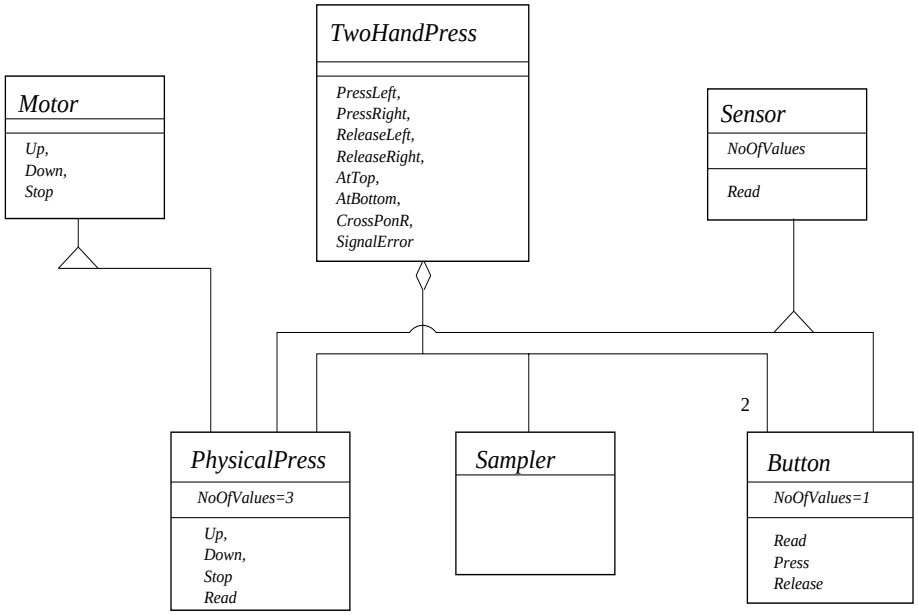

Figure 2: Class diagram of the press system

The class diagram describes a two-hand press-object as consisting of 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. If some numbers are shown, then 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. This value indicates 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, down and to stop. The physical press measures three values. These values are further described below.

In Figure 2, the Sampler and TwoHandPress classes together constitute the software part of the system. The PhysicalPress and Button classes model the behavior of 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 very 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, as shown in Figure 3.

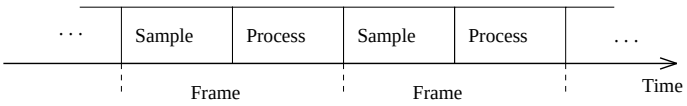

Figure 3: Time-Frame Processing

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 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. The controller requests the operation of individual buttons using natural number indices, e.g. $B1.Read$ reads values from the left button.

The communication relationships between objects of this system are summarized 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

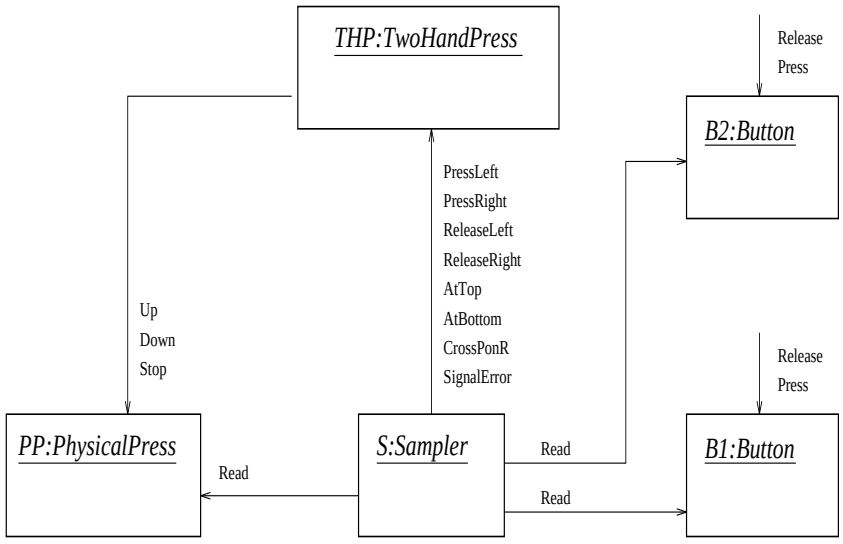

Figure 4: Object diagram showing communication relationships

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.

2.2 Reactive Model of TwoHandPress Class

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

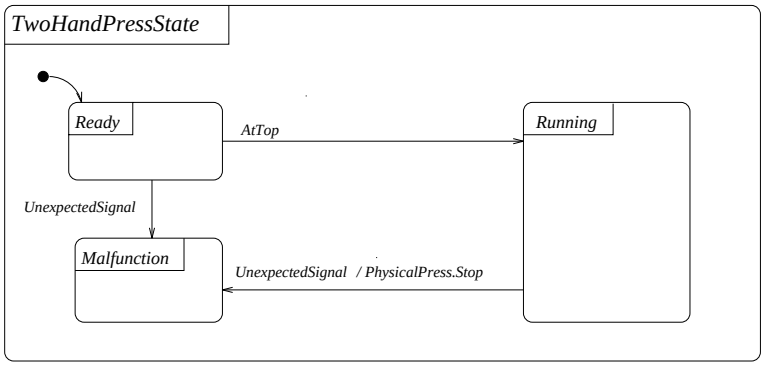

Figure 5: Top-level reactive behavior of the controller

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 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.

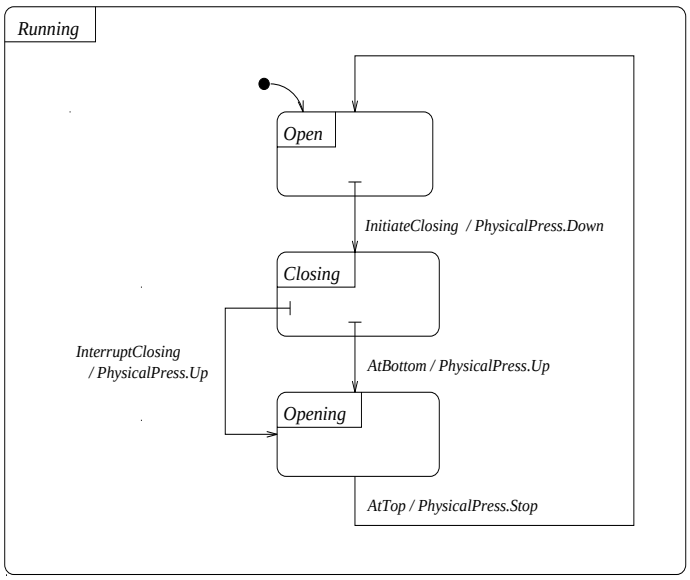

Figure 6: Refinement of the *Running* state

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 transition 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 transition diagram in Figure 7. The two arrows leaving the refined *Closing* state correspond to the two arrows leaving the

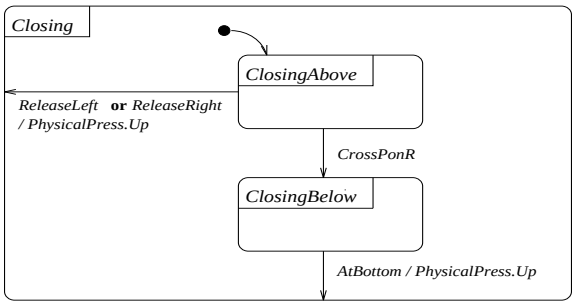

Figure 7: Refinement of the *Closing* state

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.

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

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 transition 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 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 6 can now be defined as follows:

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

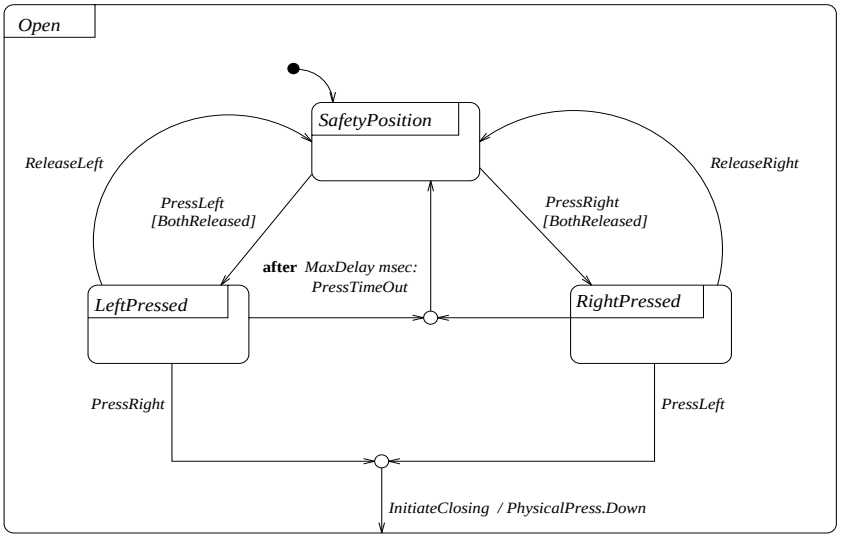

Figure 8: Refinement of the *Open* state

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 expression, 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. Timeout transitions are a very simple, but often sufficient, means to deal with time constraints. If necessary, they could be generalized to timed transitions [15].

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 transition diagrams at this point. The state transition 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 [22]. The present paper is more in the line of current work on embedding statecharts into an object-oriented setting [18] [11]. 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 [18], 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 notation and the semantics assumed here.

2.3 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.

TwoHandPress: 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 the states of the state transition diagrams in terms of our Z model.

ButtonModel

$B1, B2 : Button$

$double\_release\_required : DoubleRelease$

$(B1 = released \land B2 = released) \Rightarrow double\_release\_required = notrequired$

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.

$$TwoHandPress ::= ready \mid open \mid closingabove \mid closingbelow \mid opening \mid malfunction$$

The internal model of the press is defined by:

PressModel

$THP : TwoHandPress$

The initial state is specified by convention in the schema for *Create*:

Ready

$PressModel'$

$THP' = ready$

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

TwoHandPressState

$$PressModel$$
$$ButtonModel$$

$$PressTriggered \Rightarrow THP \neq open$$
$$THP = closingabove \Rightarrow PressTriggered$$

This schema describes the two-hand press model 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*, *ClosingAbove*, *ClosingBelow*, *Opening*, and *Malfunction* states. We therefore need definitions only for the substates of *Open*, namely *SafetyPosition*, *LeftPress*, and *RightPress*.

SafetyPosition

$$TwoHandPressState$$

$$THP = open$$
$$B1 = pressed \lor B2 = pressed \Rightarrow double\_release\_required = required$$

The second condition 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 *RightPressed* can be defined as follows.

RightPressed

$TwoHandPressState$

$THP = open$

$B1 = released$

$B2 = pressed$

$double\_release\_required = notrequired$

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 the *Open* and *TwoHandPress* states, 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.

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.

TwoHandPress: 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

$\Delta ButtonModel$

$B1 = released$

$B1' = pressed$

$B2' = B2$

$double\_release\_required' = double\_release\_required$

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

$\Delta PressModel$

$PressLeftLocal$

$(THP = open \land B2 = pressed \land double\_release\_required = notrequired)$
$\quad \Rightarrow THP' = closingabove$

$(THP \neq open \lor B2 = released \lor double\_release\_required = required)$
$\quad \Rightarrow THP' = THP$

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

$\Delta ButtonModel$

$B1 = pressed$

$B1' = released$

$B2' = B2$

$B2 = released \Rightarrow double\_release\_required' = notrequired$

$B2 = pressed \Rightarrow double\_release\_required' = required$

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

$\Delta PressModel$

$ReleaseLeftLocal$

$THP = closingabove \Rightarrow THP' = opening$

$THP \neq closingabove \Rightarrow THP' = THP$

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

$\Delta TwoHandPressState$

$THP \in \{opening, ready\}$
$\quad \Rightarrow THP' = open$
$THP \in \{opening, ready\} \land (B1 = pressed \lor B2 = pressed)$
$\quad \Rightarrow double\_release\_required' = required$
$THP \in \{closingabove, closingbelow\}$
$\quad \Rightarrow (THP' = malfunction \land$
$\quad double\_release\_required' = double\_release\_required)$
$B1' = B1$
$B2' = B2$

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.

TwoHandPress: Conditions

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

BothReleased

$TwoHandPressState$

$B1 = released$
$B2 = released$

TwoHandPress: Internal events

Finally, we specify 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.

PressTimeOut

$\Delta TwoHandPressState$

$THP' = THP = open$

$B1 = pressed \Leftrightarrow B2 = released$

$double\_release\_required = notrequired$

$double\_release\_required' = required$

$B1' = B1$

$B2' = B2$

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.

3 A Rigorous Integration

The integration of the models is made rigorous by establishing a mapping between the entities defined and used in the three models.

3.1 Mapping Between Models

The straightforward idea is to establish a mapping between the entities of each model. 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.

Figures 9 and 10 give, in tabular form, the mapping between the three models, and Figure 11 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.

3.2 Reactive Model vs. Functional Model

The reactive and functional view of an embedded system can be checked against each other by 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.

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
$\rightsquigarrow$/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

Figure 9: Mapping between entities in the models, part I

Identifier	Architectural Model	Reactive Model	Functional Model
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 10: Mapping between entities in the models, part II

Figure 12 uses a schematic example to illustrate the basic issues with respect to the relationship between the reactive and the functional model. It shows both models for a

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 11: Legend of mapping between entities in the models

simple class with two operations, two internal events and two conditions. It may be helpful to study this example when following the subsequent definitions.

3.2.1 Relating State Structures

We assume that corresponding Z schemas have been defined in the functional model for each of the states of the reactive model. Given such a mapping for a particular component, the state structure semantics of the reactive view can now be projected into properties about the corresponding Z schemas in the functional view. For an arbitrary state S of 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. This is denoted by the state S having an "Sp" entry in the reactive model column of the mapping table. The semantics of primitive states then implies that the associated Z state S_z must be nonempty, i.e.

  Consistency: $\exists\, S_z$.

- 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, where $n > 0$. This is denoted by the state S having an "Ss" entry in the reactive model column of the mapping table and S's substates being listed below it indented one level. We assume that these substates are mapped to Z-schemas S_{1z}, S_{2z}, $\cdots$, and S_{nz}. The semantics of hierarchical state decomposition then implies the disjointness of these substates.

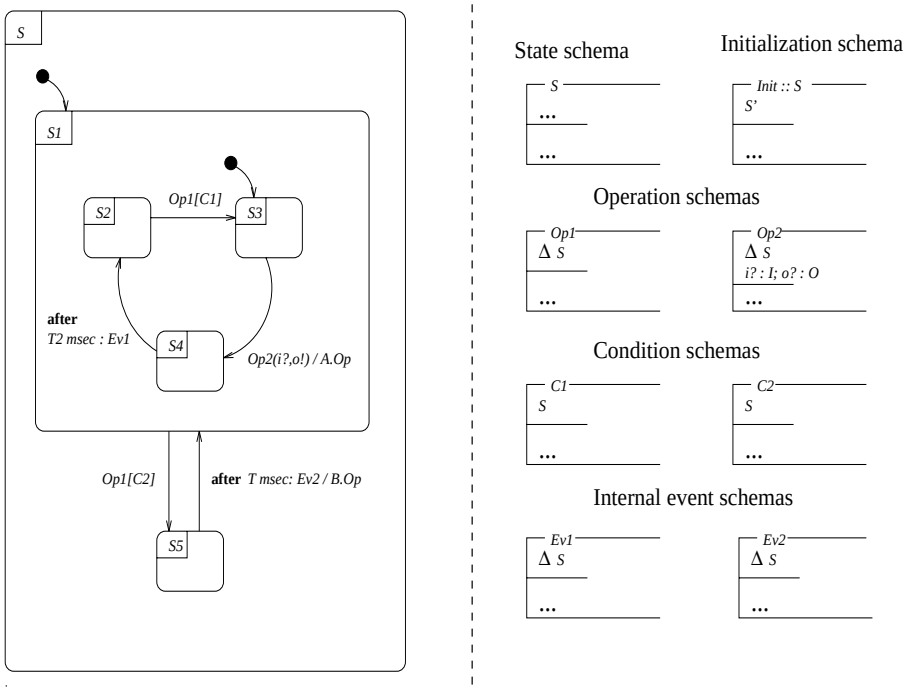

Figure 12: The reactive and the functional model of a component

Disjointness: $\neg\ (S_{iz} \wedge S_{jz})$ for all $i, j \in \{1, \cdots, n\}$, where $i \neq j$. This is abbreviated as $disjoint(S_{1z},\ S_{2z},\ \cdots,\ S_{nz})$.

If, in addition, S is mapped to Z-schema S_z, then one has to verify sufficiency and necessity of the decomposition.

Sufficiency: $S_{1z} \vee S_{2z} \vee \cdots \vee S_{nz} \vdash S_z$.

Necessity: $S_z \vdash S_{1z} \vee S_{2z} \vee \cdots \vee S_{nz}$

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

Hydraulic press example

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

$\exists\, Ready$, $\exists\, Malfunction$, $\exists\, SafetyPosition$,

$\exists\, LeftPressed$, $\exists\, RightPressed$

$\exists\, ClosingAbove$, $\exists\, ClosingBelow$, $\exists\, Opening$

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:

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

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

and

$TwoHandPressState \vdash (Ready \vee Malfunction \vee Running)$

$(Ready \vee Malfunction \vee Running) \vdash TwoHandPressState$

3.2.2 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 of the induction. If one can show these two, clearly, all corresponding states in all corresponding computations in the two models are consistent under the mapping.

3.2.3 Relating Initializations

We have to check whether the initialization schema is consistent with the initialization as indicated in the state transition diagrams. If $Create :: C$ is the name of Z schema describing the initialization and S is the state reached after initialization in the reactive model of an object, then we have to check the following proposition.

Initialization: $Create :: C \vdash S$

Hydraulic press example

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*.

3.2.4 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 in the mapping, 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 semantics of transitions from 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 state specified in the reactive model. 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 schemes mapped 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 under the mapping. 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 in which $n = 1$ and $C_1 \Leftrightarrow true$.

Hydraulic press example

Having carried out the basis for the induction in a previous section, we now consider some example inductive steps. 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 8, giving the refinement of the *Open* state. In this diagram, there are only two relevant transitions, giving rise to the obligations:

$$SafetyPosition \wedge PressLeft \wedge BothReleased \vdash LeftPressed'$$
$$SafetyPosition \wedge PressLeft \wedge \neg\, BothReleased \vdash SafetyPosition'$$
$$RightPressed \wedge PressLeft \vdash ClosingAbove'$$

The precondition of *PressLeft* can be calculated in the functional model as $THP \vdash \mathrm{pre}\, PressLeft \Leftrightarrow B1 = released$. 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

LeftPressed, following from its own definition, and

ClosingAbove, following from the second condition of *TwoHandPressState* and the definition of *PressTriggered*.

Therefore, the *PressLeft* 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, i.e.: for all states S,

$$S \wedge PressLeft \vdash S'$$

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 5 and 6 and the definition of the transition group *UnexpectedSignal*, it is clear that the 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 *Malfunction*. The other control events can be analyzed in a similar fashion.

Finally, there is one internal event *PressTimeOut*. 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 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.

4 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 descriptions of the same entity have the same name and that all specifications are specifying the same system under design.

Once the specifications are completed it is necessary to define a mapping that makes the identity of like named entities explicit. The purpose of this mapping is to show 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. This right is precisely what is needed when different, complementary specifications are given of the one system in order that all of its aspects are covered by at least one specification.

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. To misquote Marshall McLuhan, "the proof is the massage", the massage that converts one model into another.

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 an earlier draft of this paper wondered what would happen if the models were not of similar conception. As suggested by the experience in automata theory, the theory of equivalence of interpreters, multi-level formal development methods, and multi-view system design environments, cited in Section 1.1, 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 quite straightforward.

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.M. Berry. The equivalence of models of tasking. *Proc. of ACM Conf. on Proving Assertions about Programs, SIGPLAN Not.*, 7(1), January 1972.

[2] D.M. Berry. Towards a formal basis for the Formal Development Method and the Ina Jo specification language. *IEEE Trans. on SE*, SE-13(2):184–201, 1987.

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

[4] R. Büssow and M. Weber. A steam-boiler control specification with Statecharts and Z. In J.R. 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.

[5] E.M. Clarke and J.M. Wing. Formal methods: State of the art and future directions. Technical report, Carnegie Mellon University, 1996.

[6] 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.

[7] 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.

[8] G. Estrin, R.S Fenchel, R.R. Razouk, and M.K. Vernon. SARA (System ARchitect's Apprentice): Modeling, analysis, and simulation support for design of concurrent systems. *IEEE Trans. on SE*, SE-12(2):293–311, 1986.

[9] Zentralstelle 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.

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

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

[12] 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.

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

[14] 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, 1991.

[15] Y. Kestens and A. Pnueli. *Timed and Hybrid Statecharts and their Textual Representation*, volume 299 of *LNCS*, pages 591 – 620. Springer, 1992.

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

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

[18] B. Selic, G. Gullekson, and P.T. Ward. *Real-Time Object-Oriented Modeling.* John Wiley & Sons, 1994.

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

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

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

[22] M. von der Beeck. A comparison of statecharts variants. In *Symposium on Fault-Tolerant Computing*, LNCS. Springer, 1994.

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

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

[25] W.A. Wulf, R.L. 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.