Integrating Action Systems and Z in a Medical System Specification

V. Kasurinen and K. Sere

University of Kuopio, Department of Computer Science and Applied Mathematics,
P.O.Box 1627, FIN-70211 Kuopio, Finland

Abstract. Action systems and the specification language Z, are integrated and used to give different aspects of a single systems specification. The static part of the system under development, i.e., the state declarations together with their accompanying invariants, are first specified using Z. Then the reactivity and real time aspects of the same system are specified within the action systems framework making use of the Z definitions. Furthermore, Z style is used to specify the actions in the action system. We exemplify the proposed methodology through a case study on a medical system specification.

1 Introduction

This paper reports on work carried out on formal specification of a computer-based system that is used to train the reaction abilities of patients with severe brain damage. The system contains computer programs by which the patients carry out different tests that are designed to stimulate their eyes and ears. Systems of this type are new and no formal specifications for them exists to our knowledge. The system specified here is developed together with the neurological clinic of a Finnish university hospital. The patients who carry out the tests are in rehabilitation. The tests themselves are specified by the medical staff of the clinic.

Our system supports currently three reaction training tests. The formal specification of the system became important due to several reasons. First of all, there is the question of patients' integrity, because the system is used within health care: the system supports a data base on the test results of several patients. Furthermore, the test results themselves are interpreted and manipulated by the system, so the computation needs to be reliable. In addition, there is a family of related tests designed for different patient groups. All the tests are based on similar ideas, but the stimuli and the interpretation of the results vary. Hence, we wanted to create a high level specification of the systems that would be easy to adapt to the different needs.

The system consists of two parts: There is an interactive part where different stimuli appear on an ordinary computer screen at in advance determined times. A stimulus is either a picture that is drawn on the screen or a picture combined with a sound that comes from the loud speaker of the computer. This part also monitors the reactions the patient makes to different stimuli. The reactions are

accepted via an ordinary keyboard. The other part takes care of the result data base. It also specifies the test to be carried out, stores the reactions, interprets them, and computes the results. The system has certain real-time requirements, partly constraining the appearances of stimuli and partly the time a patient has to make her reactions to different stimuli.

The two parts are completely different in nature. Therefore it was felt that a single formalism would not be optimal for the specification of the entire system. As a matter of fact, we started by developing the complete specification using the specification language Z [10, 11]. The Z formalism was very powerful when specifying the result data base and defining the different tests. It turned out, however, that this formalism was not very suitable for specifying the reactive behavior of the system.

The interactive, reactive part is specified within the action systems framework of Back et al. [2, 3]. Action systems is a formalism designed for the development of verified parallel and distributed systems. It has also been successfully applied in many reactive applications and furthermore, an extensive theory exists for the stepwise development of action systems from high level specification to an implementation as described by Back and Sere [1, 3].

All the operations and activities that have real time requirements are specified within the action systems formalism. We adopt the timed action systems formalism proposed by Fidge and Wellings [6]. That part of the activity that does not have timing constraints is specified within the Z framework.

We make contributions as follows:

- we show how two formalism are integrated in a single specification task,
- we propose a mixed style of specification, where activity is specified using both formalisms, and
- we develop a formal specification of a real world medical computer-based system.

We proceed as follows. In section 2, we define the state space of the system within the specification language Z. Also a couple of Z operations are defined to give a flavor of the interface between the medical staff and the system. These operations do not involve real time. We assume that the reader is familiar with the Z language. The action system specification for the interactive part is given in section 3 together with a short introduction to this formalism. The activity specified here is constraint by some real time requirements. We conclude in section 4.

2 The Z interface

The system we are about to specify consists of three basic components, the impulses, the reactions, and the tests. For a test, a number of impulses appear on the computer screen and the patient carrying out the test is given time to react, from a keyboard, to every impulse. Let us start by defining each component separately.

2.1 The impulses

The set of possible impulses is here left unspecified. We only assume that there is a set of distinct types of impulses in the system:

$$[IMPULSE]$$

A member of the set *IMPULSE* could be a circle, a square, or a triangle. The impulses appear in bursts of at most *burst* impulses.

$$| \; burst : \mathbb{N}$$

The time stamps that denote the appearances of the impulses on the screen are members of $AbsTime$

$$AbsTime == \mathbb{N}$$

Each appearance of an impulse has two features. First, two consecutive impulses can appear on a regular, irregular, or occasional time interval. Second, an impulse appears either on a defined or an undefined position on the screen.

If an interval is regular, then every two consecutive impulses always have the same interval between them. The set *RegularInterval* gives the possible intervals between regular impulses.

$$RegularInterval == \{di : AbsTime \mid di > 0\}$$

An occasional interval is such that its length has a lower bound and an upper bound. The set *OccasionalInterval* gives the minimum and maximum interval between two consecutive impulses.

$$OccasionalInterval == \{t1, t2 : AbsTime \mid t1 > 0 \land t1 < t2 \bullet (t1, t2)\}$$

Finally, an irregular interval in our system is retrieved from a set of three possible interval lengths. The set *IrregularInterval* defines the possible irregular intervals between impulses.

$$IrregularInterval == \{t1, t2, t3 : AbsTime \mid t1 > 0 \land t1 < t2 \land t2 < t3 \bullet (t1, t2, t3)\}$$

For example, if a regular interval is 8, the impulses arise at intervals of 8 time units. If an irregular interval is $(8, 10, 12)$, this denotes that after the first impulse the next one should arise either 8, 10 or 12 time units later. If an occasional interval is $(6, 60)$, it means that between every two impulses there is at least 6 and at most 60 time units.

Next we need to specify the screen coordinates of the position where an impulse appears:

$$[Coordinates]$$

The screen coordinates and the impulses form the set *ScreenInstance*.

$$ScreenInstance == \{si : Coordinates \times IMPULSE\}$$

There is a component in the set *ScreenInstance* that models an empty screen. This is defined as a constant *empty_screen*:

$$| \; empty\_screen : ScreenInstance$$

Moreover, when the position of an impulse is *defined*, the values of the screen coordinates are given by *defplace*:

$$| \; defplace : Coordinates$$

If there is a sound stimulus connected to an impulse, sound precedes the impulse at a regular interval on some positive number of time units.

$$SoundInterval == \{sct : AbsTime \mid sct > 0\}$$

2.2 The reactions

The reactions that a patient makes correspond to keypressings from an ordinary keyboard. For every reaction, i.e., set of keypressings, we store the time it has taken for the patient to make her reaction is terms of *AbsTime*. The value 0 denotes no reaction to a stimulus. At most *burst* reactions will be stored. The set *Reactions* contains the recorded reactions during a test.

$$Reactions == \{la : \mathrm{seq}\, AbsTime \mid \#la \leq burst\}$$

Let us look at an example. A sequence

$$\langle 3, 0, 2, 0, 0, 0 \rangle$$

denotes that a patient has reacted by pressing some key for the first and third impulse. There has been no reaction for the second, fourth, fifth and sixth impulses.

2.3 The tests

The system presently supports three tests. In this paper two of them, test 1 and test 2, will be specified. Both tests contain three phases, A, B, and C. These are defined in *TestNumber*:

$$TestNumber ::= T1A \mid T1B \mid T1C \mid T2A \mid T2B \mid T2C$$

Every test phase comes with a set of parameters that will be called test limits. These parameters give particular values to the different types of interval between impulses and sound interval. Furthermore, they give the type of an impulse used, *basim*, and the time it stays on the screen. The schema type *TestLimit* specifies the possible test limits.

TestLimit

$basim : IMPULSE$
$defint : RegularInterval$
$unint : IrregularInterval$
$ocint : OccasionalInterval$
$sotime : SoundInterval$
$deftime : AbsTime$
$defsotime : AbsTime$

$defsotime < sotime$

$deftime < defint - sotime$
$deftime < first(unint) - sotime$
$deftime < first(ocint) - sotime$
$deftime > 0$

where the intuition behind the predicate is illustrated in Figure 1.

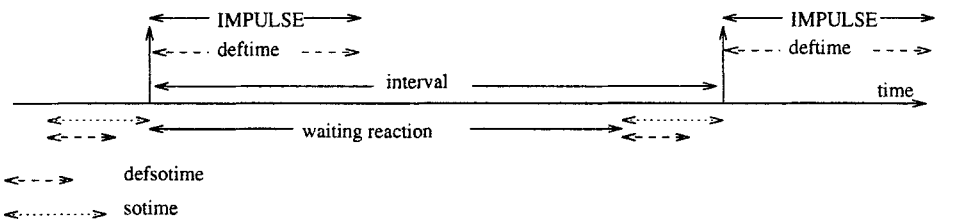


Fig. 1. Test parameters. The concept 'interval' denotes either a *defint* interval, an *ocint* interval, or an *unint* interval. The period when a patient is supposed to make a reaction is illustrated with 'waiting reaction'.

The predicate states that *sotime*, the time between a sound and the accompanying impulse, should be less than the entire interval. The parameter *defsotime* denotes time when sound is on before next impulse appears. After an impulse appears on the screen, it disappears *deftime* time units later. If a test does not include sound, a patient can make her reaction at any time between the impulses. The intuition of the test parameters for a test without sound is illustrated in Figure 2.

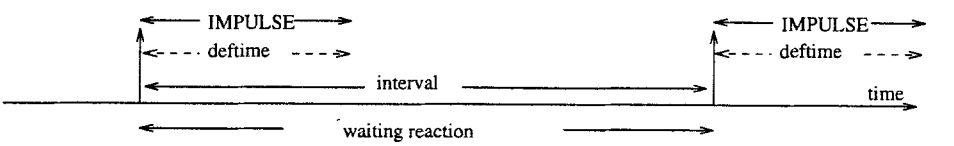


Fig. 2. Only the parameters *defint*, *ocint*, and *unint* are meaningful for defining 'waiting reaction' period, when the test does not have a sound signal.

A reaction is not taken into consideration in the following cases:

- It is made at the same time when an impulse appears.
- It is made at any time between a sound signal and the accompanying impulse.
- It is made at the same time when the sound signal is set on.

For example, a member of a valid set of test parameters *TestLimit* can be as follows: $defint = 12$, $unint = (10, 12, 15)$, $ocint = (10, 20)$, $sotime = 2$, $deftime = 6$, and $defsotime = 1$. The test parameters are interpreted as follows:

- If the test is without a sound signal, the first impulse appears at time stamp 0. Because *deftime* is 6, the impulse disappears at time stamp 6. If the interval is regular it is possible to make a valid reaction between times 1 and 9. A new impulse arises at time 10. In an interval is irregular, it is possible to make a reaction between time stamps 1 and 9, 1 and 11, or 1 and 14, because the next impulse appears either at time stamp 10, 12, or 15, respective. If an interval is occasional, a lower bound of 'waiting reaction' period is 1, and a upper bound can varies from 9 to 19.
- If the test has a sound signal included, the sound appears at time stamp 0 and it is on until time stamp 1, because *defsotime* is defined to be 1. The first impulse appears at time stamp 2. Because *deftime* is set to 6, the impulse disappears at time stamp 8. When the interval is regular, a valid reaction must be made between time stamps 3 and 11. At time 12, a new sound signal appears. In the case the interval is irregular, it is allowed to make a reaction between the time stamps 3 and 9, because a new sound signal appears at the earliest at time 10. It is also possible to make at time 10, because it might be so that the sound appears only at the time stamp 14. At the latest , the reaction can be made at the time stamp 14, because the last possible time for a sound in an irregular interval is 15. If the interval is occasional, the reaction must come at earliest at time 3 and the upper bound can varies from 9 to 19.

The global variable *testlimit* keeps track of the set of parameters currently in use.

$$testlimit : TestLimit$$

2.4 Global functions

We finally define a set of global functions. First of all, the features of an impulse depend on the test number.

The function *defso* returns the interval between an impulse and its preceding sound, if any (only test 2 includes sound).

$$defso : TestNumber \nrightarrow AbsTime$$

$$\forall tn : TestNumber \bullet$$
$$(tn \in \{T1A, T1B, T1C\} \Rightarrow defso(tn) = 0)$$
$$\wedge$$
$$(tn \in \{T2A, T2B, T2C\} \Rightarrow defso(tn) = testlimit.sotime)$$

The function *generate_impulse* when given a test number returns the corresponding screen coordinates for the designated impulse (*testlimit.basim*).

$$
\begin{array}{|l}
\hline
generate\_impulse : TestNumber \pfun ScreenInstance \\
\hline
\forall\, tn : TestNumber \bullet \\
\quad (tn \in \{T1A, T1B, T1C\} \Rightarrow \\
\qquad generate\_impulse(tn) = (defplace, testlimit.basim)) \\
\quad \land \\
\quad (tn \in \{T2A, T2B, T2C\} \Rightarrow \\
\qquad (\exists\, i : Coordinates \bullet generate\_impulse(tn) = (i, testlimit.basim))) \\
\hline
\end{array}
$$

Finally, the impulse interval, regular, irregular, or occasional, defines the time units between impulses, therefore, one of the interval parameters *defint*, *ocint*, or *unint* in *testlimit* is set to be the center of attention in a test phase. The function *defimptime* accepts a test number and outputs a time.

$$
\begin{array}{|l}
\hline
defimptime : TestNumber \pfun AbsTime \\
\hline
\forall\, tn : TestNumber \bullet \\
\quad (tn \in \{T1A, T2A\} \Rightarrow defimptime(tn) = testlimit.defint) \\
\quad \land \\
\quad (tn \in \{T1B, T2B\} \Rightarrow \\
\qquad (\exists\, a, b, c, d : AbsTime \bullet \\
\qquad\quad (a, b, c) = testlimit.unint \land d \in \{a, b, c\} \land \\
\qquad\quad defimptime(tn) = d)) \\
\quad \land \\
\quad (tn \in \{T1C, T2C\} \Rightarrow \\
\qquad (\exists\, a, b, c : AbsTime \bullet \\
\qquad\quad (a, b) = testlimit.ocint \land c \in a \,..\, b \land \\
\qquad\quad defimptime(tn) = c)) \\
\hline
\end{array}
$$

2.5 Operation schemas

We define two Z operations, *GiveTestNumber* and *GenerateResult*, for our system. These operations model the interface between the medical staff and the system.

The operation *GiveTestNumber* accepts as input the test phase.

$$
\begin{array}{|l}
\hline
GiveTestNumber \\
\hline
tn? : TestNumber \\
\hline
\end{array}
$$

The operation *GenerateResult* is activated at the end of a test phase. Its sole purpose is to compute the number of reactions.

GenerateResult

$reac? : Reactions$

$res! : (RightReactions \times WrongReactions \times Reactions)$

$\#reac? = burst$

$res! = (checkanswer(reac?), burst - checkanswer(reac?), reac?)$

Here an input *reac*? is generated by the action system. The types *RightReactions* and *WrongReactions* are defined as follows:

$$RightReactions, WrongReactions == \mathbb{N}$$

Furthermore, the function *checkanswer* computes the number of right answers for a test phase. It accepts a test result, a member of the set *Reactions*, and returns the number of acceptable reactions.

$checkanswer : Reactions \nrightarrow \mathbb{N}$

$$\begin{aligned}
&\forall at : Reactions;\ rt : AbsTime \bullet \\
&\quad rt = head\ at \wedge \\
&\quad (tail\ at = \langle\rangle \wedge \\
&\qquad (rt \neq 0 \Rightarrow \\
&\qquad\quad checkanswer(\langle at \rangle) = 1) \wedge \\
&\qquad (rt = 0 \Rightarrow \\
&\qquad\quad checkanswer(\langle at \rangle) = 0)) \wedge \\
&\quad (tail\ at \neq \langle\rangle \Rightarrow \\
&\qquad checkanswer(\langle at \rangle) = checkanswer(\langle rt \rangle) + checkanswer(tail\ at))
\end{aligned}$$

In the actual implementation, a sequence *reac*? has much more information than what is specified here. Therefore, the calculation of the results is also much more elaborate. Our objective here is to only show the style of a mixed specification.

3 An action system interface

The interactive part of the rehabilitation system is specified using the action systems framework enhanced with real time facilities. The specification language Z is used when individual actions are specified.

3.1 Actions and action systems

Actions An action is any statement in an extended version of Dijkstra's guarded command language [5]. This language includes (multiple) assignment, sequential composition, conditional choice, and iteration, and is defined using *weakest precondition* predicate transformers.

An action A is a guarded command of the form $g \rightarrow S$, where g is a predicate, and S is any statement. An action is said to be *enabled* in some state when its guard g is true in this state.

Action systems An *action system* has the form

$$\mathcal{A} ::= [\![\textbf{var } x;\ y^* \bullet I;\ \textbf{do } A_1 [] \ldots [] A_m \textbf{ od}]\!] : z$$

The action system $\mathcal{A}$ is initialized by action I. Then, repeatedly, an enabled action from $A_1 \ldots A_m$ is nondeterministically selected and executed. The action system terminates when no action is enabled, and aborts when some action aborts.

The *local* variables of $\mathcal{A}$ are the variables x and the *global* variables of $\mathcal{A}$ are the variables y (exported) and z (imported). The local and global variables are assumed to be distinct. Each variable is associated with an explicit type. The *state variables* of $\mathcal{A}$ consist of the local variables and the global variables. The actions are allowed to refer to all the state variables of an action system.

Action systems are similar to the Unity logic of Chandy and Misra [4] and are also related to Lamports Temporal Logic of Actions [9]. However, contrary to these approaches we make no assumptions about fairness.

Parallel composition of action systems Consider two action systems $\mathcal{A}$ and $\mathcal{B}$:

$$\mathcal{A} ::= [\![\textbf{var } x;\ y^* \bullet I;\ \textbf{do } A_1 [] \ldots [] A_m \textbf{ od}]\!] : z$$

$$\mathcal{B} ::= [\![\textbf{var } u;\ v^* \bullet J;\ \textbf{do } B_1 [] \ldots [] B_n \textbf{ od}]\!] : w$$

where x, y, z, u, and w are distinct variables.

We define the *parallel composition* $\mathcal{A} \,||\, \mathcal{B}$ of $\mathcal{A}$ and $\mathcal{B}$ to be the action system

$$\mathcal{C} ::= [\![\textbf{var } x;\ y^*;\ u;\ v^* \bullet I;\ J;\ \textbf{do } A_1 [] \ldots [] A_m [] B_1 [] \ldots [] B_n \textbf{ od}]\!] : z \cup w.$$

Thus, parallel composition will combine the state spaces of the two constituent action systems, merging the global variables and keeping the local variables distinct. The behavior of a parallel composition of action systems is dependent on how the individual action systems, the *reactive components*, interact with each other via the global variables that are referenced in both components. We have for instance that a reactive component does not terminate by itself: termination is a global property of the composed action system. More on these topics can be found in [1].

Timed action systems A *timed action system* $\mathcal{T}(\mathcal{A})$ is a parallel composition of an action system $\mathcal{A}$ and an action system $\mathcal{C}$ that contains actions to move time forward. The enabledness of an action in $\mathcal{A}$ might depend on time, but none of its actions is allowed to modify time. Moreover, an action in the action system $\mathcal{C}$ is enabled only when there are no enabled actions in $\mathcal{A}$.

Hence, we have that

$$\mathcal{T}(\mathcal{A}) ::= [\![now : Time \bullet \mathcal{A} \,||\, \mathcal{C}]\!]$$

This way of specifying time within action systems was suggested by Fidge and Wellings [6].

3.2 An action system specification

Let us now give an action system specification of the interactive part of our case study.

The global variable *now* denotes the time as read from a clock.

$$Time \mathrel{\widehat{=}} [now : AbsTime]$$

The variable *kp* turns to *true* when a patient has made a keypressing:

$$KeyPress \mathrel{\widehat{=}} [kp : \mathbb{B}]$$

The variable *next* keeps track of the time when the next impulse should appear on the screen:

$$NextTime \mathrel{\widehat{=}} [next : AbsTime]$$

The behavior of the patient is stored in the sequence *reac*.

$$Reaction \mathrel{\widehat{=}} [reac : Reactions]$$

The action system specification of the rehabilitation program is a parallel composition of three separate action systems *SCREEN*, *REGISTER* and *LOUDSPEAKER*:

$$SYSTEM ::= \\ [\![SCREEN \mid\mid REGISTER \mid\mid LOUDSPEAKER]\!] : Time, KeyPress$$

The reactive components communicate with the environment via the variables *kp*, and *now*. The variable *kp* is set to true when a patient makes her reaction. Moreover, there is a clock system $\mathcal{C}$ that moves time forward. The global variable *now* that models time is initially set to zero, and it is replaced by $now + 1$ only when there are no enabled actions in *SYSTEM*. The enabledness of all the actions in *SYSTEM* depends on *now* as described below.

First, we specify the action system *SCREEN*, which generates impulses on the screen. The variable *scr* represents a state of the screen. The variable l keeps track of the length of *reac*. In addition to exported variable *next*, there is another exported variable *prev* keeps track of the time when the previous impulse appeared on the screen.

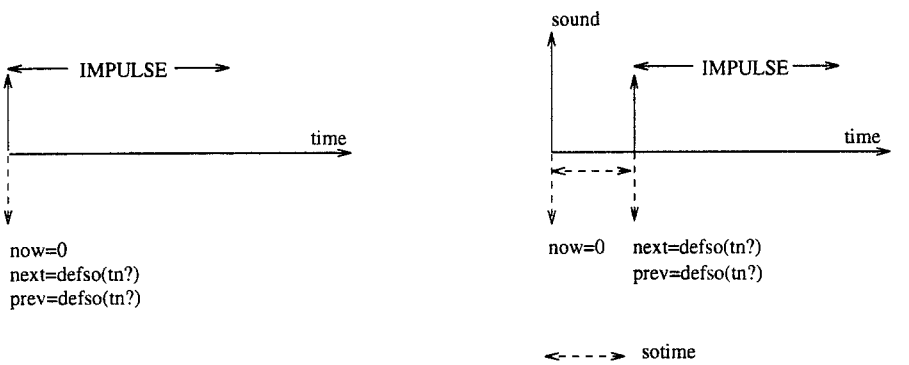
$$SCREEN ::= \\ [\![\mathbf{var}\ scr : ScreenInstance;\ NextTime(next^*);\ prev^* : AbsTime;\ l : \mathbb{N} \bullet \\ InitScreen; \\ \mathbf{do} \\ l < \#reac < burst \wedge now = next \longrightarrow ShowImpulse;\ CountNext \\ [] \\ now = prev + testlimit.deftime \longrightarrow ClearScreen \\ \mathbf{od} \\ ]\!] : reac, now$$

The operation *InitScreen* initializes variables *next*, and *prev* using the function *defso*. The schema *InitScreen* is as following, and it is illustrated in Figure 3.

InitScreen

scr; $NextTime$; $prev$; l

$scr = empty\_screen \land l = -1 \land$
$next = defso(tn?) \land prev = defso(tn?)$



a) A test without a sound.

b) A test with a sound.

Fig. 3. Test initialization. The initial values of the variables *next* and *prev* are given by the function *defso*.

The operations *ShowImpulse* and *ClearScreen* are as follows:

$$ShowImpulse \mathrel{\widehat=} [\Delta(scr) \mid scr' = generate\_impulse(tn?)]$$

$$ClearScreen \mathrel{\widehat=} [\Delta(scr) \mid scr' = empty\_screen]$$

The operation *ShowImpulse* is enabled when time has reached *next* and the variable *reac* has been updated to reflect the result of the previous impulse. The operation *ClearScreen* makes an impulse to fade away when it is taken $testlimit.deftime$ time units after the impulse appeared. The operation *CountNext* below calculates time of the next appearance of an impulse.

CountNext

$\Delta(NextTime, prev)$

$next' = next + defimptime(tn?) \land$
$prev' = next \land l' = l \dot{+} 1$

The situation after the operations *ShowImpulse*, and *CountNext* is illustrated in Figure 4.

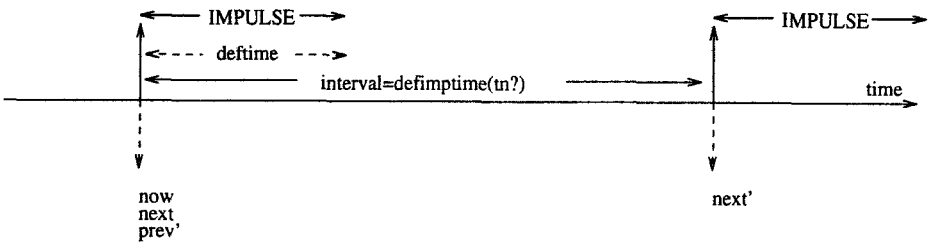


Fig. 4. The impulse appears, and time of the next impulse appearance is calculated.

The action system *REGISTER* generates the exported variable *reac* from the keypressings of a patient. The variable *kp* below is global and its value is set to *true* from outside of the system. The variable *reg* is set *true* when a patient has made a reaction in allowable period.

$$
\begin{array}{l}
REGISTER ::= \\
[\![\textbf{var } reg : \mathbb{B};\ Reaction(reac^*) \bullet \\
InitRegister; \\
\textbf{do} \\
kp \wedge \neg\, reg \wedge now < next - defso(tn?) \longrightarrow StoreReaction \\
[] \\
\neg\, kp \wedge \neg\, reg \wedge now = next - defso(tn?) \wedge now > 0 \longrightarrow StoreEmpty \\
[] \\
reg \wedge now = next - defso(tn?) \longrightarrow StartAgain \\
\textbf{od} \\
]\!] : next, now, prev, kp, tn?
\end{array}
$$

Here the initialization is defined as follows:

$$InitRegister \mathrel{\widehat{=}} [KeyPress;\ reg;\ Reaction \mid reg \wedge reac = \langle\rangle]$$

When the first reaction to an impulse is observed, i.e., the value of *kp* is *true*, and the time has not reached $next - defso(tn?)$, which it the time to generate the next stimulus, the value of the variable *reg* becomes *true*. The operation *StoreReaction* then records the reaction.

$$StoreReaction \mathrel{\widehat{=}} [\Delta(reg, Reaction) \mid reg' \wedge reac' = reac \frown \langle now - prev \rangle]$$

Both situations, i.e., a test with and a test without sound is are illustrated in Figure 5, and Figure 6.
If there has been no observable reaction to a stimulus, *kp* is *false* and time is already $next - defso(tn?)$, the operation *StoreEmpty* concatenates *reac* and an empty time stamp 0.

$$StoreEmpty \mathrel{\widehat{=}} [\Delta(Reaction) \mid reac' = reac \frown \langle 0 \rangle]$$

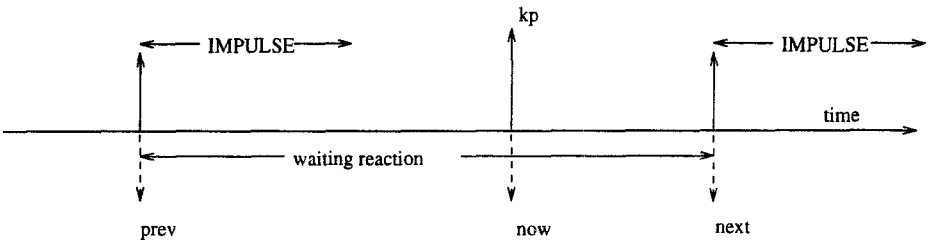


Fig. 5. There is no sound signal before the appearance of an impulse.

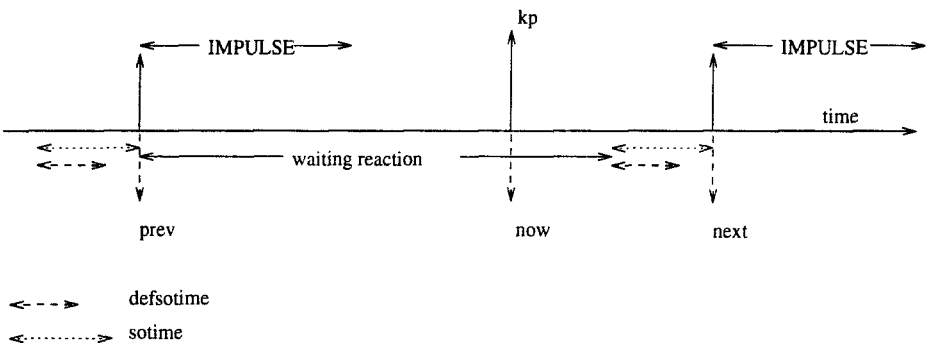


Fig. 6. There is a sound signal before the appearance of an impulse.

Finally, only the first keypressing to an impulse is registered. Therefore, immediately prior the next impulse, the operation *StartAgain* resets the variable *reg* to *false*.

$$StartAgain \mathrel{\widehat{=}} [\Delta(reg, KeyPress) \mid \neg\, reg']$$

The action system *LOUDSPEAKER* inserts a sound signal to the test. There is only one variable *sound* to denote if a sound signal on or off.

$LOUDSPEAKER ::=$
$[\![$ **var** $sound : \mathbb{B} \bullet$
$InitLoudSpeaker;$
do
$now = next - testlimit.sotime \wedge defso(tn?) > 0 \longrightarrow SoundOn$
$[]$
$now = next - (testlimit.sotime - testlimit.defsotime) \wedge sound \longrightarrow SoundOff$
od
$]\!] : next, now, tn?$

The initialization and the operations are straightforward:

$$InitLoudSpeaker \mathrel{\widehat{=}} [sound \mid \neg\, sound]$$

$$SoundOn \mathrel{\widehat{=}} [\Delta(sound) \mid sound']$$

$$SoundOff \mathrel{\widehat{=}} [\Delta(sound) \mid \neg\, sound']$$

The value of *testlimit.sotime* gives the interval between a sound and its accompanying impulse. The value of *testlimit.defsotime* gives the time a sound is on. This is illustrated in Figure 7.

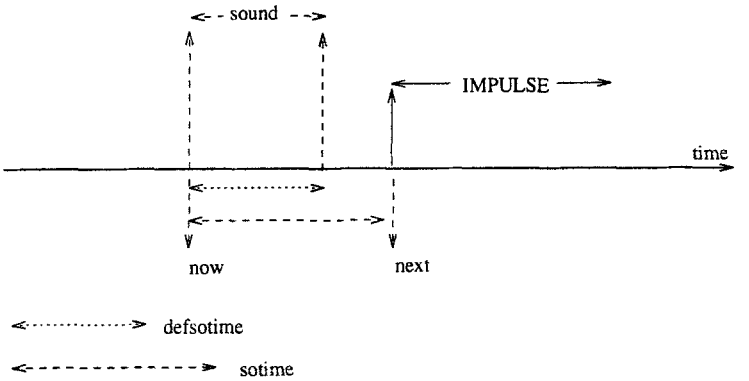


Fig. 7. Sound signal precedes an impulse.

4 Concluding remarks

We have formally specified a system that train the reaction abilities of different groups of patients suffering from severe brain damage. Our specification was based on existing programs and on the available documentation on these. It was, however, found important for the security and future development of such programs to redo the informal specifications and hence, a formal way of specifying this type of systems was attempted.

We found the idea of integrating Z and the action systems framework very natural to work with. The action systems could easily refer to variables and functions defined using Z. The concurrent and real time aspects are very naturally specified via the actions.

This work is part of our research on integrating formal methods and the system design process. We are particularly interested in the interplay between informal and formal methods in the definition phase when creating a requirements specification for a system [7, 8].Furthermore, the *refinement calculus* for action systems as described by Back and Sere [1, 3] can be directly used for further development of the system.

Acknowledgments

The authors like to thank the personnel at the neurological clinic of the University Hospital in Kuopio, and at the Brain Research and Rehabilitation Center

of Finland for cooperation within the development of the Rekku system. We also thank Michael Butler and Marina Waldén for helpful comments on Z specification. The research is supported by the Academy of Finland and Tekes, The Technology Development Center of Finland.

References

1. R. J. R. Back. Refinement calculus, part II: Parallel and reactive programs. In J. W. de Bakker, W.–P. de Roever, and G. Rozenberg, editors, *Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness. Proceedings. 1989*, volume 430 of *Lecture Notes in Computer Science*. Springer–Verlag, 1990.
2. R. J. R. Back and R. Kurki-Suonio. Decentralization of process nets with centralized control. In *Proc. of the 2nd ACM SIGACT–SIGOPS Symp. on Principles of Distributed Computing*, pages 131–142, 1983.
3. R. J. R. Back and K. Sere. From modular systems to action systems. *Proc. of Formal Methods Europe'94*, Spain, October 1994. *Lecture Notes in Computer Science*. Springer–Verlag, 1994.
4. K. M. Chandy, J. Misra. *Parallel Program Design: A Foundation*. Addision–Wesley, 1988.
5. E. W. Dijkstra. *A Discipline of Programming*. Prentice–Hall International, 1976.
6. C. Fidge, A. Wellings. *An action-based formal model for concurrent, real-time systems*. Software Verification Research Center, Technical report 95-1, January 1995.
7. V. Kasurinen. *Informal and Formal Requirements Specification*. Manuscript, University of Kuopio, Department of Computer Science and Applied Mathematics, 1995.
8. V Kasurinen, K. Sere. Data Modelling in ZIM. To appear in *Proc. of Methods Integration Workshop*, Leeds Metropolitan University, 1-2 April 1996.
9. L. Lamport. *A Temporal Logic of Actions*. Research Report No. 57, DEC System Research Center, 1990.
10. B. Potter, J. Sinclair and D. Till. *An Introduction to formal specification and Z*. Prentice Hall, Englewood Cliffs NJ, 1991.
11. J. B. Wordsworth. *Software Development with Z*. Addison-Wesley, Wokingham, England, 1992.