Formal Specifications of Calendar Scheduler

- The Z and CSP approach

Models of Software Systems, Fall 2000
PhD Project Report

Lu Luo
December 2000

Formal Specifications of Calendar Scheduler
- The Z and CSP approach

Abstract
This report presents formal specifications using Z and CSP for a calendar scheduler system.

Introduction
This report is for a PhD project of the ISRI course Models of Software Systems at School of Computer Science of Carnegie Mellon University. The report is composed of three parts. In the first part, a Z specification for the software system of a calendar scheduler is given; in the second part, the same system is specified using the language of Communication Sequential Processes (CSP); a comparison of the two specifications will be given in the third part of this report.

The system

Calendar management is one of the beastly problems of computing. The purpose of a calendar scheduler is to maintain consistent meetings schedules for a number of people. These schedules record at least the time, duration, and participants in each meeting. Some of the meetings may include people whose schedules are not maintained by the calendar scheduler. Meetings may be added or dropped at any time (up to the moment when they occur), and participants of a meeting can be added or removed. A meeting may be scheduled at any time that is convenient for all (or enough) of the meeting participants, except that some of the meetings may need to occur in a particular order. The scheduler may maintain information about the scheduling preference of the people it involves. [1]

Z specification language

The Z formal specification language is used for the specification of large software systems. The language is based on set theory with certain extensions, such as types and schemas. The language draws on the full expressive power of sets and relations and may be used to specify systems at both an abstract and a detailed level. The most cost-effective use of Z is for abstract specific ation of system or software requirements. At this level, it is possible to use Z to create a model of the system requirements and analyze that model to ensure that the system exhibits desired properties and does not have undesired properties. [2]

CSP specification language

The language of CSP was designed for describing systems of interacting components, and it is supported by an underlying theory for reasoning about them. The conceptual framework taken by CSP is to consider components, or processes, as independent self-contained entities with particular interfaces through which they interact with their environment. Processes can be combined to form a larger system, which is again a self-contained entity with a particular interface – a (larger) process. [3]

Part I The Z Specification for Calendar Scheduler

1. The state space

The calendar scheduler deals mainly with meetings, time, and people. Although scheduling a meeting in the real world might need to take into consideration details such as its location and subjects, within the scope of our specification, we only care the time when the meeting is going to be held, how long will it last and who will attend the meeting. A meeting is distinguished from other meetings by its id, which is defined as primitive type:

$[MEETINGID]$

In reality, the format of the time a meeting is held should be at least expressed as the format "Hour-Minute-Month-Day-Year". For the sake of focusing on the key points of the specification and reducing complexity on trivial operational details, the time in this calendar scheduler system is defined as two abbreviations of the set of natural numbers $\mathbb{N}$, *AbsTime* and *Duration. AbsTime* stands for the set of absolute time of each minutes, e.g. if an natural number 0 stands for the absolute time of 12:00 am, January 1, 2000, then the number 26 will stands for 12:26 am, January 1, 2000, and so forth. Duration stands for the set of relative time counted in minutes, e.g. 5 stands for a period of 5 minutes, etc.

$AbsTime == \mathbb{N}$

$Duration == \mathbb{N}$

An abbreviation *TimeSlot* is defined as the Cartesian Product of *AbsTime* and *Duration*, an element in *TimeSlot* indicating a time slot which has a begin time *AbsTime* and a *Duration*.

$TimeSlot == AbsTime \times Duration$

A global variable *Now* is defined indicating the current absolute time, as will be explained later on in this report, some operations will need current time for comparison purposes.

$Now: AbsTime$

It is required that a meeting should be scheduled at a time that is convenient for all (or enough) of the meeting participants. The minimum number that meets the requirement of "enough" is defined as follows:

$MinimumSize: \mathbb{N}$

There will be situations when operating on meetings and personal schedules that two time slots are overlapped, the axiom of overlapping is defined as follows:

$Overlap: TimeSlot \leftrightarrow TimeSlot$

$\forall s_1, s_2: TimeSlot$
  $\bullet\ (s_1, s_2) \in Overlap$

$$\Leftrightarrow s_1 = s_2 \lor s_1 . 1 < s_2 . 1 + s_2 . 2 \lor s_2 . 1 < s_1 . 1 + s_1 . 2$$

When two time slots overlap, the *AbsTime* field in one time slot must fall into the other time slot, that means, the beginning time of one time slot is either equal to the beginning of the other time slot, or it is later than the beginning time of the other time slot but earlier than its end time, which is acquired by adding the begin time *AbsTime* and the *Duration*.

There are also needs to measure whether an absolute time is within the scope of a time slot, the axiom is defined as TimeIn as follows:

$$TimeIn: AbsTime \leftrightarrow TimeSlot$$

$$\forall t: AbsTime \bullet \forall s: TimeSlot \bullet (t, s) \in TimeIn \Leftrightarrow s . 1 \leqslant t \leqslant s . 1 + s . 2$$

Some detailed facts of a person attending the meetings are not in the scope of our problem. The time slots preferred by this person in which h/she is willing to attend a meeting is defined as *preferredTime* and the time slots in which this person is engaged in meetings is defined as *occupiedTime*, both of them are finite power set of *TimeSlot.*

Person

$preferredTime: \mathbb{F}\ TimeSlot$

$occupiedTime: \mathbb{F}\ TimeSlot$

A Meeting has a unique *meetingId* of the type *MEETINGID* that distinguish this meeting from other meetings. The local variable *time* in a meeting is of the type *TimeSlot*, which contains both the beginning time and the duration of this meeting. The finite power set of *Person*, namely *attendance*, contains a group of person who will attend this meeting.

Meeting

$meetingId: MEETINGID$

$time: TimeSlot$

$attendance: \mathbb{F}\ Person$

The state space of the calendar scheduler is described with the schema *CalendarScheduler*:

CalendarScheduler

$meetings: \mathbb{F}\ Meeting$

$group: \mathbb{F}\ Person$

$personalSchedule: Person \nrightarrow \mathbb{F}\ Meeting$

$\operatorname{dom} personalSchedule = group$

$\operatorname{ran} personalSchedule = \mathbb{F}\ meetings$

$\forall m_1, m_2: meetings$

$\bullet\ m_1 \neq m_2$

$\Leftrightarrow m_1 . meetingId \neq m_2 . meetingId \land (m_1 . time, m_2 . time) \in Overlap$

$$\Leftrightarrow m_1 . attendance \neq m_2 . attendance$$

The variables declared in above schema represent different observations of the calendar scheduler:

- *meetings* is a finite set of meetings of the basic type *Meeting*, where each meeting is planned and maintained in the calendar scheduler system;
- *group* is a finite set of *Person*, the calendar scheduler system only maintains the personal schedule (defined later in this report) for the members of the *group*, although any person outside the group can attend any meetings;
- *personalSchedule* is a function from *Person* to $\mathbb{F}$ *Meeting*, which, when applied to a specific person, gives all the meetings at which this person is to be an attendance. Only the personal schedule of a member of a certain group is maintained by this calendar schedule system.

The invariants of *CalendarScheduler* give the relationships between state variables and will be true in every state of the system and will be maintained by every operation on it. Where:

- The domain of function *personalSchedule* is *group*, only the personal schedules of members in the group are maintained by calendar scheduler system;
- The range of function *personalSchedule* is a finite subset of *meetings*, the meetings one person in the group attends is among the *meetings* maintained by calendar scheduler;
- The predicate:

$$\begin{aligned}&\forall m_1, m_2 : meetings\\ &\quad \bullet\ m_1 \neq m_2 \Leftrightarrow m_1 . meetingId \neq m_2 . meetingId\\ &\qquad \wedge (m_1 . time, m_2 . time) \in Overlap\\ &\quad \Leftrightarrow m_1 . attendance \neq m_2 . attendance\end{aligned}$$

gives the relationship between two meetings in the set of *meetings*. For any two meetings m_1 and m_2 in *meetings*, m_1 is different from m_2 if and only if either the two meetings have different meeting ids; if the time slots of meeting m_1 and m_2 overlap, it must be true that not all the attendance of these two meetings are the same. An interesting fact here is that, one person can attend two overlapping meetings.

2. Initialization

The initial state of calendar scheduler is defined as follows:

InitCalendarScheduler

CalendarScheduler

$$meetings = \varnothing$$
$$group = \varnothing$$
$$personalSchedule = \varnothing$$

At the very beginning, the calendar is blank, there is no record for meetings and personal schedules in the system, and nobody is yet a group member.

3. Operations

The operations in calendar scheduler can be classified into three categories: operation on meetings, operation on people and operation on the group. Operations in the same category have similar form and can be encapsulated as follows:

MeetingOp

$\Delta CalendarScheduler$

$group' = group$

The operation on meetings will change most variables in *CalendarScheduler*, but leave the set *group* unchanged.

The operation on person only changes the personal schedule of a person, provided he/she is a member of *group*, and leaves the set of *meetings* and *group* unchanged.

PeopleOp

$\Delta CalendarScheduler$

$meetings' = meetings$

$group' = group$

The operation on group changes the composition of the *group* and the personal schedules of group members, leaving the set *meetings* unchanged in *CalendarSchduler*.

GroupOp

$\Delta CalendarScheduler$

$meetings' = meetings$

Plan a meeting

A meeting may be scheduled at any time that is convenient for all (or enough) of the meeting participants, except that some of the meetings may need to occur in a particular order. The operation *PlanMeeting* plans a meeting according to the personal time preferences of potential attendances and to the schedules of other planned meetings before adding this meeting to the calendar. The minimum (enough) number of participants is defined above as a natural number *MinimumSize*:

$MinimumSize: \mathbb{N}$

The *PlanMeeting* operation has four input variables:

- *id?* is a *MEETINGID* that identifies the meeting being planned;
- *who?* is a finite power set of *Person* who are going to attend the meeting being planned;

- *expBeginTime?* is a finite set of absolute time *AbsTime*, indicating all the suitable absolute time on which the meeting being planned is expected to be scheduled. This input variable reflects the requirement for scheduling a meeting that some of the meetings may need to occur in a particular order. For example, if we hope this meeting to be scheduled between two other meetings, the set *expBeginTime?* should indicate some absolute time that is in the middle of the time slots of the other two meetings.

The *PlanMeeting* operation outputs a finite power set of *AbsTime*, *time!*, any absolute time in the set *time!* is a suitable time to schedule the meeting, without violating personal time preferences and personal meeting schedules of the group members, therefore, the new meeting can be scheduled at any absolute time in the set *time!*.

The *PlanMeeting* operation does not change the system state since it only gives out a set of suitable begin time for the meeting:

PlanMeeting

$\Xi CalendarScheduler$
$id?: MEETINGID$
$who?: \mathbb{F}\ Person$
$expBeginTime?: \mathbb{F}\ AbsTime$
$prfTimeSlot, ocpTimeSlot: \mathbb{F}\ TimeSlot$
$prfTime, ocpTime: \mathbb{F}\ AbsTime$
$time!: \mathbb{F}\ AbsTime$

$\forall m: meetings \bullet m . meetingId \neq id?$
$prfTimeSlot = \{ t: TimeSlot \mid \forall p: group \bullet p \in who? \wedge t \in p . preferredTime \}$
$prfTime = \{ t: AbsTime \mid \forall ts: prfTimeSlot \bullet (t, ts) \in TimeIn \}$
$ocpTimeSlot = \{ t: TimeSlot \mid \forall p: group \bullet p \in who? \wedge t \in p . occupiedTime \}$
$ocpTime = \{ t: AbsTime \mid \forall ts: ocpTimeSlot \bullet (t, ts) \in TimeIn \}$
$prfTime \cap expBeginTime? \neq \emptyset \Leftrightarrow time! = prfTime \cap expBeginTime?$
$prfTime \cap expBeginTime? = \emptyset \wedge expBeginTime? \setminus ocpTime \neq \emptyset$
$\Leftrightarrow time! = expBeginTime? \setminus ocpTime$
$prfTime \cap expBeginTime? = \emptyset \wedge expBeginTime? \setminus ocpTime = \emptyset \Leftrightarrow time! = \emptyset$

The precondition:

$\forall m: meetings \bullet m . meetingId \neq id?$

restricts that the meeting being planned should not be already in the set of *meetings*, in which every meeting has been scheduled in the calendar scheduler.

Several local variables are needed in order to get more straightforward predicates instead of long, complicated ones in the schema:

- *prfTimeSlot* is a finite power set of *TimeSlot*, which collects the time slots that are preferred by all the people who are both a group member and an attendance of the meeting being planned, i.e. the preferred time slots of the people in the intersection of set *who?* and *group*;

$$prfTimeSlot = \{\ t: TimeSlot \mid \forall p: group \bullet\ p \in who? \wedge t \in p\ .\ preferredTime\ \}$$

- *ocpTimeSlot* is a finite power set of *TimeSlot*, which collects all the time slots that are occupied by a meeting on the personal schedules of attending group members;

$$ocpTimeSlot = \{\ t: TimeSlot \mid \forall p: group \bullet\ p \in who? \wedge t \in p\ .\ occupiedTime\ \}$$

- *prfTime* is a finite set of AbsTime, which indicating that the meeting can begin on any absolute time in this set;

$$prfTime = \{\ t: AbsTime \mid \forall ts: prfTimeSlot \bullet\ (t, ts) \in TimeIn\ \}$$

- *ocpTime* is a finite set of AbsTime, which indicating that the meeting cannot begin on any absolute time in this set.

$$ocpTime = \{\ t: AbsTime \mid \forall ts: ocpTimeSlot \bullet\ (t, ts) \in TimeIn\ \}$$

Now we have two sets of absolute time that match the personal preference of everybody in the group as well as their personal schedules. *prfTime* tells us when all these people are willing to attend a meeting and *ocptime* tells us when all the group members are not available, within the given expected beginning time of the meeting. If we can find some absolute time in both sets prfTime and expBeginTime? we are free to choose any time as the beginning time of the new meeting:

$$prfTime \cap expBeginTime? \neq \emptyset \Leftrightarrow time! = prfTime \cap expBeginTime?$$

It is possible that we cannot find any expected begin time that the attending group members prefer in common, i.e. $prfTime \cap expBeginTime? = \emptyset$, but some expected begin time not occupied by meetings on the attendances' personal schedules, then we can schedule the new meeting on these absolute time, which will not disturb group member's personal schedule, although not preferred by all:

$$\begin{aligned} & prfTime \cap expBeginTime? = \emptyset \wedge expBeginTime? \setminus ocpTime \neq \emptyset \\ & \Leftrightarrow time! = expBeginTime? \setminus ocpTime \end{aligned}$$

If neither any preferred begin time nor non-occupied time is found, the meeting planning operation fails and returns an empty set of absolute time, in order the begin time of the meeting to be planned successfully, another expected begin time of the meeting should be chosen.

$$prfTime \cap expBeginTime? = \emptyset \wedge expBeginTime? \setminus ocpTime = \emptyset \Leftrightarrow time! = \emptyset$$

Assign the attributes to a meeting

By above operation *planMeeting*, if a set of absolute time (either is preferred by attending group members, or not occupied by those people) can be found, the beginning time of the new meeting can be chosen from the set of absolute time. Before adding the new meeting to the calendar scheduler, all the local variables should be given a value:

The operation *AssignTime* has following input variables:

- *id?* is the meeting ID that is unique to this meeting;
- *time?* is an absolute time chosen from the result absolute time set of PlanMeeting operation, or a time that doesn't conflict with most of the group members attending this meeting;
- *who?* is the set of people who will participate in this meeting, there could be non-group-member people who want to attend the meeting.

AssignMeeting

Ξ *CalendarScheduler*
id?: MEETINGID
time?: TimeSlot
who?: $\mathbb{F}$ *Person*
groupmember: $\mathbb{F}$ *Person*
meeting!: Meeting

$\forall m: meetings \bullet m . meetingId \neq id?$
$groupmember = \{ p: Person \mid p \in group \land p \in who? \}$
$\# groupmember \geqslant MinimumSize$
$meeting! . meetingId = id?$
$meeting! . time = time?$
$meeting! . attendance = who?$

The local variable *groupmember* is a temporary set of *Person* who are both attendance of the meeting and a group member, the size of *groupmemter* should be greater than or equal to *MinimumSize*, which makes sure that enough group member attend the meeting.

The post-conditions of *AssignMeeting* operation simply assign the *meetingId*, *time* and *attendance* to the output variable *meeting!*.

Add a meeting to the calendar

After assigning the attributes to a meeting, the schedule and attendance of this meeting can be added to the *CalendarScheduler*:

AddMeeting

MeetingOp
newmeeting?: Meeting

$$\begin{array}{l}
\forall m: meetings \bullet newmeeting?.meetingId \neq m.meetingId \\
Now \leqslant newmeeting?.time.1 \\
\forall p: group \\
\quad \bullet\ p \in newmeeting?.attendance \\
\qquad \Leftrightarrow personalSchedule'\ p = personalSchedule\ p \cup \{newmeeting?\} \\
\qquad\quad \wedge\ p.occupiedTime = p.occupiedTime \cup \{newmeeting?.time\} \\
meetings' = meetings \cup \{newmeeting?\}
\end{array}$$

The operation *AddMeeting* has a input variable:

- *newmeeting?* is the meeting that is ready to be added to and maintained by *CalandarScheduler*, where the *time*, *attendance* and *meetingId* of this meeting have been assigned by AssignMeeting operation defined above.

The precondition of the *AddMeeting* operation is that no existing meeting in the set *meetings* has the same meeting Id as the new meeting, this makes sure that the meeting is new to the system and no existing meeting and personal schedule is modified during this operation.

$$\forall m: meetings \bullet newmeeting?.meetingId \neq m.meetingId$$

The time that a new meeting is added to the system should be no later than the beginning time of this meeting. As defined above, the global variable *Now* indicates current absolute time, and the first element of a TimeSlot is an absolute time, which, when used on meeting, tells us the beginning time of the meeting:

$$Now \leqslant newmeeting?.time.1$$

On adding a new meeting to the system, the personal schedules of group members will be changed, i.e. the new meeting is added to the *personalSchedule* of those who are both a group member and the attendance of the new meeting, the time slots this new meeting takes will also be marked on the *occupiedTime* list of those group members:

$$\begin{array}{l}
\forall p: group \\
\quad \bullet\ p \in newmeeting?.attendance \\
\qquad \Leftrightarrow personalSchedule'\ p = personalSchedule\ p \cup \{newmeeting?\} \\
\qquad\quad \wedge\ p.occupiedTime = p.occupiedTime \cup \{newmeeting?.time\}
\end{array}$$

The new meeting is also added to the set *meetings*:

$$meetings' = meetings \cup \{newmeeting?\}$$

Drop a meeting from the calendar

If, for some reason, a meeting has to be cancelled, it is necessary to remove the schedule for this meeting from the calendar. The *DropMeeting* operation serves this purpose:

$$
\begin{array}{|l}
\textit{DropMeeting} \\
\hline
\textit{MeetingOp} \\
\textit{abortmeeting?}: \textit{Meeting} \\
\hline
\textit{abortmeeting?} \in \textit{meetings} \\
\textit{Now} \leqslant \textit{abortmeeting?} \,.\, \textit{time} \,.\, 1 \\
\textit{meetings}' = \textit{meetings} \setminus \{\textit{abortmeeting?}\} \\
\forall p: \textit{group} \\
\quad \bullet\ p \in \textit{abortmeeting?} \,.\, \textit{attendance} \\
\qquad \Leftrightarrow p \,.\, \textit{occupiedTime} = p \,.\, \textit{occupiedTime} \setminus \{\textit{abortmeeting?} \,.\, \textit{time}\} \\
\qquad\quad \wedge\ \textit{personalSchedule}'\ p = \textit{personalSchedule}\ p \setminus \{\textit{abortmeeting?}\} \\
\hline
\end{array}
$$

The input variable *abortmeeting?* is the meeting that is being cancelled.

The preconditions of this operation restrict that the *abortmeeting?* must already be an element of *meetings* set, i.e. the meeting being aborted is already a meeting maintained by calendar scheduler system. It is also required that the time *Now* to drop the meeting should be before the time the meeting begins or during the process of this meeting:

$$
\begin{array}{|l}
\textit{abortmeeting?} \in \textit{meetings} \\
\textit{Now} \leqslant \textit{abortmeeting?} \,.\, \textit{time} \,.\, 1 + \textit{abortmeeting?} \,.\, \textit{time} \,.\, 2
\end{array}
$$

The post-condition of *DropMeeting* eliminates the dropped meeting from calendar scheduler. The post-condition

$$
\begin{array}{|l}
\textit{meetings}' = \textit{meetings} \setminus \{\textit{abortmeeting?}\}
\end{array}
$$

The post-condition

$$
\begin{array}{|l}
\forall p: \textit{group} \\
\quad \bullet\ p \in \textit{abortmeeting?} \,.\, \textit{attendance} \\
\qquad \Leftrightarrow p \,.\, \textit{occupiedTime} = p \,.\, \textit{occupiedTime} \setminus \{\textit{abortmeeting?} \,.\, \textit{time}\} \\
\qquad\quad \wedge\ \textit{personalSchedule}'\ p = \textit{personalSchedule}\ p \setminus \{\textit{abortmeeting?}\}
\end{array}
$$

maintains the personal schedule of group members who planned to attend this meeting. For every person in the group, if this person is the attendance of the meeting being aborted, the meeting should be removed from the set of meetings (*personalSchedule*) this person attends. All the time slots this meeting occupied on this person's schedule should also be removed indicating the person will not be engaged in these time slots.

Add a participant to a meeting

A new participant can be added at any time to the participants of a meeting, provided that the new attendance must be added before the meeting ends.

AddParticipant

$PeopleOp$
$existmeeting?: Meeting$
$newperson?: Person$

$existmeeting? \in meetings$
$newperson? \in group \wedge existmeeting? \notin personalSchedule\ newperson?$
$\vee\ newperson? \notin group$
$Now \leqslant existmeeting? \,.\, time \,.\, 1$
if $newperson? \in group$
then $personalSchedule'\ newperson?$
$\quad = personalSchedule\ newperson? \cup \{existmeeting?\}$
$\quad \wedge\ newperson? \,.\, occupiedTime$
$\quad\quad = newperson? \,.\, occupiedTime \cup \{existmeeting? \,.\, time\}$
else $personalSchedule' = personalSchedule$

The operation *AddParticipant* has two input variables:

- *existmeeting?* is the meeting that the new participant wants to attend;
- *newperson?* is an element of the set *Person*, who wants to attend a meeting that is already scheduled on calendar. It doesn't matter whether the person is a member of the group or not;

The preconditions of *AddParticipant* constraint on the input variables:

- The meeting the new participant wants to attend must be in the set of *meetings* that is maintained by calendar scheduler;

$existmeeting? \in meetings$

- If the new attendant is a member of the group, the meeting he/she wants to attend shall not be already on his/her personal schedule, i.e. a person cannot attend the same meeting twice. If the new attendant is not a group member, the system doesn't examine his/her personal schedule.

$newperson? \in group \wedge existmeeting? \notin personalSchedule\ newperson?$
$\vee\ newperson? \notin group$

- The time this new participant wants to join the meeting should be no later than the time the meeting is over, that means the new participant attends the meeting before it begins or during its process, it is nonsense for a person to attend a meeting after the meeting is over.

$Now \leqslant existmeeting? \,.\, time \,.\, 1 + existmeeting? \,.\, time \,.\, 2$

If the new attendant is a member of the group whose schedule is maintained by the system, the personal schedule of meetings and time will be updated by the post condition:

if $newperson? \in group$
then $personalSchedule'\ newperson?$

$= \mathit{personalSchedule\ newperson?} \cup \{\mathit{existmeeting?}\}$
$\wedge\ \mathit{newperson?} . \mathit{occupiedTime}$
$= \mathit{newperson?} . \mathit{occupiedTime} \cup \{\mathit{existmeeting?} . \mathit{time}\}$

where the new meeting this person will attend is added to his/her personal schedule and the personal schedule is marked with the time slots this meeting occupied indicating that this person will be engaged in these new time slots.

If the new attendant is not a group member, the set *personalSchedule* and *occupiedTime* stay unchanged: the system won't maintain the personal schedule of non-group members.

else $\mathit{personalSchedule}' = \mathit{personalSchedule}$

Remove a participant from a meeting

Similarly, an existing participant can be removed from the participant list of a meeting, if the meeting has not been held yet.

The operation *RemoveParticipant* has three input variables:
- *existmeeting?* is the meeting that a participant wants to quit attending;
- *existperson?* is an element of the set *Person*, who wants to quit a meeting that is already scheduled on calendar, it doesn't matter whether the person is a member of the group or not;

```
RemoveParticipant
  PeopleOp
  existmeeting?: Meeting
  existperson?: Person
  ----
  $\mathit{existmeeting?} \in \mathit{meetings}$
  $\mathit{existperson?} \in \mathit{existmeeting?} . \mathit{attendance}$
  $\mathit{Now} \leqslant \mathit{existmeeting?} . \mathit{time} . 1 + \mathit{existmeeting?} . \mathit{time} . 2$
  $\mathit{existmeeting?} . \mathit{attendance} = \mathit{existmeeting?} . \mathit{attendance} \setminus \{\mathit{existperson?}\}$
  if $\mathit{existperson?} \in \mathit{group}$
  then $\mathit{personalSchedule}'\ \mathit{existperson?}$
      $= \mathit{personalSchedule\ existperson?} \setminus \{\mathit{existmeeting?}\}$
    $\wedge\ \mathit{existperson?} . \mathit{occupiedTime}$
      $= \mathit{existperson?} . \mathit{occupiedTime} \setminus \{\mathit{existmeeting?} . \mathit{time}\}$
  else $\mathit{personalSchedule}' = \mathit{personalSchedule}$
```

The preconditions of *RemoveParticipant* give some constraints on the input variables:
- The meeting the participant wants to quit must be already in the set of *meetings*, which is maintained by calendar scheduler;

$\mathit{existmeeting?} \in \mathit{meetings}$

- The person who wants to quit the meeting must be on the attendant list of the existing meeting he/she wants to quit.

$$existperson? \in existmeeting?\,.\,attendance$$

- The time the person wants to quit the meeting should be no later than the time the meeting is over.

$$Now \leqslant existmeeting?\,.\,time\,.\,1 + existmeeting?\,.\,time\,.\,2$$

The system updates the attendance function for the existing meeting by removing the person from its attendance set.

If the quitting attendant is a member of the group whose schedule is maintained by the system, the personal schedule of meetings and time will be updated by the post condition:

$$\begin{array}{l}
\textbf{if } existperson? \in group \\
\textbf{then } personalSchedule'\ existperson? \\
\quad = personalSchedule\ existperson? \setminus \{existmeeting?\} \\
\quad \wedge\ existperson?\,.\,occupiedTime \\
\quad\quad = existperson?\,.\,occupiedTime \setminus \{existmeeting?\,.\,time\}
\end{array}$$

where the exist meeting this person will quit is removed from his/her personal meeting list and the time slots this meeting occupied on this person's personal schedule are released which indicates that this person will be free in these time slots.

If the quitting attendant is not a group member, the sets *personalSchedule* and *occupiedTime* in the person's stay unchanged, the system won't maintain the personal schedule of non-group members.

$$\textbf{else } personalSchedule' = personalSchedule$$

Add a member to the group

The group of people whose personal schedules are maintained by the calendar scheduler can be added to new members at any time:

$$\begin{array}{|l}
\hline
\textit{AddMember} \\
GroupOp \\
member?: Person \\
\hline
member? \notin group \\
member?.\ occupiedTime = \varnothing \\
group' = group \cup \{member?\} \\
personalSchedule' = personalSchedule \cup \{(member? \mapsto \varnothing)\} \\
\hline
\end{array}$$

The input variable of *AddMember* operation, *member?,* is an element of set *Person.*

The precondition of *AddMember* operation restricts that the person joining the group should not be an existing member in the group. There should be no record of occupied time slots in the new member's personal schedule.

The post-condition of *AddMember* updates the *group* set by adding the new member in it. When a member is newly added to the group, there is nothing in his/her personal schedule of meetings and time slots engaged to meetings:

$$personalSchedule' = personalSchedule \cup \{(member? \mapsto \emptyset)\}$$

Remove a member from the group

The member in the group of people whose personal schedules are maintained by the calendar scheduler can be removed at any time:

```
──RemoveMember──────────────────────────────
│ GroupOp
│ member?: Person
├───────────────────
│ member? ∈ group
│ group' = group \ {member?}
│ personalSchedule' = {member?} ⩤ personalSchedule
└────────────────────────────────────────────
```

The input variable of *RemoveMember* operation, *member?,* is an element of set *Person*.

The precondition of *RemoveMember* operation restricts that the person leaving the group shall be an existing member in the group.

The post-condition of *RemoveMember* updates the *group* set by getting rid of the old member in it. When a member is newly removed from the group, his/her personal schedule of meetings should not be maintained by the system any longer, therefore it should be removed from calendar scheduler. We update the *personalSchedule* by domain subtraction:

$$personalSchedule' = \{member?\} \lhd personalSchedule$$

4. Issues

The part has demonstrated the feasibility of formally specifying the calendar scheduler system. Some key points are taken into consideration when specifying the system using Z, as follows:

What can this specification do?

- Maintains consistent meetings schedules for a number of people;

- Records the time, duration, and participants in each meeting includes people whose schedules are not maintained by the calendar scheduler;
- Meetings may be added or dropped at any time (up to the moment when they finish);
- Participants of a meeting can be added or removed;
- A meeting can be scheduled at any time that is convenient for all (or enough) of the meeting participants;
- The set of individual preferences on time is expressed and accommodated;
- Different personal calendar schedules are maintained;
- It is easy to define and manipulate the time of meetings, provided that a suitable time slot can be found within the expected time scope;

What is missing/difficult in this specification?

- When scheduling a meeting, there is the risk that no proper beginning time can be found according to the expected begin time chosen for the meeting, the expected time scope should be revised under this situation.

  This is one of the problems that cannot be solved by a better specification with more complicated operations, a system cannot do everything automatically, and sometimes it's necessary that the user intervenes the system operations.
- Limited by the scope of this class report, only the personal schedules of one group of people are maintained, however it is easy to maintain several different groups.
- The representations of personal calendar are not distinguished and personalized, it is possible to do so with longer time and more detailed design.
- There are chances that some aspects of a meeting (e.g. time, attendance) need to be changed instead of being simply dropped, where there's no *ChangeMeeting* operation in this specification. However this can be acquired by first dropping the meeting, then assigning new attributes to it and adding it back to the system.
- It is hard to reflect in this specification how to solve the conflicts between personal schedules and the schedules of meetings. For example, if some one wants to attend a meeting scheduled on a time slot that this person is already engaged in other meetings. Currently no solution for this problem is provided in this specification, but I am sure it is possible to solve it with Z specification without too much pain.
- When failed to schedule a meeting (cannot find a proper time), it is not clearly answered in this specification how many personal schedule mismatch the expected time scope, one or one hundred. However it is possible to tell by further working given enough time.

Part II: The CSP Specification for Calendar Scheduler

1. General ideas of the specification

Using the language of CSP, I specify the calendar scheduler system with a top-down approach, i.e. beginning with a primary system function description process as the "root", then decompose and detail the process into sub-processes corresponding to different operations on the system. Some of the sub-processes will be further divided into smaller processes and thus be described with detail, which helps better understand the specific operation.

2. The system

The Calendar Scheduler system deals mainly with meetings, time and people, it maintains the records of meetings, schedules and attendance of meetings, the personal schedule of group members and, of course the group membership. It is not absolutely necessary to define the state spaces delicately but definitions on some sets that will be used in future specification are helpful.

Set definition

There are two kinds of time concepts in my specific ation, absolute time and time duration, defined as *AbsTime* and Duration accordingly. When the system is being implemented, the format of both sets should be closer to the time format in the real world, but in a system specification the conceptual aspects serve enough.

AbsTime = {t | t is a moment}
Duration = {d | d is a time period}

The *Meeting* set is a gathering of all meetings that can be maintained by the calendar scheduler system. Every meeting in *Meeting* has a *begin time* of the type *AbsTime*, a *duration* of the type *Duration* and some *attendance* of the type *People* (defined below).

Meeting = {m | m is a meeting}

The *People* set stands for the whole universe of man-kind, under the context of this specification, it is the set of people who can attend the meetings in the system. Every person in People has a "*preferred time slots set*" and an "*occupied time slots set*", which are subsets of *AbsTime* $\times$ *Duration*, on these time slots this person is willing to attend a meeting, or is engaged in a meeting, accordingly.

People = {p | p is a human being}

The *Group* set is a group of people whose personal calendar is maintained and whose preferences towards meetings are taken care of by the system.

Group = {p | p $\in$ *People and p's personal schedule is maintained}*

The Calendar

The calendar scheduler system is initialed at the very beginning with the event *initCalendar*. After the initialization different operations can be performed on the system. The process *OPERATION* will be further decomposed in following sections.

$CALENDAR\_SCHEDULER = initCalendar \rightarrow OPERATION$

There are three kinds of operations on the system. The operation on meetings selects time slots for a meeting to be scheduled on, adds a new meeting to the calendar, or removes an existing meeting from the calendar. The operation on people makes a person to be a new participant of a meeting, removes the person from meeting attendance and updates his/her personal schedule. The operation on group adds and removes member of the group, once being removed, the person's schedule is no longer maintained by the system.

$OPERATION = MEETINGOP \square PEOPLEOP \square GROUPOP$

The *OPERATION* process is composed of *MEETINGOP*, *PEOPLEOP* and *GROUPOP.* When to perform which sub-process is up to the users so that the external choice is used here.

3. Operating on meetings

Description

The meeting process deals with the consistent schedules for a number of meetings. These schedules record the time, duration, and participants in each meeting. The scheduling of a meeting should take into consideration a common time that is preferred by the participants who are the members of the group, or at least the meeting should be scheduled at a time that all of the group member participants are free. People out of the group can also take part in meetings but their scheduling preferences are not considered. Up to the moment a meeting occurs, it can be added to or dropped from the system.

The *MEETINGOP* process

A meeting operation can be either adding a meeting to the calendar or dropping a meeting. The *MEETINGOP* process is the external choice between these two processes: *ADD_MEETING* and *DROP_MEETING*.

$MEETINGOP = ADD\_MEETING \square DROP\_MEETING$

Add a meeting

In order to add a meeting to the calendar scheduler system, we should first schedule a time for the meeting. The time slot to be selected should be based on the personal preferences of attendance that are members of the group, if no time is preferred by all group-member attendance, the meeting should be scheduled at a time that no attending group member is engaged in other meetings. After successfully selecting the meeting time, the record for the information of this meeting is added to the calendar.

$ADD\_MEETING = CHECK\_SCHED; SELECT\_TIME; ADD\_TO\_CALENDAR$

The *ADD_MEETING* process is the sequential composition of processes *CHECK_SCHED, SELECT_TIME* and *ADD_TO_CALENDAR.* When a meeting is added to the calendar scheduler system, all these processes will execute in accordance with the order as showed in *ADD_MEETING*.

The *CHECK_SCHED* process checks the personal schedules of all the group members who will attend the meeting, and gives out a preferred time set, the elements of the preferred time set are time slots preferred in common by these attending group members.

CHECK_SCHED =
participant? p: People → (NON_MEMBER □ MEMBER)

The process has an input channel *participant?*, through which the information of one potential participant is input at a time. For each person *p*, the process makes an external choice whether to execute *NON_MEMBER* process or *MEMBER* process, depending on the membership of *p*,

NON_MEMBER = not_group_member . p → (ANY_MORE □ NO_MORE)

The *NON_MEMBER* process makes a judgment that the person *p* is not a group member, which means that the *CHECK_SCHED* process doesn't care *p*'s scheduling preference. If there are meeting participants whose schedules are not checked yet, the *CHECK_SCHED* process continues; if no more checking is needed, the process ends successfully

ANY_MORE = any_more_participant → CHECK_SCHED

NO_MORE = no_more_participant → SKIP

If the person *p* is a group member, *CHECK_SCHED* process chooses to execute *MEMBER* process.

MEMBER = group_member . p → check_schedule . p → preferredtime! p . preferredtime →
occupiedtime! p. occupiedtime → (ANY_MORE □ NO_MORE)

The *MEMBER* process has two out put channels: *preferredtime!* and *occupiedtime!*. According to the information of a person *p* via the input channel of *CHECK_SCHED*, *MEMBER* process checks the personal schedule of *p*, and outputs *p*'s preferred time slots *p.preferredtime* and occupied time slots *p.occupiedtime* via the two output channels *preferredtime!* and *occupiedtime!* correspondingly. Again, if all participants' personal schedules have been checked, the process gets a successful termination, otherwise continues checking.

The *SELECT_TIME* process accepts a scope of abstract time as input via the channel *exptimescope?*, the input time scope within this scope is an absolute time set, indicating that the organizers of the meeting want it to be scheduled on some special time instead of scheduling the meeting freely. The preferred time and the occupied time slots output from *CHECK_SCHED* process are also input via the input channels with the same channel name corresponding to the output channels in *CHECK_SCHED* process.

SELECT_TIME = exptimescope? ts: $\mathbb{F}$ *AbsTime → FIND_TIME_P*

SELECT_TIME executes process *FIND_TIME_P first. T*he process looks for a suitable time that meets the requirements of both the preferred time set and the time scope set, if the event *find_suitable_time_p*, succeeds, *FIND_TIME_P* puts the select time in the value *result*, outputs the result via an output channel *meetingtime!*, and terminates successfully.

> *FIND_TIME_P = preferredtime? pt: 𝔽 AbsTime ⟶ ((find_suitable_time_p ⟶ meetingtime!. result ⟶ SKIP) □ (not_found_p ⟶ FIND_TIME_O))*

If *FIND_TIME_P* cannot find a compatible time that satisfied both the preferred time set and the expected time scope, as indicated by event *not_found_p*, the process turns into process *FIND_TIME_O*.

> *FIND_TIME_O = occupiedtime? ot: 𝔽 AbsTime ⟶ (find_suitable_time_o ⟶ meetingtime! result ⟶ SKIP) □ (not_found_o ⟶ SELECT_TIME)*

FIND_TIME_O checks the input from the other channel *occupiedtime?*, event *find_suitable_time_o* indicates that an expected time slot can be found on which all the attending group members are not engaged in meetings, and the process ends up with outputting the result via the same channel *meetingtime!.*

It is quite possible that a suitable time cannot be found during *FIND_TIME_O* process. Under this situation, the *FIND_TIME_O* process goes back to *SELECT_TIME* process, the meeting has to be re-planned by choosing some other expected time slots and do the same time selection again until a good time is found.

After successfully selecting the time slot that a meeting could be scheduled in, the meeting with its information, i.e. begin time, duration and attendance, can be added to the calendar system now:

> *ADD_TO_CALENDAR = meeting?m : Meeting ⟶ time?t : AbsTime ⟶ duration?d : Duration ⟶ check_calendar.m ⟶ (ERROR_MTN_EXIST □ ERROR_OVERDUE □ SUCCESS_ADD)*

The process adding a meeting to the calendar system inputs the meeting from a channel *meeting?*. The meeting is a component of the set *Meeting*, the begin time *t* of the meeting, scheduled with above processes, and the duration *d* of the meeting are also input via channel *time?* and *duration?* correspondingly.

Having the input information of the new meeting, the process checks the calendar. This checking procedure can be either successful or erroneous. If the process gets to know by *meeting_exist* that the meeting being scheduled is an existing meeting in the system , the process gives out an error message "meeting exists" via the output channel *error!* and returns to the process *OPERATION*. The output shows the meeting *m* has already existed.

> *ERROR_MTN_EXIST = meeting_exists.m ⟶ error! mtnexist.m ⟶ OPERATION*

If the time when this meeting is added to calendar is later than the time it is supposed to finish, it is nonsense to add this meeting to the system any more, so if the process finds that the meeting is overdue, *overdue.m*, the process gives out an error message "overdue" via the output channel *error!* and return to *OPERATION*.

$ERROR\_OVERDUE = overdue.m \rightarrow error!\,overdue.m \rightarrow OPERATION$

If the meeting is neither an existing meeting nor overdue, the *ADD_TO_CALENDAR* process gets the event *add_to_calendar.m* by marking the new meeting on the calendar. Then the process inputs the attendance set made up of group members, which should be a subset of the real attendance of the meeting, since we don't care the personal schedules of those participants who are not group members. For every person *p* in the input set, the process updates his/her personal schedule using the event *mark_on_schedule.m.p*, and terminates successfully. These processes adding meeting information to personal schedules synchronize with each other.

$$SUCCESS\_ADD = meeting\_not\_exist.m \rightarrow not\_overdue.m \rightarrow add\_to\_calendar.m \rightarrow attendance?attd : \mathbb{F}\,Group \rightarrow ||_{p \in m.\,attd}\,(mark\_on\_schedule.m.p \rightarrow OPERATION)$$

Of course there are often the needs to change the schedule of a meeting, limited by time and space, I do not specify an operation like *CHANGE_MEETING_SCHEDULE* in this report, however it is very easy to specify such kind of operation. By using the *DROP_MEETING* and *ADD_MEETING* operations specified in this report, the same functional target can be acquired with only a little more effort.

Drop a meeting

In order to drop a meeting, the *DROP_MEETING* process inputs a meeting *m* through channel *meetings?*.

$$DROP\_MEETING = meeting?.m : Meeting \rightarrow check\_calendar.m \rightarrow (ERROR\_MTN\_NOT\_EXIST \,\square\, ERROR\_OVERDUE \,\square\, SUCCESS\_DROP)$$

It checks the calendar first, if the meeting does not exist in the system, it is impossible to drop it any way. The process *ERROR_MTN_NOT_EXIST* leads an end to dropping meeting process giving out an error message "meeting does not exist" and return.

$$ERROR\_MTN\_NOT\_EXIST = meeting\_not\_exist.m \rightarrow error!mtnnotexist.m \rightarrow OPERATION$$

Similarly, if the meeting is being dropped when it is already overdue, the *overdue(m)* events gives out error message "overdue" via output channel *error!* , terminates this dropping meeting operation and returns to the beginning of *OPERATION*. The overdue process is defined in the last section.

If the meeting is an existing meeting and is not overdue, the process removes the meeting from the calendar. Then the process inputs the attendance set made up of group members, which should be a subset of the real attendance of the meeting, since we don't care the personal schedules of those participants who are not group members. For every person *p* in the input set, the process updates his/her personal schedule using the event *remove_from_schedule.m.p*, and terminates successfully. These processes synchronize with each other.

$$SUCCESS\_DROP = meeting\_exists.m \rightarrow not\_overdue.m \rightarrow remove\_from\_calendar.m \rightarrow attendance?attd : \mathbb{F}\,Group \rightarrow ||_{p \in attd}\,(remove\_from\_schedule.m.p \rightarrow OPERATION)$$

4. Operation on people

Description

The people process deals with the personal schedules for a number of people. These people should be members of the group.

The *PEOPLEOP* process

The operation on people should pay attention to the membership of this person. A people operation can be either adding a person to the attendance of a meeting, or removing the person from the participant list of the meeting. The *PEOPLEOP* process is an external choice between these two processes: *ADD_PARTICIPANT* and *REMOVE_PARTICIPANT*.

PEOPLEOP = ADD_PARTICIPANT □ REMOVE_PARTICIPANT

Add a participant to a meeting

The process adding a participant to a meeting inputs the person *p* from the input channel *input_people?*. The person can either be a group member or not. A meeting *m* that this person wants to attend is also an input of the process via the channel *input_meeting?*.

ADD_PARTICIPANT = input_people? p: People → input_meeting? m: Meeting → check_calendar.m.p → check_participant.m.p → (ERROR_MTN_NOT_EXIST □ ERROR_OVERDUE □ ERROR_PARTICIPANT_EXIST □ SUCCESS_ADD_PARTICIPANT)

Based on the information of the two inputs, the process checks the calendar, if the meeting is overdue, the process gives out the error message “overdue”, then returns. Similarly, if the meeting isn’t in the system at all, “meeting doesn’t exist” error will be given out, process returns (these two error handling processes are defined in previous sections); if the person is already a participant of the meeting, the process returns with error “participant exists”. All the error messages are given out via channel *error!*.

ERROR_PARTICIPANT_EXIST = participant_exists.m. p → error! pttexist → OPERATION

If the both the meeting and the participant are good enough that they satisfy all the requirements, i.e. the meeting is neither overdue nor non-exist in the calendar, and the person is not already a participant, then the process add this new participant to the meeting.

SUCCESS_ADD_PARTICIPANT = not_overdue. m → meeting_exists. m → participant_not_exist.m. p → add_participant. m. p → (NON_MEMBER □ ADD_MEMBER)

If the person is not a group member, the process terminates successfully without further action; if the person is a group member, the process mark the meeting and its parameters to the personal schedule of this new participant.

$$\mathit{NON\_MEMBER} = \mathit{not\_group\_member}.\ p \rightarrow \mathit{OPERATION}$$

$$\mathit{ADD\_MEMBER} = \mathit{group\_member}.p \rightarrow \mathit{mark\_on\_schedule}.m.\ p \rightarrow \mathit{OPERATION}$$

Remove a participant from a meeting

The process removing a participant from a meeting inputs the person *p* from the input channel *participant?*. The person can either be a group member or not. A meeting *m* that this person wants to attend is also an input of the process via the channel *meeting?*.

$$\begin{aligned}&\mathit{REMOVE\_PARTICIPANT} = \mathit{participant?}.\ p\!: \mathit{People} \rightarrow \mathit{meeting?}.\ m\!: \mathit{Meeting} \rightarrow\\ &\quad \mathit{check\_calendar}.m \rightarrow \mathit{check\_participant}.m.p \rightarrow (\ \mathit{ERROR\_MTN\_NOT\_EXIST} \ \square\\ &\quad \mathit{ERROR\_OVERDUE} \ \square\ \mathit{ERROR\_PARTICIPANT\_NOT\_EXIST} \ \square\\ &\quad \mathit{SUCCESS\_REMOVE\_PARTICIPANT}\ )\end{aligned}$$

Based on the information of the two inputs, the process checks the calendar, if the meeting is overdue, the process gives out the error message “overdue”, then returns. Similarly, if the meeting isn’t in the system at all, “meeting doesn’t exist” error will be given out, process returns; if the person is not a participant of the meeting, the process returns with error “participant doesn’t exist” therefore the participant cannot be removed. All the error messages are given out via channel *error!*. Except for process *ERROR_PARTICIPANT_NOT_EXIST* defined below, other error outputting processes are defined in previous sections.

$$\begin{aligned}&\mathit{ERROR\_PARTICIPANT\_NOT\_EXIST} = \mathit{participant\_not\_exist}.m.\ p \rightarrow\ \mathit{error!}\ \mathit{pttnotexist}\\ &\quad \rightarrow\ \mathit{OPERATION}\end{aligned}$$

If the meeting is neither overdue nor non-exist in the calendar, and the person is a participant to this meeting, then the process remove this new participant from the attendance of the meeting. If the person is not a group member, the process terminates suc cessfully without further action; if the person is a group member, the process get rid of the meeting and its parameters from the personal schedule of this quitting participant.

$$\begin{aligned}&\mathit{SUCCESS\_REMOVE\_PARTICIPANT} = \mathit{not\_overdue}\ .\ m \rightarrow \mathit{meeting\_exists}.\ m \rightarrow\\ &\quad \mathit{participant\_exists}.m.\ p \rightarrow \mathit{remove\_participant}.\ m.\ p \rightarrow (\ \mathit{NON\_MEMBER} \ \square\\ &\quad \mathit{REMOVE\_MEMBER})\end{aligned}$$

$$\mathit{REMOVE\_MEMBER} = \mathit{group\_member}.p \rightarrow \mathit{remove\_from\_schedule}\ .\ m.\ p \rightarrow \mathit{OPERATION}$$

5. Operation on group

Description

The group process deals with the membership of people.

The *GROUPOP* process

The operation on group adds a person as a new member of the group, or removes the person from the group. The *GROUPOP* process is the external choice between processes *ADD_MEMBER* and *REMOVE_MEMBER*.

$GROUPOP = ADD\_MEMBER \; \Box \; REMOVE\_MEMBER$

Add a person to the group

This process adds a person to the group, after the person has the new membership, his/her personal schedule will be maintained by the system.

The process inputs the person *p* from the input channel *input_people?*. If the person is an existing member of the group, the process gives out the error message "member already exist" and returns. For a new member, the person is added to the group and the process terminates successfully.

$ADD\_MEMBER = input\_people?\ p: People \rightarrow (\ (member\_exist.p \rightarrow error\_mbrexist!\ p \rightarrow OPERATION) \; \Box \; (member\_not\_exist.p \rightarrow add\_to\_group.\ p \rightarrow OPERATION)\ )$

The personal preferences and engagements of the new group member are not input explicitly here, by default, these personal schedule-related things are maintained.

Remove a pe rson from the group

This process removes a person from the group, the person should be an existing group member but after the person quits the group, his/her personal schedule is no longer maintained by the system.

The process inputs the person *p* from the input channel *input_people?*. If the person is not an existing member of the group, the process gives out the error message "member doesn't exist" and returns. For an old member, the person is removed from the group and the process terminates successfully.

$REMOVE\_MEMBER = input\_people?\ p: People \rightarrow (\ (member\_not\_exist\ .\ p \rightarrow error\_mbrnotexist!\ p \rightarrow OPERATION) \; \Box \; (member\_exist.p \rightarrow remove\_from\_group.p \rightarrow OPERATION)\ )$

6. Issues

This part has demonstrated a specification of a Calendar Scheduler system using the language of CSP. For the sake of comparing these two kinds of specification on the same system, the operations in this CSP specification is intentionally written in the same way as in the previous Z-spec part, although the CSP approach can be written otherwise.

What does this specification do?

- Maintains the schedules and attendance of meetings, maintains personal schedules of a group of people, while the attendance of meeting can be non-group-member;

- Chooses a suitable time to arrange a new meeting, maintaining the personal preference of group members and avoiding conflicts;

- Adds or drops meetings if the meeting is not overdue at the time; adds or removes participants of a meeting, no matter they are group member or not; adds or removes people from the group;

- Gives out detailed error messages whenever an error or an abnormal event occurs;

- The processes of how an operation works is given in detailed in each sub-processes of the specification.

- The concept of parallel processes is well expressed when adding/removing a meeting to/from many people's schedule, although there are limited number of processes in this system that can be paralleled.

What is missing/difficult in this specification?

- The number of "enough" participants is not explic itly given, also, it could not be seen from this specification how the personal scheduling preference is like. The specification just tells us there are personal scheduling preferences according to which to arranging meeting time, but we don't know in deta il how the preferences looks like. There is no process on adding/changing/removing the personal preferences, too, given enough time it is not difficult.
- If a suitable time cannot be found, this specification requires iteratively input expected time scope until a time is found, this may lead to endless loop of the system if expected time is indeed cannot be changed;

- It is hard to express the concept of storage in CSP, i.e. the state space in Z, sometimes it will make things clear if we have state spaces in CSP. For example, when we check the calendar to give out the preferred time of all the participating group members, we can only say: perform this process until all members' scheduling preferences are covered and output. There is no way to know how the personal preferences will be like if we put all of them together and make an intersection, since the outputs are merely flowing to the next process;

- It is hard to express the idea that two processes are in sequential no matter the first process terminates successfully or not, the sequential operator (;) only permits the first process to end with success. It is clumsy to have a long queue of events.

Part III: Comparison of Z and CSP Specification

A comparison of the two specifications is presented in this part, where the pros and cons of each approach are analyzed and compared with the other. The common advantage and weakness of these two approaches are presented. Some comments on the methods learned during the course are also briefly stated.

1. Z versus CSP in the context of the same system

The Calendar Scheduler System itself is not complicated, but it has some practical features that can raise interesting questions when specifying the system, especially from different points of view of different specifying methods. In this section, approaches specifying the same system features using Z and CSP are compared, the advantages of one approach over the other are presented. In the next sections, the overall strongpoint and weakness of the two specification methods are analyzed.

1.1 Advantage of Z over CSP

Delicate description of system states

Every software system can be intuitively modeled and understood as state machines, the basic ideas of state machines is the changing of states in the system and the transitions between states. As a formal method, the Z notation can by nature provide precise, unambiguous descriptions of the states when specifying a system. The basic infrastructure in Z is schema, a pattern of declaration and constraint. When specifying a system with Z, schemas are used to describe the states of the system, and the ways in which the states may change.

The schemas can also be used to describe system properties. In Z, constraints on the changing of states changes are reflected by pre-conditions and post-conditions on state variables. The pre and post conditions draw a limit to the operations causing system to change states that some particular conditions are needed to make this operation, and the operation will not lead to unexpected system actions.

Having the delicate description of system states, we get a clear idea on the composition of the whole system, which can lead to further design details (with an object oriented approach) such as the demarcation of objects, the definition of data structure, local and global variables, etc.

In contrast, CSP doesn't provide any mechanism to describe the states in the system. From a CSP specification it is impossible to know what the objects the processes are working on, and how the processes affect the characters of them. From this point of view, CSP is less feasible than Z for a possible object oriented system design and implementation in the future.

Mathematical representations

In Z, mathematical languages based on set theory, prepositional and predicate logic are widely used to state the problem, to discover solutions and to prove that the chosen design meets the specification. The nature of mathematics provides Z the ease to restrict, map and prove the design, at the same time providing a useful link to programming practice. For example, a characteristic feature of Z is the use of types, with the mathematical language used in Z, every object has a unique type, represented as a maximal set in the current specification. This notion of types means that an algorithm can be written to check the type of every object in a specification. Another example is the mapping used in Z from one set to another, which gives a precise idea of the relationships between system objects for later design and implementation stages.

Although the notations in CSP are more intuitive than in Z, they don't facilitate the describing of detailed objects construction and communication (very possibly) needed for future design. There are notations such as input and output, channels and alphabetized process operations, but precise mathematical notations lack very much.

No risk of introducing deadlock

It is true that Z is not intended for concurrent behavior, but no deadlock will be introduced because of this feature of Z. In Z, the way making sure the state changes is to use state invariants defined in the state space of the system. Once an operation is consistent with the state invariants, we can say that the state operations satisfy expects on the system. No further interacting or paralleling of these operations are needed. Thus deadlocks won't be introduced. While in CSP, it is very easy to have two deadlocked processes when they parallel with each other.

Clear segments leading to less brain work

This may not be a functional key point in modeling a system but it helps if a specification is clear enough for people with little idea of the intricate structure of the system to understand without too much pain what's going on in the specification. In Z, the distinction among schemas are clear, when you are reading one schema, you may only focus on the content of this very schema, without thinking too much on the other schemas. Even if there is some common operation shared by other operations, it is still not hard to understand the operation schema using this common operation. In CSP, due to the nature of process definitions, a process will inevitably parallel with, or become part of the internal/external choices for other processes. Even a process as simple as the process *SHOP* defined in [3] takes me quite a while to understand how the events will be like when all the sub-processes parallel together. As is mentioned in last item, deadlock will be introduced with great ease this way.

1.2 Advantage of CSP

Support of concurrent behaviors

The biggest advantage of CSP over Z is the support of concurrent behaviors. Although in the calendar scheduler system I specified in part II only a couple of processes are in parallel with other processes, it is indeed the most important feature of CSP, which is not supported at all in Z. In the real world, a very large portion of systems have concurrent behaviors, these processes synchronize by sharing system resources or communicating with one another. If not handled properly, the introduction of parallel will either lead to combinatorial explosion or dead lock. CSP puts a limitation on the parallel of processes to (some of) the events these processes have in common, which models the processes executing at the same time. The FDR checker makes sure that the parallel processes won't deadlock.

Detailed operation steps described by events

In Z, how an operation makes changes on the states of the system is presented by pre-conditions and post-conditions. Though it is true that a more rigorous specification can be built via the consistency of operations over pre and post conditions, only WHAT the operation does on the state is modeled. In CSP HOW the processes operate on the system is presented by events. The events can be from a skeleton of system behaviors to very detailed operation steps. By this means, how the behaviors are like in the systems are very obvious.

Easier to understand for people with less mathematical background

The events in CSP processes are written with a natural-language-like way, e.g. leave, shop, etc. There are not too many notations that have abstruse meanings on sets or functions as used in Z, even a people with few mathematics background can understand what is going on in the process with great ease, provided there are not too many intricate parallel interactions between processes, while in Z, understanding a schema needs good knowledge of the symbols used.

Clear clue of inputs and outputs among processes

The concept of channel in CSP makes it possible to know where the outputs from a process will go to another process, if applicable. Two processes are connected like a "pipeline" by using the same channel name. In Z, this input/output connection between operations is not explicitly shown.

Internal choices

A remarkable feature of CSP is the using of internal choices. In all software systems there exists non-determinism, under which circumstance the user has no control over the choice the system will make. In the context of the calendar scheduler system though, this feature is not so obvious that a sound comparison can be performed. But internal choice does provide a good representation of modeling non-determinism.

2. Sharing features of Z and CSP

2.1 Common advantage

Both methods have refinement

Good understanding of basic system characteristics can be gained from a simple description, but we may also wish to develop a specification in such a way that it leads us towards a suitable implementation, which is more favorite from a practical points of view. Without a good mechanism leading to implementation, a "good" specification language is only something interested by researchers. Fortunately both Z and CSP have refinement, by which more precise information can be added to the primitive but correct specification. Whether the detailed description is consistent with the original one should be proved in both Z and CSP. The process of improvement brought by refinement also involves the removal of non-determinism.

Both methods have supporting tools

It facilitates me greatly using Zed and FDR as syntax and proof checkers for my specification. With the help of the tools, the consistency of my specification is checked, some of the defects mentioned above, such as deadlock, can be avoided.

2.2 Common weakness

Too much natural language explanation

Both Z and CSP need a great deal of natural language as the explanation of the schemas and processes, which should be the REAL main part of a specification. It seems that without the bulky explanations neither specification will make better sense or even be correctly understood. It is announced in the book of Woodcock and Davies [3] that one of the advantages of Z specification is that they use natural language to relate the mathematics to objects tin the real world, and a well-written specification should be perfectly obvious to the reader. In my opinion a good specification should by itself contains enough information at the same time being abstract.

None of them specify non-functional system aspects

There are other aspects of a system that worth paying attention to when designing, the usability, performance, size and reliability of the system. Is the system efficient? Is it easy to use? How much computing resources does the system use? Will there be endless loop? Neither Z nor CSP answers this kind of questions. Both of them provide good modeling methods for the functional areas of the system, but we cannot draw a conclusion from the specification. Further analysis should be performed on the missing aspects of them.

Little support for transit to implementation

We have to admit that the aim of software development is to create an executable and useful software system. A good system specification presents an appropriate level of abstraction and can be used to support the design process, and as a guide to subsequent development, testing, and maintenance. Specification for a large, complicated system costs considerable effort of system designers, the result of a specification should not be only a documentation for future reference, but be the assistance and origin of subsequent development stages as well. Based on my experience on Z and CSP, I don't see enough support in both for a smooth transition without too much human effort from the specification document to a prototype of the system. In a word, neither language makes good use of the abstraction result it has gained for the next development step, we have to transit our mind to other detailed design languages such as UML for a deeper look at the system.

Experience based excellence

I noticed during my working on specifying the calendar scheduler system that the more time I spent on studying the modeling language, the more free I felt specifying the system. There are still much more aspects to be paid attention to, and might have been specified in other better ways, and revealing more out of the system, provided I had years of experience working on these languages.

3. Comments on modeling languages thus far

Throughout the course I have learned quite a few notations that can be used as modeling language for my system. Each of them, when used to model a system at the abstraction level, can represent some aspects of the system while meager in other aspects. Z manipulates the state changes in the system but is weak in describing processes; CSP models mostly on processes while pays no attention to state status; Petri Nets gives intuitive graphical way of modeling concurrency and system execution but it is sort of painful to draw intricate graphs for a large system; Larch and Algebraic Specifications give a taste closer to implementation but leave less space for abstraction and proof. My concern is how to utilize these specification languages more than they are currently being used into the design of real systems. A combination of several languages used for a comprehensive description of different aspects of a system, like the using of different kinds of diagrams in UML, seems to be a good idea for me.

4. Reference

[1] Mary Shaw, David Garlan, etc. Candidate Model Problems in Software Architecture, January 1995
[2] Jim Woodcock and Jim Davies, Using Z Specification, refinement, and proof.
[3] Steve Schneider, Concurrent and Real-time Systems, The CSP approach.