Survey of Labour and Income Dynamics (SLID) Project at Statistics Canada

Richard Chan
Operations Research and Development Division
Statistics Canada

Background

Statistics Canada has been conducting CATI social surveys for a long time. However, most of the surveys are not truly CATI. The interviewers just phone the respondents and conduct a CAPI survey. There is no callscheduler to handle the distribution of cases. Other CATI surveys are not in Blaise. Recently, the bureau decided that all CATI social surveys should be implemented in Blaise with callscheduler to provide full CATI functions. As a result, we have had to redesign the infrastructure and the survey applications to accommodate all the changes.

Survey of Labour and Income Dynamics (SLID) was chosen to be the first survey implemented in full Blaise. It is a longitudinal survey designed to follow the same respondents for several years to get a long-term picture of how changes affect people financially. Interviews for SLID are conducted twice a year, in January and in May, for everyone in the sample who is aged 16 or over. Like many other social surveys, SLID consists of Entry and Exit questions, Demographic and Relationship questions, and subject matter questions. In January, the subject matter is Labour. Currently, some of the social surveys are implemented in Visual Basic and Blaise. The Demographic and Relationship part is done in Visual Basic, and the subject matter component is done in Blaise.

Design

The redesign of the SLID application is a challenge. Not only do we have to replicate the functionality of the existing application, we are also required to replicate the "look and feel". Since the survey is divided into three parts, there are two possible designs for the application. One is the single datamodel approach, where we combine all three parts into a single datamodel, like a typical Blaise application. The other is the multi-datamodel approach, where we keep the three parts separated, and use three separate datamodels to implement a single application.

The single datamodel design is obviously the easiest way to implement the application. Technically, it will capture all the required information for the survey, so there is no problem in terms of functionality. However it does not quite satisfy the "look and feel" requirement. Currently, the Visual Basic portion of the application displays a list of members who are the candidates for the Labour component at the end of the Demographic and Relationship (DemRel) component. This list is called the component list. In general, it contains all the subject matter components for each household member. In the case of SLID, there is only the Labour component for each member. The interviewer can select a member from the list and start the Labour component. When the Labour component of that member is finished, the application will return to the list and the selection process is repeated until the interviewer is ready to stop. After that, the application will proceed to the Exit component. The other concern we have is the size of the datamodel. Since we have a roaster of forty members and each of them could be a candidate for the Labour component, the datamodel would contain forty Labour question blocks. Even if handling huge datamodels is not a problem, it is still a waste of space. For a household with only one member, we will have one block of information and thirty-nine blocks of empty space.

The multi-datamodel design is more complex, however it provides more flexibility. With this design, we separate the Labour component from the DemRel component. We can execute certain functions before switching from one component to the other, which makes displaying a component list possible. Since Labour is separated from DemRel, we no longer have to reserve space for potential Labour records. We will have a record in the Labour datamodel when it is necessary. A household with only one candidate for Labour will only have one record in the Labour database. Although we have more than one datamodel, the size of each datamodel is now smaller. With the multi-datamodel design, we can make the datamodels independent of each other. We can modify and prepare one datamodel without affecting the others. It is actually one of the requests made by the social survey clients. Once the subject matter component has been tested and accepted by the clients, that component should not be modified and prepared again. This is not possible by using a single datamodel. If DemRel and Labour are in the same datamodel, we have to re-prepare the whole datamodel regardless of what changes we are going to make. When we separate them into two datamodels, we can modify and prepare the DemRel part without touching the Labour component. The problem with the multi-datamodel design is passing information from one datamodel to another. For SLID, only respondents aged 16 or over have a record in the Labour database. However, the age is stored in the DemRel database. Also, the flow of the questions in the Labour component is based on the responses to questions in the DemRel component. In order to pass information, we will need a shell program to manage the execution of each datamodel, and the extraction of data from one to the other.

Both designs have their advantages and disadvantages. The single datamodel design is easier to implement. We have all the needed information in one database. Extraction and population of the database will be simple and it does not require a complicated shell program to facilitate the communication among databases. It is self-contained, so it may be easier to maintain and support. However, everything has to be done within the Data Entry Program, which does not provide much flexibility, especially in terms of user interface and handling multiple subject matter components. With this design, certain functions such as the component list cannot be implemented, many of the "look and feel" requirements cannot be satisfied. With the roster size of forty, the database will be very large. The multi-datamodel design, on the other hand, is more complicated and is more difficult to implement. It requires a shell program to drive the whole application. It has to control when to call which datamodel, and extract and route data from one datamodel to the others. The shell program has to be carefully coded in order to keep all datamodels in sync. Nevertheless, this design allows us to display the component list between datamodels, to divide the survey application into distinct components, and to treat each component individually. Since both designs will get the job done, and the "look and feel" requirement is a very important aspect of the application, we decided to choose the multi-datamodel design.

Implementation

Like many other social surveys, SLID can be divided into four different parts: Entry, Demographic and Relationship, Subject Matter (in this case, Labour), and Exit. Although modulising the survey gives us more flexibility, we do not want to over do it as it is may complicate the implementation. We know that Labour will be a separate datamodel by itself. Entry is relatively small and is followed by Demographic and Relationship. Since some surveys do not capture Demographic and Relationship at all, it is easier to combine the two parts together and call it EntryDemRel. Exit is the last part of the survey, and it will be a separate datamodel. We cannot combine Labour and Exit, because Labour is at the person or component level, and Exit is at the household or case level. Since each household in SLID can have many components, there are two levels for each case, household level, and component level. There should be only one Exit for each case, regardless of how many components it has. There is an outcome code associated with each level, which indicates the status of the case and its components.

Now that we have divided the survey into three datamodels, we have to decide which datamodel should inherit CATI. The datamodel that inherits CATI should be the entry point and the exit point of a case. We need the callscheduler to deliver cases, and assign treatments to cases upon exit. It seems logical to have EntryDemRel inherit CATI, since it is the first part of the survey. However, we do not exit through

EntryDemRel, but rather exit through the Exit datamodel. We have to assign a treatment to a case, or make an appointment, and assign an outcome code whenever we exit a case. Therefore, the Exit datamodel has to inherit the CATI function.

The core part of the survey is now divided into three datamodels called CallSchedulerExit, EntryDemRel, and Labour. The CallSchedulerExit datamodel inherits the CATI functions and is the entry and exit point of the application. It also contains all the Exit questions for the survey. The EntryDemRel datamodel contains the Entry, Demographic, and Relationship questions for the survey. Although CallSchedulerExit is the starting point of the application, it is in the EntryDemRel datamodel where the actual interview begins. This is why the Entry questions are implemented in this datamodel. The Labour datamodel contains all the subject matter questions of SLID. We also need a datamodel to implement the component list. For each household, the component list database stores a list of components and their outcome codes. We call this datamodel ComponentList.

Here is the overview of the application. We start with the CallSchedulerExit datamodel, which inherits CATI. As it only contains the Exit questions, once we have the case id, we use it to call the EntryDemRel datamodel to start the interview while CallSchedulerExit remains open. When EntryDemRel is finished, the ComponentList database will be updated based on the household information in EntryDemRel. The person id of each household member who is a candidate for the Labour component is stored in ComponentList. A list of the members is displayed, hence the required component list. From the component list, we pick one of the members and use the person id to call Labour. When the Labour component of that member is finished, we return to the component list and repeat the same steps for each remaining members. When we exit the component list, we return to CallSchedulerExit. Whether the case can be finalized depends on the combination of the outcome codes for each component of that case. If the case can be finalized, an outcome code will be assigned automatically, and the Exit questions will be asked. Otherwise, the interviewer will have to assign an outcome manually, which can be final or in-progress. The Exit questions will be asked to complete the case, or the interviewer will assign a treatment or make an appointment to exit the case.

Basically, there are two parts of the application. One is within CallSchedulerExit; the other is outside CallSchedulerExit. When we are outside CallSchedulerExit, we are working with EntryDemRel, ComponentList, and Labour. These datamodels have to pass information to the others. The component list is created based on the age of each member from EntryDemRel. The Labour component is activated by the person id in ComponentList. It also needs other information from EntryDemRel, such as the roster of the household. Since Labour is a proxy questionnaire, any member can answer the questions for any member in the household. Certain flags are also stored in EntryDemRel, which control the flow of questions in Labour. At this moment, the only way to read from, and write to a Blaise database and to create a user interface is by using Manipula/Maniplus. Therefore, we need a Maniplus program to call and manage these three datamodels.

The Program Manager

The Program Manager is the Maniplus program that manages the three datamodels. Its functions include:

- call the EntryDemRel datamodel with a specific case id;
- create a new record in CallSchedulerExit, and EntryDemRel if the household splits;
- read from EntryDemRel to get information for creating the component list of the case;
- update the ComponentList database using the information from EntryDemRel;
- create a lookup table to display the component list of the household;
- call the Labour datamodel with the person id chosen from the component list;
- pass information from EntryDemRel to the selected Labour record;

- read the outcome code of the component after Labour is finished;
- update the outcome code in the ComponentList database.

Essentially, the Program Manager is the main part of the survey. It controls all the major datamodels of the survey except the Exit part. The CallSchedulerExit datamodel calls the Program Manager to start the interview. When EntryDemRel and Labour are done, the Program Manager is terminated, and it returns to CallSchedulerExit to start the Exit component.

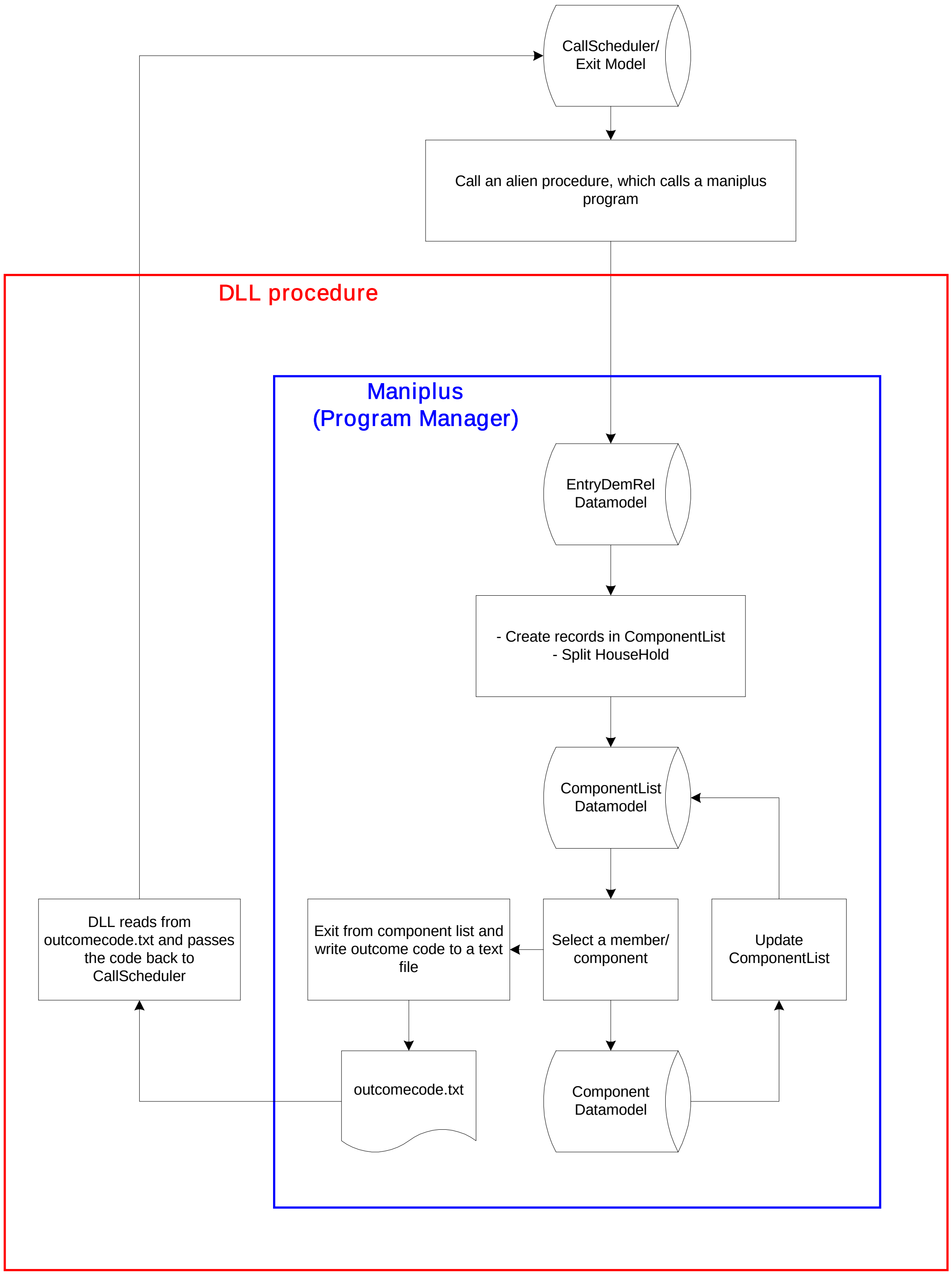
Controlling the Program Manager
Since the application begins with CallSchedulerExit, we need a way to call the Program Manager from CallSchedulerExit. We can simply use a function key to execute an external program. However, it is difficult to control the sequence of events. We only want the Program Manager to be executed once at the beginning of the application. By using the function key, it can be executed whenever the function key is pressed, which is not what we want. Also, we have to pass certain information from CallSchedulerExit to the Program Manager, which cannot be done with the function key. Moreover, the clients do not like the idea of pressing a function key to start EntryDemRel. They would like to see the application to continue with the normal flow of the questions. Function keys should be used for specific tasks that deviate from the application, instead of continuing with the next part of the application.

The other way to call the Program Manager is to use a DLL procedure. It is easier to control a DLL with an alien procedure in the application. We can simply set a flag in the application to make sure the procedure will only be executed once. Responding to a specific question in the application can trigger the alien procedure, appearing like we are following the normal flow of the questions, rather than starting a new process. We can also pass any fields in CallSchedulerExit to the alien procedure as parameters, so that they can be used in the DLL procedure and the Program Manager. The DLL procedure also allows us to change the values of the parameters, i.e. updating those fields in CallSchedulerExit. Updating a field of the current form with an external program is not possible because the form is locked by the DEP. The only way to do it is to use a DLL procedure. The field that we need to update is the outcome code, which can be assigned automatically based on the outcome codes of the components of that case. We can update other fields the same way for other surveys in the future. For now, we will only update the outcome code.

How does it work
The DLL procedure is not complicated. Its main function is to call the Program Manager with some parameters. For SLID, we are passing household id, outcome code, interviewer language, and respondent language as the parameters. In CallSchedulerExit, filling in the first field will call the alien procedure that activates the DLL procedure. The DLL procedure uses the parameters to call the Maniplus program, Program Manager. Program Manager uses the household id to call the corresponding case in EntryDemRel. Using the interviewer language and the respondent language, the corresponding question texts and menu will be used to open the case in EntryDemRel. The DEP for EntryDemRel is then opened for data entry. When EntryDemRel is finished, it returns control to the Maniplus program. Using the household id as the primary key, the Maniplus program reads from EntryDemRel. After reading the case from EntryDemRel, the Maniplus program has to do two things. First, it determines if anyone moves away from the household by checking the roster information in EntryDemRel. If someone moves away, it splits the household by adding a new case in CallSchedulerExit and EntryDemRel in order to create a new household. Second, it checks the demographic information to see who is eligible for the Labour component, and then updates the ComponentList database with the person ids of all the candidates. A lookup table window, ComponentList, showing all the candidates in that household will then be displayed. The interviewer picks a member from the lookup table, and the Maniplus program uses the person id as the primary key to call Labour. The language parameters from the DLL procedure will allow the DEP to use the proper menu and language texts. After Labour is finished, the Maniplus program

regains control and reads the Labour database to get the outcome code of the Labour component. It updates the outcome code in the ComponentList database, and displays ComponentList with updated information. The same process is repeated until the interviewer exits ComponentList. At this point, the Maniplus program uses the household id as the secondary key to search for all component outcome codes for that household in the ComponentList database. The outcome code for the household, if applicable, is determined by the combination of all the component outcome codes. The household outcome code is written to a text file, and the Maniplus program is terminated. The DLL procedure regains control and reads from the text file to get the household outcome code. It updates the outcome code parameter and returns to CallSchedulerExit. After it returns to CallSchedulerExit, it goes to the Exit block and asks the proper questions to finish the case. When the case is closed, the callscheduler delivers another case, and repeats the whole process. The diagram below (figure 1) shows the sequence of events for the process.

Figure 1. SLID (Jan 2001)


CallScheduler/ Exit Model
Call an alien procedure, which calls a maniplus program
DLL procedure
Maniplus (Program Manager)
EntryDemRel Datamodel
- Create records in ComponentList
- Split HouseHold
ComponentList Datamodel
DLL reads from outcomecode.txt and passes the code back to CallScheduler
Exit from component list and write outcome code to a text file
Select a member/ component
Update ComponentList
outcomecode.txt
Component Datamodel

Other parts of the application

The Program Manager is the core part of the application that collects data for the survey. However, we also need other functions to support and complete the application. For example, being able to display demographic information, and call summary at anytime during the interview. Also, the ability to view and edit notes at anytime. These functions can be implemented using Maniplus. For demographic information, we use a Maniplus program to read from EntryDemRel and display the information in a lookup table. For call summary, we read the CatiMana block from CallSchedulerExit and display the information in a lookup table. We use a separate Blaise database to store the notes. Since we have to be able to edit as well as view the notes, it is easier to use a separate database. If we put the notes on one of the main datamodels, we will be able to view the notes of the current case with a Maniplus program, but we will not be able to edit and save the notes because the DEP is locking the current case. So we use a separate database to implement the notes, and use a Maniplus program to access the notes database. These three Maniplus programs are assigned to three different function keys, F3 for demographic, F8 for call summary, and F11 for notes. The household id is passed as parameter when calling the Maniplus program. It uses the household id as the primary key to access the information for that household. Unlike the Program Manager, we do not have to call these three programs using a DLL procedure. We want to be able to call these programs whenever we want, and we do not have to update any information when we return from the programs to the main application. Therefore, it is not necessary to use a DLL procedure to control the Maniplus programs. We can simply use the function keys to call the programs directly.

There are two other databases that are important to the application. One is the Users database and the other is the Routing database. The Users database stores the id, the main group and the other groups for each interviewer. The Routing database stores the routing rules for each outcome code. When an interviewer from a certain group exits a case with a certain outcome code, that case could be routed to another group of interviewers, or routed back to home if it is finalized. For example, if an English case is assigned the outcome code that indicates the respondent prefers the other official language, that case will be routed to the French interviewers. Routing is accomplished by changing the ToWhom field. If a case has to be routed to a specific group, the ToWhom field will be changed to the appropriate group name. If a case has been finalized, the field will be changed to "Home". Each routing rule consists of the survey id, the original group, the status, the outcome code, and the destination group. The application uses the survey id, the original group, the status, and the outcome as the primary key to search the Routing database and retrieve the destination group. It uses the destination group to update the ToWhom field. Each case is assigned to a certain group, which is indicated by the ToWhom field. Changing the ToWhom field makes the case accessible to the proper group of interviewers. In the application, CallSchedulerExit reads the external database, Users, to get the main group of the interviewer. After an outcome code has been assigned to the case, it reads the external database, Routing, to get the destination group for the ToWhom field. The ToWhom field of the case is used as the original group when searching the Routing database. However, that field may contain the interviewer id instead of the interviewer group, if the manager assigns the case to a specific interviewer using the CATI Management program. This is why we have to read the Users database to get the main group of the interviewer. If the ToWhom field is the interviewer id, there will be no match in the Routing database. So we use the interviewer main group as the original group to do the search instead.

The last part of the application is the UpdateDaybatch program. It is a Maniplus program that puts certain cases back to the daybatch. Normally, the Blaise callscheduler will put a case back to the daybatch if it is Busy, No Answer, Answering Service, or Appointment. However, many of our cases do not fall into those four categories. Cases such as tracing required, language barrier, refusal, request for interview by another interviewer, etc. They should all return to the daybatch in the same day with default priority. Assigning these cases to one of the four categories will not be appropriate because those treatments will assign certain priority to a case. We want those cases to come back as if they were untouched cases. Since currently there is no treatment in Blaise to do what we want, we have to create our own solution. In the application, we treat all the special cases as Others. The UpdateDaybatch program reads the CallSchedulerExit database to check if a case has been treated as Others. If the last dial result is Others, and the ToWhom field is not “Home”, that means the case has not been finalized and should be returned to the daybatch. The program uses the daybatch_add function to add the Others cases back to the current daybatch. By running the program in a regular interval manually or automatically, the daybatch is being updated constantly. All the cases that have not been finalized would be accessible through the Blaise callscheduler.

The Results
After almost a year of analysis, design, implementation and testing, the SLID application was made available January 2001. The application was deployed to all seven regional offices, and production began with the first full Blaise CATI application. Like any new applications, SLID did encountered some problems.

One was a delay between CallSchedulerExit and Program Manager. When CallSchedulerExit activated the DLL procedure to call Program Manager, there was a delay about 10 to 30 seconds. The reason for the delay was the initialization of the Maniplus program. Program Manager controlled the communication between all the datamodels, so we had to put them all in the Uses section of the program. Every time Program Manager was called, it had to load the meta-info of all the datamodels, thereby creating the delay. The duration of delay increased as the network traffic increased. We tried to use the filter setting to filter out part of the datamodel, but there was no obvious improvement. We were told that if a Maniplus program uses all the datamodels, and it calls a second Maniplus program that uses the same datamodels, the process of initialization of the second Maniplus program would be much shorter. However, it did not work in our case. We used a Maniplus program to call CallScdehulerExit, but that same program did not call Program Manager. It was called by a DLL procedure within CallSchedulerExit. Therefore, we had two separate Manipula processes running. Opening all the datamodels in the first Maniplus program did not benefit the second one in this case. Fortunately, the interviewers were supposed to phone the respondents after Program Manager started EntryDemRel, so there was no interruption during the interviews. However, it is still a technical problem remaining to be solved.

Another problem was in CallSchedulerExit. After the application returned from Program Manager to CallSchedulerExit, we started to ask the Exit questions. Sometimes there was a few seconds delay between questions in the Exit block. Later we discovered that the reason for the delay was the external read to the Users and the Routing database. In the early stages of development, the watch window had not been introduced as a debugging tool, so there was no way to tell when and how the external database was read. After the watch window became available, we obtained a better understanding of how the external database was accessed. We improved the situation by changing the way that we did the external read. The delay remained however not as serious as before. We suspect that there is a pending search for any unsuccessful search on the external database. Basically, the application would not stop the external read until it successful read something from the database. This is only our theory, and we are still investigating this issue.

There were other problems that were not related to the design or the implementation of the application. Sometimes the application froze on one workstation, and the whole system would start to slow down. Eventually, other workstations seemed to freeze as well. When this happened, we found multiple entries of the same case in the history file (.BTH file). These entries had the same start time but different end time. We located the workstation that was accessing the case at that time. After we rebooted that workstation, the system returned to normal and the rest of the workstations were unfrozen and working again. We concluded that it was not an application problem but a Blaise problem, as the problem surfaced in two other applications that went into production after SLID. The two applications were very different from SLID, i.e. they were single datamodel applications, and they were less complex than SLID.

Also, there was problem with the Blaise callscheduler. It was looping within a certain part of the daybatch. As a result, it ignored most of the untouched cases, and kept delivering previously attempted cases to the interviewers. To solve this problem, we had to explicitly exclude all previously attempted cases from the daybatch for a few days in order to get to those untouched cases. We reported this problem to Westat and Statistics Netherlands. They confirmed our finding, and fixed this problem in the subsequent release of Blaise.

Since the Blaise callscheduler was not delivering cases properly, the interviewers were trying to use the databrowser to get cases, which created another problem. Whenever several interviewers tried to use the databrowser at the same time, everything started to slow down. It took a very long time for the databrowser to scroll up or down, and the interviewers experienced long delays within the application. This did not happen when the databrowser was not used. Since the default databrowser was constantly connected to the database, it created a lot of network traffic. The more the databrowser was used, the worse the problem became. To solve this problem, we had to implement our own databrowser using Maniplus. The Maniplus databrowser program required the interviewers to search on the primary or secondary keys to limit the number of cases to be displayed. It read from the database in shared mode, at the rate of about 2 cases per second. This databrowser might not be fast while retrieving cases, but it did not slow down the system afterward as it was no longer connected to the database. It gave the interviewers an instant view of the database, and also provided other valuable functions for the interviewers. Before, the interviewers had to actually get into a case in order to use the function keys F3, F8, and F11 to view demographic, call summary, and notes, which was quite time consuming. In our browser, we included three control buttons that activate the Maniplus programs for those functions. The interviewers could pick a case from the browser, and they could then decide whether to access it or just look at the information. Not only did we solve the databrowser problem, we even enhanced the application and created a prototype browser for future use.

Conclusions

SLID 2001 was a success despite the problems we encountered. It showed that our new design is feasible and sound. We now need to focus on improving the design or the way that we implement the application, so that we can eliminate the existing problems. The major concern that we have is the delay between CallSchedulerExit and EntryDemRel. We know that the delay is the result of the long initialization of the Maniplus program. So the question is, how can we shorten that time? There are many things that could contribute to the delay, such as the number of datamodels, the size of datamodels, the number of interviewers, traffic on the network, and the way that we open the datamodels. Although this delay does not interrupt an interview, it will eventually affect productivity, so we must shorten the delay to an acceptable level. Basically, there is no delay within each individual datamodel. The only datamodel that has occasional delay is CallSchedulerExit. We know that this delay is probably caused by a constant external read. If we control the external read properly, we will likely eliminate this problem.

There are other improvements which can be made if additional functionality is made available in Blaise. For example, if Blaise can introduce a treatment that returns a case to the daybatch with default priority, we would not have to use our UpdateDayBatch program. The concept of the UpdateDayBatch program works, however we have to schedule it to run regularly. When it runs, it has to access the database and update it, thereby competing with the interviewers, which may affect the performance of the application. Almost every social survey has the same kind of cases that need to be returned to the daybatch. Without the proposed treatment, we will need to prepare an UpdateDayBatch program for each and every survey. This makes it hard to maintain the applications and the standards.

The efficiency of the Blaise databrowser is another thing that requires improvement. Although the databrowser we developed is an ideal replacement for the default one, we still have to develop one for each survey. For some small surveys that do not require the full functionality of our customized browser, it may be easier to use the default one if it does not slow down the system. Nevertheless, in the process of handling the browser problem, we developed our own browser with customized functionality, which clients have found very useful. In fact, it is so useful that we are going to make it a standard feature for all new surveys. Our solution to the problem actually becomes the prototype and the blueprint for the next customized browser.

The experience with SLID 2001 reinforced the potential of the multi-datamodel design, and will enable us to further improve our upcoming surveys. These surveys will also benefit from functions like UpdateDayBatch and the customized browser. SLID 2001 also highlighted some limitations and problems in Blaise and Manipula. The reporting of these problems, as well as their subsequent correction in new releases of Blaise, is of benefit to all Blaise users. With Blaise being constantly enhanced and with our growing expertise with the product, we will continue to see new and improved Blaise CATI application in future. SLID was just the beginning for us.