Using Formal Methods for Teaching Software Engineering: a Tool-Based Approach

P. Ciancarini and C. Mascolo

Dipartimento di Scienze dell'Informazione

Università di Bologna

Mura Anteo Zamboni 7, I-40127 Bologna, Italy

phone:+39 51 354506, fax:+39 51 354510

e-mail: {`ciancarini`, `mascolo`}`@cs.unibo.it`

Abstract

In this paper we describe and review the course plan and syllabus we use in a course on formal methods in software engineering currently included in the degree in Informatics of the Faculty of Sciences at the University of Bologna. The course matches the theory of formal methods with their practice based on actual tool usage. In fact, the course is centered upon a project whose main goal is to let students learn some formal specification techniques all supported by specific tools. The students use well-known notations for both requirements specification and formal design. The formal methods we use are based on the Z notation for requirements specification and on the Larch family of languages for design specifications.

1 INTRODUCTION

Most of our students in the "Corso di Laurea in Informatica" of the Faculty of Sciences of the University of Bologna ask for software engineering courses, because they know that the real world needs software engineers. On the other side, software industries in our area ask us for software engineers able to produce more, cheaper, and better software.

However, developing a software engineering course in a science-oriented Informatics degree is not an easy task. Our students acquire a background biased towards theoretical computer science, namely on algorithm analysis, on programming language semantics, on formal logics, and on program verification, and it is not easy to match this kind of background with a standard software engineering syllabus.

In fact, when we planned the contents of a *first* course in Software Engineering, our initial idea was to use as main references books like [Sommerville 1991] or [Ghezzi *et al.* 1991]. Admittedly, these books are quite complete, however they are oriented towards an engineering background, because they put emphasis on project management, on the software process life-cycle, and on the design phase and related tools.

We felt that their practically oriented approach was not fully appropriate for our students, because it did not exploit their cultural background. Thus, we decided to give a strong formal content to the course, although keeping a strong bias toward the use of tools.

We state the course goal as follows:

> *This course will teach you to understand and evaluate methods for working in teams on developing real distributed software systems, studying and exploiting some paradigms and tools of specification and design in-the-large. The emphasis is on formal models and notations to write and reason with software specifications and architectures, presenting real systems useful to design new systems.*

We decided to follow two guiding principles for structuring the sequence of lectures:

a. the notion of software development process should be presented early in the sequence, to offer a structured conceptual framework suitable to classify the methods and tools presented, and to suggest the students a method to organize their involvement in the project;

b. the notion of formal specification document should be presented introducing several realistic examples, both at the requirements level and at the design level, aiming at showing to the students useful case studies and methods of reasoning on them. In fact, we present to the students several specification and design notations, either formal or informal.

We inspire ourselves to the course plan described in [Garlan *et al.* 1992; Garlan 1992]. What follows is the complete course plan we actually follow, together with the papers used for lectures or suggested to students:

1. Introduction to Software Engineering [Berry 1992; Ghezzi *et al.* 1991]
2. The software production process and software process models [Boehm 1988; Curtis *et al.* 1992];
3. Software engineering environments and tools [Perry and Kaiser 1991; Ambriola *et al.* 1990];
4. Informal and Semi-formal specification [Wing 1988; Wing 1990];
5. Formal specification: theoretical background [Cohen *et al.* 1986];
6. Formal Notations for declarative specification [Kemmerer 1985];
7. The Z language [Bowen 1996] and its tools (LaTeXstyles, Fuzz, ZTC/ZANS, Z/EVES, Z/HOL, PiZa) [Spivey 1992; Jia 1994; Saaltink 1997; Bowen and Gordon 1995; Hewitt *et al.* 1997];
8. Practical examples based on Z [Hinchey and Bowen 1995; Jacky 1997];
9. Software Architectures [Perry and Wolf 1992; Shaw and Garlan 1996];
10. Object oriented design and languages [Booch 1986];
11. Larch, Larch/C++ and tools [Guttag and Horning 1993; Cheon and Leavens 1994].

The students are encouraged to search the Web for other references, in particular we suggest a page on Formal Methods where useful links are provided
(`http://www.comlab.ox.ac.uk/archive/formal-methods.html`).

We also suggest that they browse the USENET newsgroups `comp.specification.z` and `comp.specification.larch` to familiarize themselves with research issues in formal methods.

The course plan is covered in about 70 hours divided in 35 lectures. The course starts in November and ends in May. it is roughly divided in three parts:

1. The software production process (6 lectures);
2. Formal specification methods (12 lectures);
3. Design and formal properties of software architectures and software components (17 lectures).

The final examination is intended to test the level of comprehension of software engineering methods and techniques by a single student, and to discuss a project suggested by the teachers and elaborated by each student team. The project quality is evaluated and its mark is used as basis for individual grading: each student is required to prove her effective involvement in the project. The individual proficiency on theoretical issues establishes the final grade for each student.

2 NOTATIONS AND TOOLS

The project we give to our students aims at teaching the different attitudes which are typical of specifiers and designers, and put them in contrast with the basic attitude that the students hopefully have when they start the course: namely, they are expert programmers. Thus, we have chosen two different notations for the two roles: a very abstract, model oriented, requirements specification notation, namely Z; and a design notation oriented towards abstract data types and objects, namely Larch (more precisely, Larch/C++).

Moreover, we put a lot of emphasis on the use of tools. Since software engineering tools are usually expensive, one of the reasons for choosing Z and Larch is that there are several free tools available for academic use.

We suggest the use of two different notations for requirements and design because they are different phases and a unified specification language could not adequately express all the required features. In short, the specifier has to pay attention to a precise description of requirements; whereas, the designer should worry about the organization of the system implementation to satisfy the requirements. Students use Z for requirements specification because its declarative nature forces them to abandon their "operation" bias. Instead, Larch is adequate for the design phase because it is oriented to abstract data, since it emphasizes structural issues and module interfaces.

2.1 Z

The notation we have chosen for writing requirements documents is Z. It is a formal specification language based on set theory and first-order logic. Basically, a Z specification is composed of a set of constructs, called *schemas*. A schema consists of a declarative part (the *signature*), where typed variables can be declared, and other schemas can be included, and a predicative part consisting of assertions, properties, and invariants which constrain the declared variables.

There are two basic kinds of schemas: those describing abstract states and those representing operations on them, including some *initialization* schemas which are needed to specify the creation of abstract states. The description of an operation schema is based on predicates on state transformation, that is a transition from a pre-state (represented by unprimed variables) to a post-state (represented by primed variables). Any schema representing an operation states that all the invariants of the abstract state involved by the operation still hold in the post-state.

A major feature of Z is the *schema calculus*, that is a way of composing schemas. It uses some special symbols describing how different parts of the specification document can be combined together through logical connectives. For a detailed description of the Z notation the reader should refer to [Spivey 1992].

2.2 Larch

The notation we have chosen to write design specifications is Larch, that is actually a whole family of specification languages [Guttag and Horning 1993; Guttag *et al.* 1985]. A Larch specification is in fact two-tiered, that is, it includes two components written using two different (but integrated) notations:

- the Larch Shared Language (LSL), which allows the designer to define a library of abstract data types [Guttag and Horning 1986];
- a Larch Interface Language (LIL), syntactically close to an implementation language, is used to specify module interfaces [Wing 1987].

While LSL is independent of any programming language, LIL syntax depends on the specific language to be used in the subsequent implementation phase. Basically, LIL modules are used to describe interfaces for operations on the system (using a notation similar to the implementation language to be used). Operations at LIL level follow a pre- and post-condition paradigm and refer to the algebraic operators defined at LSL level, where a well-defined semantics is given for them.

2.3 The tools

Ideally, a number of tools can help a software engineer during the early phases of a development process:

- Pretty printers and text formatters, to improve readability and to provide indexing. For instance, both notations we use (namely Z and Larch) have a LaTeX-based syntax.
- Type checkers, to verify consistency in a specification document containing typed entities [Evans *et al.* 1994; Spivey 1988].

- Theorem provers, to improve confidence in correctness so helping in prove formal properties [Bowen and Gordon 1994; Garland and Guttag 1989].
- Animators, to refine and execute specification documents written in non-executable formal languages. Animators of formal specifications support rapid prototyping, but also help in proving that a specification is really implementable [Sterling *et al.* 1996].
- Test case generators, namely tools able to generate test suites directly from specifications, that can then be used by programmers to test and debug the real implementation [Richardson *et al.* 1992].
- Metricators, to analyze some internal qualities of the design documents.

In our course we solicit students to use most of these tools.

3 THE PROJECT

The lectures on notations and tools are integrated by a project that students, organized in teams of four members each, have to elaborate. A clear software development process is enacted since the first lessons to manage the coordination of teams. Teachers, playing the role of customers, propose an initial document describing a desired product using natural language. Students are given some weeks to study, discuss and clarify the informal requirements with customers and colleagues. This results in a new, final draft of the informal requirements. This is the starting point for the formal requirements specification in Z, to be completed over a period of two months.

The formal document which is produced has to be checked with Z syntax and type checkers `fuzz` or `ZTC`.

All formal documents are cross reviewed and peer evaluated by the teams themselves, using a system similar to that used in scientific conferences. Each team reviews three projects. To avoid "friendly" reviews, each team is compelled to give different marks to each document. Each document is thus independently reviewed by three teams and two teachers. A "best document" is chosen as reference document for the next phase. This document is modified incorporating changes suggested by the reviewers. The next phase is the formal design specification phase. This phase consists in writing another document written in Larch/C, Larch/C++ or Larch/Java.

Students are encouraged to reuse the available LSL library of traits (abstract data types) integrating their own modules. Finally, each team is required to provide at least a partial verification of the final design using the Larch prover to prove properties of the system being designed. Such properties should correspond to properties stated in the Z document. Some teams were also required to provide a validation, using a system to animate the specification document.

In the following subsections we describe one of the projects assigned to students.

3.1 Informal requirements

The informal requirements document is given to students in December. They get three weeks to study the document and "improve" it, namely to clarify its obscurities and ambiguities. Students are usually also given some software that partially implements the required functions, to demonstrate the intended interface and functionalities.

In January we get a number of comments and suggestions, that after a general discussion we use to produce the final draft of the informal requirements.

As an example, what follows is the informal requirements document of "ChessNet", a service to play chess over the Internet:

> Design a service for chess players connected to the Internet.
>
> The service should allow:
>
> - remote connection protected by a password;
> - visualization of the connected users and of their attributes (eg. playing ability);
> - visualization of any match in progress;
> - playing using a graphic interface able to check the legality of the moves and the playing times;
> - sending the match score by e-mail to both players and any observers;
> - broadcast communication to all connected players, or multicast restricted to groups of players in a channel;
> - maintenance of a database of all the played matches;
> - world-wide handling of the rating (ELO) for all the registered users
>
> The informal requirements document describes in detail the different services: we omit this part.

3.2 Formal specification in Z

The final draft containing the informal requirements is given to students at the end of January: they have to produce a formal specification using Z by the end of March. The formal specification in Z has to be written in LaTeX and checked with Fuzz (or Latex2e and ZTC [Jia 1994]). Some teams also use an animator (ZANS [Jia 1994]) to produce a specification prototype.

We now show a sample of part of a specification produced by a team; its purpose is to define the abstract state of the system.

We use capital letters for basic types. The other types are user defined types (their definition is often not shown for simplicity).

The schema *Server* represents the abstract state of a server. Fields *IdUser*, *StateUser*, and *TimeUser* respectively associate an address, a state (i.e *playing*, *not_connected*, *connection_phase*, *not_willing_to_play*), and a connection time to each user. *Pending* associates to pairs of users the pending matches they are playing. The schema *Match* (not shown) records, for every match, an identifier, the names of the two players, the players' moves, the result if the match is finished, and a date. Function *InProgress* ensures that a user is playing "live" at most one match, whereas pending matches are "suspended". Field *Challenges* is a set of *CHALLENGE*: a challenger can challenge at most one user, and all the challengers must be connected. Field *Channels* associates with every channel a set of connected users. A channel is simply a natural number ($Channel == \mathbb{N}$).

Server

$$
\begin{array}{l}
IdUser : USERNAME \rightarrowtail\!\!\!\!\!+ ID \\
StateUser : USERNAME \nrightarrow Status \\
TimeUser : USERNAME \nrightarrow Time \\
Pending : PairPlayers \rightarrowtail\!\!\!\!\!+ Match \\
InProgress : IDMATCH \nrightarrow MatchInProgress \\
Challenges : \mathbb{F}\ Challenge \\
Channels : Channel \rightarrow \mathbb{F}\ USERNAME \\
\hline
\mathrm{dom}\ IdUser = \mathrm{dom}\ StateUser \land \mathrm{dom}\ IdUser = \mathrm{dom}\ TimeUser \\
\forall x : PairPlayers \mid x \in \mathrm{dom}\ Pending \bullet x = Pending(x).Players \\
\forall n, m : IDMATCH \mid n \neq m \land \{n, m\} \subseteq \mathrm{dom}\ InProgress \bullet \\
\quad InProgress(n).Players \cap InProgress(m).Players = \varnothing \land \\
\quad InProgress(n).Players \subseteq \mathrm{dom}\ IdUser \\
\forall x, y : Challenge \mid \{x, y\} \subseteq Challenges \land \\
\quad x.Challenger \neq y.Challenger \land \{x.Challenger, x.Challenged\} \subseteq \mathrm{dom}\ IdUser \\
\forall c : Channel \bullet Channels(c) \subseteq \mathrm{dom}\ IdUser
\end{array}
$$

The following operation specifies the sending of a message from a registered sender to all users connected to one of the channels containing the sender. The conditions are: the user is a server user, she is not in a connection phase, the command she wants to be executed is *Shout* (the function *is* is a global function checking the command variable against a string, in this case *Shout*), and the user is registered. In this case the variable *Receivers*! contains the user connected to the same channels of the user and the variable *Content*! contains the text that has to be sent to them.

$$
\begin{array}{|l}
\hline
\textit{ShoutRegisteredOk} \\
\Xi Server \\
Command? : Content \\
Sender? : ID \\
User : USERNAME \\
Receivers! : \mathbb{F}\, ID; \\
Content! : Content \\
\hline
Sender? \in \operatorname{ran} IdUser \\
User = IdUser^{-1}(Sender?) \\
Connection\_phase \notin StateUser(User) \\
is(Command?, Shout) = TRUE \\
Registered \in StateUser(User) \\
\exists\, chans : \mathbb{F}\, Channel;\ set : \mathbb{F}\, ID \bullet (\forall\, c : Channel \mid \\
\qquad c \in chans \land User \in Channels(c)) \land \\
\qquad\qquad set = \bigcup \{\ c : Channel \mid c \in chans \bullet IdUser(\!|\ Channels(c)\ |\!)\ \} \bullet \\
\qquad\qquad Receivers! = (\mathbf{if}\ set = \varnothing\ \mathbf{then}\ (\operatorname{ran} IdUser)\ \mathbf{else}\ set) \setminus \{\ Sender?\ \} \\
Content! = ResultShout(User, ExtractText(Command?)) \\
\hline
\end{array}
$$

The schema *TellSomebody* specifies the sending of a message from one registered sender to another (specified) connected user: we distinguish the cases in which the sender is playing or not, in the former case the receiver of the "tell" message is the opponent; in the latter the receiver should be indicated. *TellSomebody* specifies the case in which the user is not playing.

$$
\begin{array}{|l}
\hline
\textit{TellSomebody} \\
\Xi Server \\
Command? : Content \\
Sender? : ID \\
User : USERNAME \\
Receiver! : ID; \\
Content! : Content \\
\hline
Sender? \in \operatorname{ran} IdUser \\
User = IdUser^{-1}(Sender?) \\
Connection\_phase \notin StateUser(User) \\
is(Command?, Tell) = TRUE \\
Registered \in StateUser(User) \\
\forall\, p : MatchInProgress \mid p \in \operatorname{ran} InProgress \bullet User \notin p.Players \\
ExtractUsername(Command?) \in \operatorname{dom} IdUsers \\
Receiver! = IdUser(ExtractUsername(Command?)) \\
Content! = ResultTell(User, ExtractText(Command?)) \\
\hline
\end{array}
$$

The state of the Chessboard on the client display is defined in the following schema: *ToMove* indicates the next color to move.

$$
\begin{array}{|l}
\hline
\textit{StateChessBoard} \\
ToMove : Color \\
MatchState : StateBoard \\
\hline
\end{array}
$$

StateEditor indicates the state of the client display: the current configuration of the chess-board, the color, the games currently observed, the messages received from the selected channels. *Moves* is a sequence of moves. The present configuration must be one of the possible configuration (boolean function *CheckMoves* checks against illegal moves).

$$
\begin{array}{|l}
\hline
\textit{StateEditor} \\
\textit{PresentPos} : \textit{StateChessBoard} \\
\textit{MyColor} : \textit{Color} \\
\textit{ObservedMatches} : \textit{IDMATCH} \pfun \textit{StateChessBoard} \\
\textit{BoxMsg} : \operatorname{seq} \textit{STRING} \\
\textit{Video} : \textit{VIDEO} \\
\hline
\exists\, s : \textit{Moves} \bullet \textit{CheckMoves}(\textit{PosInitial}, s) = \textit{TRUE} \land \\
\qquad \textit{PresentPos} = \textit{FindPos}(\textit{PosInitial}, s) \\
\textit{Video} = \textit{visual}(\textit{PresentPos}, \textit{MyColor}, \textit{BoxMsg}) \\
\hline
\end{array}
$$

The following schema defines the state of a client: it is defined by an interface (*StateEditor*) and the address of the chess-server to which it is connected.

$$
\begin{array}{|l}
\hline
\textit{Client} \\
\textit{server} : \textit{ID} \\
\textit{StateEditor} \\
\hline
\end{array}
$$

The following schema defines the operation of performing a move: every action performed by the client is recorded and sent to the server to be transmitted to the opponent.

$$
\begin{array}{|l}
\hline
\textit{MyMoveOk} \\
\Delta \textit{Client} \\
\textit{Move}? : \textit{MOVE} \\
\textit{Receiver}! : \textit{ID} \\
\textit{Content}! : \textit{Content} \\
\hline
\textit{MyColor} = \textit{PresentPos}.\textit{ToMove} \\
\textit{CheckMove}(\textit{PresentPos}, \textit{Move}?) = \textit{TRUE} \\
\textit{PresentPos}' = \textit{UpdatePos}(\textit{PresentPos}, \textit{Move}?) \\
\textit{Receiver}! = \textit{server} \\
\textit{Content}! = \textit{MoveInMessage}(\textit{Move}?) \\
\hline
\end{array}
$$

The actual delivery is not shown here: *MyMoveOk* only encodes the move in message format.

At the end of March starts a phase (one week) in which each team reviews the specifications of three other teams.

We find this sort of work helpful in finding specification errors, for example, in the case of "Chessnet", one team, not exploiting the properties of the sets, had written the predicate:

$\forall\, x, y : \textit{Challenge} \mid \{x, y\} \subseteq \textit{Challenges} \land x.\textit{Challenger} \neq y.\textit{Challenger}$

$\land \{x.\textit{Challenger}, x.\textit{Challenged}\} \subseteq \operatorname{dom} \textit{IdUser}$

of the schema *Server* as:

$\forall\, x, y : \textit{Challenge} \mid x \neq y \land \{x, y\} \subseteq \textit{Challenges} \land x.\textit{Challenger} \neq y.\textit{Challenger} \land$

$\{x.Challenger, x.Challenged\} \subseteq \mathrm{dom}\ IdUser$

Students gradually learn to exploit several features characteristics of the set-theory and the first-order logic that Z offers. Often type-checking is sufficient to find most errors. In some cases like this one we found helpful the cross revision phase, in which students compares their schemas to the other teams' specifications schemas.

At the end of the cross revision we have a "winner" document, that, opportunely revised, is given as input document for the next phase, formal design.

This is a crucial point in our project: we make explicit to our students that now they have to change their project role and attitude: they are no more specifiers, they are now designers. Thus, the main question for them to answer is: how can the requirement specification document be transformed in a design specification document?

Before starting the design phase the specification document is analyzed using a graphical notation to show the main relationships among the schemas it contains. In Fig. 1 it is shown such a graphic notation. The picture shows how the operation schemas are encapsulated using schema inclusion and schema calculus; it also shows the links between operation schemas and state schemas (the arrows).

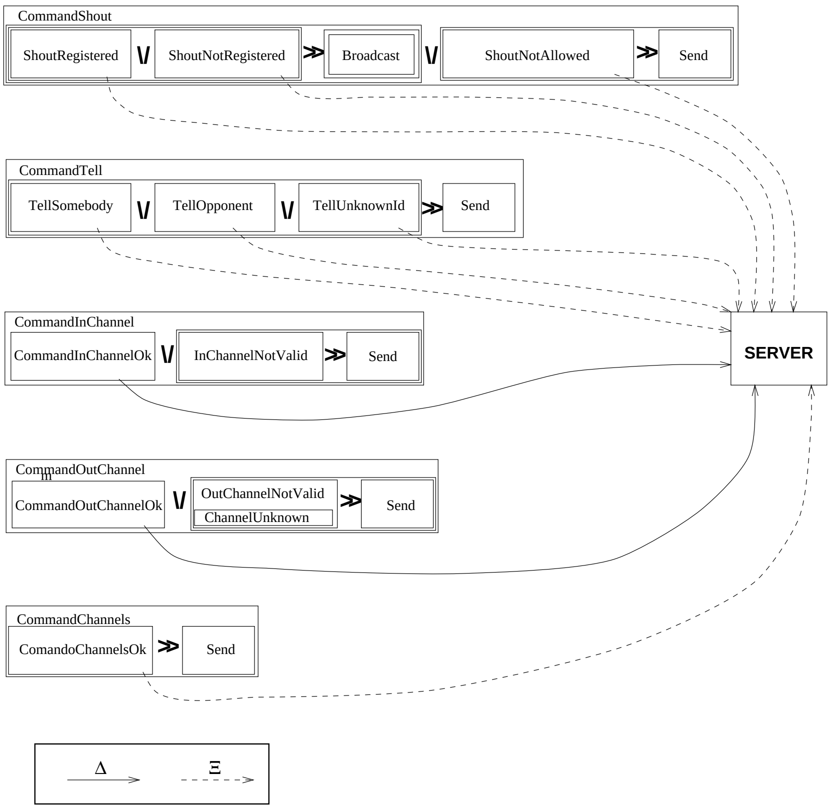

Figure 1: A graphical view of the schema calculus representation

For instance, the *Tell* command is composed of an "or" of three schemas defining the different *tell* operations (in case the user is playing or not, and in case the receiver id is unknown) piped to a *Send* schema specifying how a message is delivered to a receiver. The dashed arrows (Ξ) indicate that all the operation schemas use the *Server* variables but they do not modify them.

We noticed that students are happy to exploit the Z "schema calculus" but they often make mistakes using pipes ($\gg$). They have problems in understanding that a pipe ($\gg$) matches the output variables of a schema (eg., *receiver*! of *TellSombody* schema) with input variables (e.g. *receiver*? of the schema *Send*) only if they have the same name. The schema *Send* is not shown but its interaction with the other schemas is described in Fig. 1. The problem is that the type checker does not warn for this kind of errors.

Students find helpful pictures like Fig. 1 in checking dependencies of schemas. This representation is the actual starting point for the design phase. Interestingly, different teams build different representations for the same Z document, because usually they use the graphics to convey some initial ideas on the software architecture they intend to use.

3.3 Formal design: abstract data types in Larch Shared Language

In the design phase our goal is to show students which is the use of the formal requirements document in the next step, namely architectural design. Teams are required to write a formal design document in Larch. The emphasis is simultaneously on the overall structure of the *software architecture* to be designed [Shaw and Garlan 1996] and on *reusing* traits included in a library. They also are required to use the Larch Prover to interactively verify at least one relevant property of the system being designed. Figure 2 shows a design document corresponding to the Z specification given in Fig. 1.

The picture represents the algebra of the system: it provides a global view of the inclusion and assumes relations that form the ChessNet algebra. The square boxes show the library traits while the oval boxes show the ChessNet traits. The boldface oval boxes are the complex traits graphically redefined in different pictures.

We include a LSL *trait* (the basic unit of specification in Larch) where operators and their properties are introduced. It corresponds to the *Channels* function in the Z schema *Server* Such a trait includes some previously defined traits. The `introduces` section describes some functions: RemoveUser defines the removal of a user from a channel, while Partners returns, given a user and a channel, the set of users that are into the same channel. The `implies` section contains theorems that will be proved (using the Larch Prover). The `converts` clause states the fact that the functions Partners, RemoveUser and emptychannels are completely defined by the assertional part of the trait.

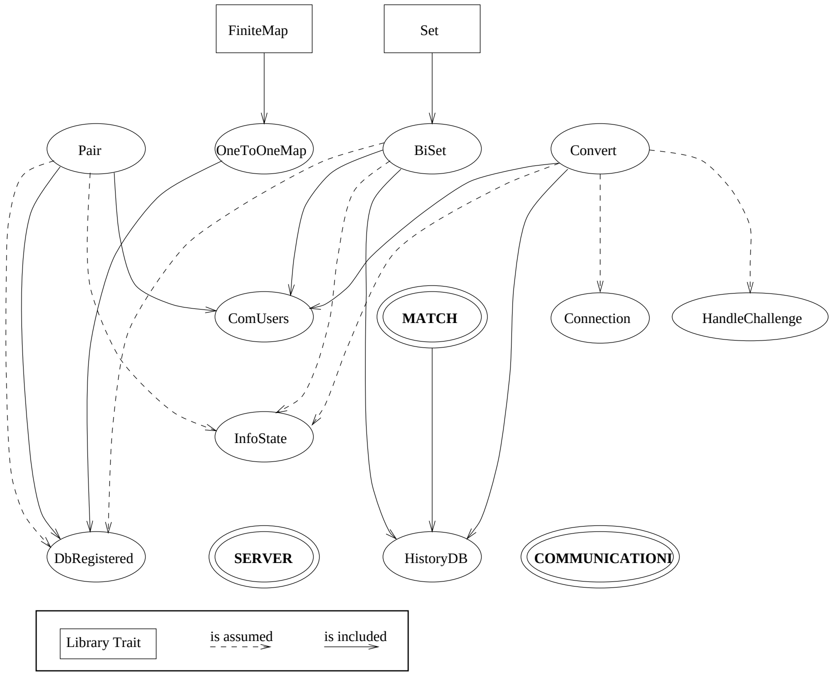

Figure 2: A Structure of LSL Traits used in the project

Channels: **trait**

includes

BiSet(USERNAME,USet),

FiniteMap(CHANNELS,CHANNEL,USet)

introduces

Partners: USERNAME, CHANNELS $\rightarrow$ USet

RemoveUser: USERNAME, CHANNELS $\rightarrow$ CHANNELS

emptychannels: $\rightarrow$ CHANNELS

asserts

$\forall$ u: USERNAME, chann: CHANNELS, c: CHANNEL, uset: USet

Partners(u,{}) = {};

Partners(u,update(chann,c,uset)) ==

if u $\in$ uset then (uset - {u}) $\cup$ Partners(u,chann)

else Partners(u,chann);

RemoveUser(u,{}) = {};

RemoveUser(u,update(chann,c,uset)) == update(RemoveUser(u,chann),c,uset-{u});

apply(emptychannels,c) = {};

implies

$\forall$ u: USERNAME, chann: CHANNELS

Partners(u,RemoveUser(u,chann)) = {};

converts Partners, RemoveUser, emptychannels

Using this trait we show an example of proof which students built using the Larch Prover (LP). The use of LP has revealed to be quite difficult and time-consuming for students. They have to help the prover to build the proof suggesting the way in which it has to be carried on (i.e. by induction, by case, by implication, and so on).

The following script is the result of a proof by induction on the structure of the function Partners: the set of user in communication with the user u is empty if u has been removed from each channel.

```
    set name Channels

declare sort USERNAME, CHANNEL, CHANNELS, USet
declare variable u: USERNAME, c,c': CHANNEL, chann,chann': CHANNELS, uset: USet

declare operators
  nomap: → CHANNELS
  update: CHANNELS, CHANNEL, USet → CHANNELS
  apply: CHANNELS, CHANNEL → USet
  defined: CHANNELS, CHANNEL → Bool
  Partners: USERNAME, CHANNELS → USet
  RemoveUser: USERNAME, CHANNELS → CHANNELS
  ..

assert CHANNELS generated by nomap, update
assert CHANNELS partitioned by apply, defined
assert
  apply(update(chann, c', uset), c) == if(c = c', uset, apply(chann, c))
  not(defined(nomap, c))
  defined(update(chann, c', uset), c) == c = c' | defined(chann, c)
  Partners(u,nomap) = empty
  Partners(u,update(chann,c,uset)) ==
              if(u ∈ uset, (uset \ singleton(u)) ∪ Partners(u,chann),
                      Partners(u,chann))
  RemoveUser(u,nomap) = nomap
  RemoveUser(u,update(chann,c,uset)) ==
              update(RemoveUser(u,chann),c,uset \ singleton(u))
  ..

prove Partners(u,RemoveUser(u,chann)) = empty by induction on chann
  <> 2 subgoals for proof by induction on 'chann'
    [] basis subgoal
    [] induction subgoal
  [] conjecture
```

qed

A team noticed that the complexity of a proof often depends on the kinds of suggestions given: here are two scripts of the same proof. The former one progresses by implication, instead the latter one goes on by induction and then by case.

The proof shows that if a server and a client are not connected, then opening and closing a connection between them does not alter the set of connected users.

For clarity we first show the LSL trait and then the proof:

```
Connections: trait

  Connection tuple of
    server: ID,
    client: ID

  includes
    Set(Connection,Connections,NoConnection for {})

  introduces
    Connected:ID,ID,Connections → Bool
    OpenConnection:ID,ID,Connections → Connections
    CloseConnection:ID,ID,Connections → Connections

  asserts
    ∀ u1, u2, i1, i2: ID, Coll: Connections
    Connected(u1,u2,Coll) == [u1,u2] ∈ Coll ∨ [u2,u1] ∈ Coll;
    OpenConnection(u1,u2,Coll)== if Connected(u1,u2,Coll) then Coll
      else insert([u1,u2],Coll);
    CloseConnection(u1,u2,NoConnection)==NoConnection;
    CloseConnection(u1,u2,insert([i1,i2],Coll))== if ((u1=i1 ∧ u2=i2) ∨
      (u1=i2 ∧ u2=i1))
      then Coll
      else insert([i1,i2],CloseConnection(u1,u2,Coll));

implies
    ∀ u1, u2: ID, Coll: Connections

    Connected(u1,u2,OpenConnection(u1,u2,Coll));
    ~Connected(u1,u2,Coll) ⇒
      CloseConnection(u1,u2,OpenConnection(u1,u2,Coll))=Coll
```

And these are the two proofs
by implication:

```
prove ~Connected(u1,u2,Coll) ⇒
    CloseConnection(u1,u2,OpenConnection(u1,u2,Coll))=Coll by ⇒
        <> ⇒ subgoal
        [] ⇒ subgoal
    [] conjecture
```

and by induction and by case:

```
prove ~Connected(u1,u2,Coll) ⇒
    CloseConnection(u1,u2,OpenConnection(u1,u2,Coll))=Coll
    by induction on Coll using Set
    <> basis subgoal
    [] basis subgoal
    <> basis subgoal
    resume by case ~([u1, u2] = c4 ∨ [u2, u1] = c4)
        <> case ~([u1c, u2c] = c4c ∨ [u2c, u1c] = c4c)
        [] case ~([u1c, u2c] = c4c ∨ [u2c, u1c] = c4c)
        <> case ~(~([u1c, u2c] = c4c ∨ [u2c, u1c] = c4c))
        [] case ~(~([u1c, u2c] = c4c ∨ [u2c, u1c] = c4c))
    [] basis subgoal
    <> induction subgoal
    resume by case ~([u1, u2] ∈ Collc1∨ [u1, u2] ∈ Collc∨ [u2, u1] ∈ Collc1
        ∨ [u2, u1] ∈ Collc)
        <> case
        [] case
        <> case
        [] case
    [] induction subgoal
[] conjecture
```

3.4 Using a LIL

An important feature of the Larch method is the availability of a family of Larch Interface Languages, namely special notations by which the designer can write modules which refer the traits but also are oriented to a specific programming (implementation) language. The main interface languages currently available are LClint, for C, and LCC, for C++ [Cheon and Leavens 1994]. Students use LILs to define the dynamics of the design: the procedures that will be implemented on the static structure of the system defined in LSL. What follows is an example of an LCC module:

```
imports Server;
imports Channels;

class ServerChess: public Server,private Channels {
    uses ServerChess;

invariant ∀ c:CHANNEL (apply(self.channels,c) ⊆ dom(self.users));


public:
    ServerChess(){ // constructor
        modifies self;
        ensures self′=InitServerChess;
    }
    ~ServerChess() { // destructor
        modifies self;
        ensures trashed(self);
    }
    Message& Shout(Message& const m) const {
        requires Is(Msgtext(m),Shout) ∧
            Msgsender(m) ∈ Identifiers(ran(self.users)) ∧ CONNECTION ∉ apply(self.users,
            Username(self.users,Msgsender(m))).state;
        ensures ∃ u:USERNAME ( u=Username(self.users,Msgsender(m)) ∧
            (if (Partners(self.channels,u) ≠ {})
            then result=Set_AllMsg(Msgreceiver(m),
                    Partners(self.channels,u),Extractext(Msgtext(m)))
            else (if (REGIS ∈ (apply(self.users,u)).state)
                then result=Set_AllMsg(Msgreceiver(m),delete(Msgsender(m),
                        Identifiers(ran(self.users))),Extractext(Msgtext(m)))
                else result=Set_AllMsg(Msgreceiver(m),Msgsender(m),CreateContent(CommandNotAllowed)))));
    }
    Message& Tell(Message& const m) const {
        requires Is(Msgtext(m),Tell) ∧
            Msgsender(m) ∈ Identifiers(ran(self.users)) ∧
            CONNECTION ∉ apply(self.users,
            Username(self.users,Msgsender(m))).state;
        ensures ∃ u:USERNAME (u=Username(self.users,Msgsender(m))∧
            (if (PLAY ∈ apply((self.users).state,u))
            then result = Set_AllMsg(Msgreceiver(m),
```

```
            Opponent(Msgsender(m)),Extractext(Msgtext(m)))
        else (if (ExtractUsername(Msgtext(m)) ∈ dom(self.users))
            then result=Set_AllMsg(Msgreceiver(m),
                apply(self.users,ExtractUsername(Msgtext(m))).id,Extractext(Msgtext(m)))
            else result=Set_AllMsg(Msgreceiver(m),Msgsender(m),CreateContent(UserNotConnected)))));
    }
    ....
};
```

Such a module refers a number of LSL traits (clause `uses`). The “imports” indicates the imported lcc classes while clause “uses”, the LSL traits the class refers to.

The operations *Shout* and *Tell* are specified using pre and post conditions (requires and ensures). The predicates expressed there refers to functions and variables of the LSL traits imported. The invariants in the class are properties preserved by all the operations. Students usually encode Z state schema predicative parts using these invariants. The invariant property of this class ensures that every user in a channel is connected to the Chess-Server. The Z operations are encoded as lcc functions while Z state schema are LSL traits.

4 DISCUSSION

We have given such a course for five years and are re-evaluating this approach. Moreover, we have developed a complete description and analysis of the method we have chosen in an other paper [Ciaccia *et al.* 1997].

We believe that the main advantage of our approach consists of proving our students that specifying, designing, and programming are very different activities which need different skills and adequate tools. Formal methods are very successful in shoving the difference between specifying and programming. We believe we have been less successful in teaching design issues. In fact, Z and Larch were invented both as specification languages. We are now gradually putting more emphasis on object oriented analysis and design patterns.

Although the emphasis is on formal methods, our students use a lot of tools. We feel that this use of tools cannot be over-emphasized in a software engineering course. The students are requested to enact the process using personal workstations and tools like LaTeX, xfig, Fuzz and ZTC (for checking Z specifications), the LSL checker (for checking Larch traits), the LCC checker (for checking Larch/C++ modules), and the Larch Prover (to check properties of abstract data types). We believe that this approach was very successful,

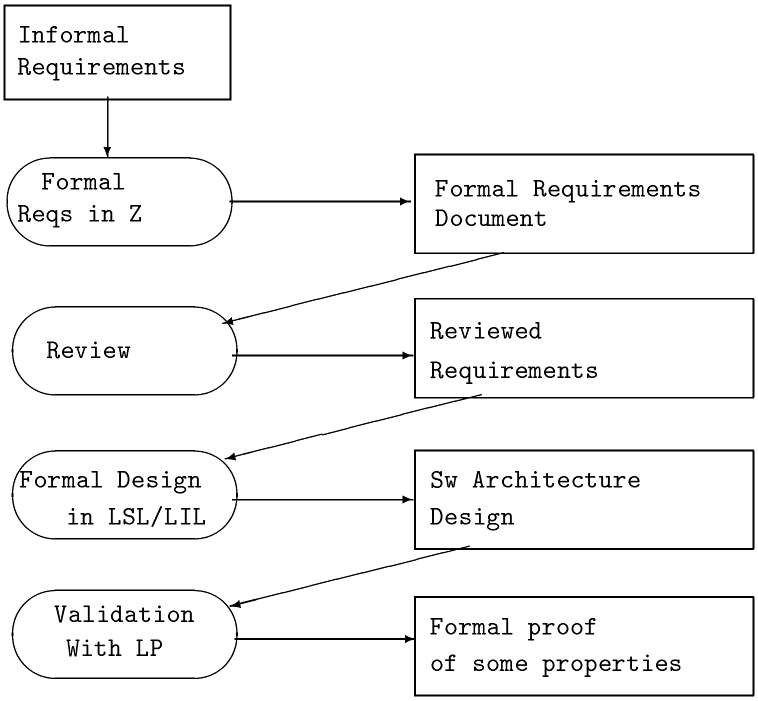

Figure 3: The development process used by our students

because the students could appreciate how expensive can become a structured software development process, and how useful some tools are.

Probably the most important aspect of our approach was the methodology we suggested, based on a form of contractual development process using strongly formalized notations. The overall process is depicted in Fig.3. Each rounded box is a process phase; each box is a document used as input for lower phases.

The best formal requirements document was used as input for the design phase, where they had to use an algebraic language to provide the design document, paying attention to any occasion of reuse of existing modules. Such a document had to be at least partially verified, proving with the help of a theorem prover some relevant properties of the system being designed.

We believe that the choice of the project assignment is critical: it has to be neither too small nor too complex. Every year a different project is given to students: an auction system or a stock-exchange system are examples of projects that students had to develop.

In any case, it is especially important the very first presentation of informal requirements. For instance, in the case we discussed in this paper, students initially discussed a lot of chess-related questions, then they realized that they were required to develop a system whose main components are an editor and a database, and their final documents were fairly centered upon these features, forgetting about chess.

We strongly argue for using two different formalisms for requirements and architecture specification (even if, admittedly, Larch is not very good as an architectural design language). Students understood quite well the differences in the goals of the different phases and resulting documents.

Most teams tried to follow the format we suggested for specification in Z: top-down development, first specifying data, then states, and finally operations. We were quite satisfied with the level of proficiency in Z demonstrated by the students; they skillfully used advanced features of the language, and also the experience with the type checker Fuzz was positive. All the teams reported that Fuzz was a good help for checking the schemata. A team developed also an original graphical documentation to help examiners in understanding the specification; see fig. 1. However, most teams concentrated much more on algorithmic features than on declarative model-oriented presentation. The schema calculus was scarcely used; comments were almost absent. Almost nobody tried spontaneously to prove properties of the abstract system: we had to force them.

The Larch-based formal design phase is more difficult to assess. Basically, the Larch method requires to write algebraic theories giving a semantics to the basic operators (this is done using the Larch Shared Language). These operators need not to be implemented, but can be used in the definition of module interfaces (written in a Larch Interface Language specific to the implementation language), where the functions that the module makes available to its client are introduced. This so-called *two-tiered* style of specification of modules encourages modularity and separation of concerns, but it challenges specifiers as to how and where split the tiers. The teams were instinctively led to put in the LSL tier most of the required functionalities of the modules. This resulted in scarcely reusable design specifications. Sometimes, also error-handling considerations were present.

We observed that all teams tried to reuse, and possibly extend, some of the traits in the LSL library. According to the Larch style, the design mainly proceeded bottom-up, with traits and operators somehow resembling, respectively, types and operations specified in Z. Finally, the formal proof of properties with the Larch Prover was the most difficult and time-consuming task to perform, however the students appreciated its importance and cost.

Probably the most important shortcoming we found in Larch is the lack of support for architectural specifications.

The approach we have chosen generates some important questions, to which we cannot yet provide definitive answers.

- *Do student learn better design practices if they start from formal requirements rather than informal ones?* Although we observed that in most final documents the separation of concerns between the two phases of requirements and design specification was not so clear-cut (e.g., design considerations were often found in Z documents) the bottom-up design activity was guided by the need of providing adequate support to the high-level functionalities expressed in Z. In some sense, this simplified the activity of designing the overall architecture of the system. It seems that the choice of an appropriate abstraction level is a crucial point for formal requirements to be effectively advantageous.

- *What is the relationship between a model-based (i.e., Z) and an algebraic (i.e., Larch) style of specification?* Translating from one language to the other is not trivial. The main difficulty, from the student point of view, appeared to be moving from one language where the concept of *system state* is a basic one, to another where it is completely meaningless.

  A formal link is given by verification: in fact, it is possible to specify and prove properties in Z, that then can to be expressed in Larch and finally proved by means of the Larch Prover. However, this was only done with properties expressed in LSL, because the nature of the available tools. We conjecture that the glue between the two styles could be represented by the interface specification, where the state concept is peculiar to the specific implementation language, and algebraic-defined terms are first introduced.

Acknowledgments. We would like to thank P. Ciaccia and W. Penzo for discussion and cooperation. We also have to thank our students who were patient enough to attend this course, and provided us with interesting project solutions. We also are very grateful to the people who built and made available the tools we used in this course. We thank the anonymous reviewers for their comments.

This paper has been partially supported by the Italian Ministry of University and Scientific and Technological Research (MURST).

REFERENCES

Ambriola, V., L. Bendix, and P. Ciancarini (1990), "The Evolution of Configuration Management and Version Control," *IEE Software Engineering Journal 5*, 6, 303–310.

Berry, D. (1992), "Academic Legitimacy of the Software Engineering Discipline," Technical Report CMU-SEI-92-34, Software Engineering Institute, Carnegie Mellon Univ.

Boehm, B. (1988), "A spiral model of software development and enhancement," *IEEE Computer 21*, 5, 61–72.

Booch, G. (1986), "Object-Oriented Development," *IEEE Transactions on Software Engineering 12*, 2, 211–220.

Bowen, J. (1996), *Formal Specification and Documentation using Z: A Case Study Approach*, International Thomson Computer Press.

Bowen, J. and M. Gordon (1994), "Z and HOL," In *Proc. 8th Z Users Workshop (ZUM)*, J. Bowen and J. Hall, Eds., Workshops in Computing, Springer-Verlag, Berlin, Cambridge, UK, pp. 141–167.

Bowen, J. and M. Gordon (1995), "A Shallow Embedding of Z in HOL," *Information and Software Technology 37*, 5-6, 269–276.

Cheon, Y. and G. T. Leavens (1994), "A Quick Overview of Larch/C++," *Journal of Object-Oriented Programming 7*, 6, 39–49.

Ciaccia, P., P. Ciancarini, and W. Penzo (1997), "Formal Requirements and Design Specifications: The Clepsydra Methodology," *Int. Journal on Software Engineering and Knowledge Engineering 7*, 1, 1–42.

Cohen, B., W. Harwood, and M. Jackson (1986), *The Specification of Complex Systems*, Addison-Wesley.

Curtis, B., M. Kellner, and J. Over (1992), "Process Modeling," *Communications of the ACM 35*, 9, 75–90.

Evans, D., J. Guttag, J. Horning, and Y. Tan (1994), "LCLint: a Tool for Using Specifications to Check Code," In *Proc. 2nd ACM SIGSOFT Symp. on Foundations of Software Engineering*, D. Wile, Ed., volume 19:5 of *ACM SIGSOFT Software Engineering Notes*, New Orleans, USA, pp. 87–96.

Garlan, D. (1992), "Formal Methods for Software Engineers: Tradeoff in Curriculum Design," In *Software Engineering Education, Proc. SEI Conference*, C. Sledge, Ed., volume 640 of *Lecture Notes in Computer Science*, Springer-Verlag, Berlin, SanDiego, CA, pp. 131–142.

Garlan, D., M. Shaw, C. Okasaki, C. Scott, and R. Swonger (1992), "Experience with a Course on Architectures for Software Systems," In *Proc. Conf. on Software Engineering Education*, volume 640 of *Lecture Notes in Computer Science*, Springer-Verlag, Berlin, pp. 23–43.

Garland, S. and J. Guttag (1989), "An overview of LP, the Larch Prover," In *Proc. 3rd Int. Conf. on Rewriting Techniques and Applications*, volume 355 of *Lecture Notes in Computer Science*, Springer-Verlag, Berlin, pp. 137–151.

Ghezzi, C., M. Jazayeri, and D. Mandrioli (1991), *Fundamentals of Software Engineering*, Prentice-Hall.

Guttag, J. and J. Horning (1986), "A Larch Shared Language Handbook," *Science of Computer Programming 6*, 135–156.

Guttag, J. and J. Horning (1993), *Larch: Languages and Tools for Formal Specification*, Springer-Verlag, Berlin.

Guttag, J., J. Horning, and J. Wing (1985), "The Larch Family of Specification Languages," *IEEE Software 2*, 5, 24–36.

Hewitt, M., C. O'Halloran, and C. Sennett (1997), "Experiences with PiZA, an Animator for Z," In *Proc. 10th Int. Conf. on the Z Formal Method (ZUM)*, J. Bowen, M. Hinchey, and D. Till, Eds., volume 1212 of *Lecture Notes in Computer Science*, Springer-Verlag, Berlin, Reading, UK, pp. 37–51.

Hinchey, M. G. and J. P. Bowen, Eds. (1995), *Applications of Formal Methods*, Prentice-Hall.

Jacky, J. (1997), *The Way of Z: Practical Programming with Formal Methods*, Cambridge University Press, UK.

Jia, X. (1994), *ZTC: A Type Checker for Z – User's Guide*, Institute for Software Engineering, Department of Computer Science and Information Systems, DePaul University, Chicago, IL 60604, USA.

Kemmerer, R. (1985), "Testing Formal Specifications to Detect Design Errors," *IEEE Transactions on Software Engineering 11*, 1, 32–43.

Perry, D. and G. Kaiser (1991), "Models of Software Development Environments," *IEEE Transactions on Software*

Engineering 17, 3, 283–295.

Perry, D. and A. Wolf (1992), "Foundations for the Study of Software Architecture," *ACM SIGSOFT Software Engineering Notes 17*, 4, 40–52.

Richardson, D., S. Aha, and T. O'Malley (1992), "Specification-based Test Oracles for Reactive Systems," In *Proc. 14th IEEE Int. Conf. on Software Engineering*, Melbourne, Australia, pp. 105–118.

Saaltink, M. (1997), "The Z/EVES system," In *Proc. 10th Int. Conf. on the Z Formal Method (ZUM)*, J. Bowen, M. Hinchey, and D. Till, Eds., volume 1212 of *Lecture Notes in Computer Science*, Springer-Verlag, Berlin, Reading, UK, pp. 72–88.

Shaw, M. and D. Garlan (1996), *Software Architecture. Perspectives on an Emerging Discipline*, Prentice-Hall.

Sommerville, I. (1991), *Software Engineering*, Fourth Edition, Addison-Wesley.

Spivey, J. (1988), *The fUZZ Manual*.

Spivey, J. (1992), *The Z Notation. A Reference Manual*, Second Edition, Prentice-Hall.

Sterling, L., P. Ciancarini, and T. Turnidge (1996), "On the Animation of Not Executable Specifications by Prolog," *Int. Journal on Software Engineering and Knowledge Engineering 6*, 1, 63–88.

Wing, J. (1987), "Writing Larch Interface Language Specifications," *ACM Transactions on Programming Languages and Systems 9*, 1, 1–24.

Wing, J. (1988), "A Study of 12 Specifications of the Library Problem," *IEEE Software 5*, 4, 66–76.

Wing, J. (1990), "A Specifier's Introduction to Formal Methods," *IEEE Computer 23*, 9, 8–24.