GENESYS: An Integrated Environment for Developing Systemic Functional Grammars

KUMANO, Tadashi* TOKUNAGA, Takenobu
INUI, Kentaro TANAKA, Hozumi

Department of Computer Science
Tokyo Institute of Technology

Abstract

In order to develop rich language resources systematically and efficiently, we need not only well-founded linguistic theories but also software tools that facilitate writing and examining them. This paper reports on the GENESYS system, which provides an integrated environment for developing systemic functional grammars (SFGs). GENESYS has a special editor for writing SFGs. Further, the user can examine grammars by running the GENESYS' surface generator. The information of the intermediate states of the generation process can be monitored through the graphical user interface.

1 Introduction

In the field of natural language generation, several experimental systems have been developed. Most of them, however, can generate only restricted variations of language expressions since their language resources are insufficient. Among those systems, the Penman system [9] is an exception, which has a farely large English grammar named Nigel [8]. The current version of Nigel, however, seems too complicated and difficult to maintain without help by computers. In any case, rich language resources are indispensable for practical applications of natural language generation.

In order to develop such resources systematically and efficiently, we need not only well-founded linguistic theories but also software tools that facilitate writing and examining them. For a linguistic foundation, we rely on the framework of systemic functional linguistics [5]. For software tools, we developed a system that provides an integrated environment for developing systemic functional grammars (SFGs). The system is called GENESYS. This paper presents an overview of GENESYS.

Figure 1 illustrates an overview of the task of developing a lexico-grammar on GENESYS. First the user writes a grammar on the editor of GENESYS. Then the user examines it through actual generation process. GENESYS provides an environment in which the user can monitor generation processes. The information of the intermediate states of those processes is presented through the graphical user interface (GUI). Such information is often useful for the user to detect problems (bugs) in the current grammar. If any problems are detected, the user modifies the grammar and then examine it again.

In the following, section 2 gives a brief overview of SFG and a computational frame-

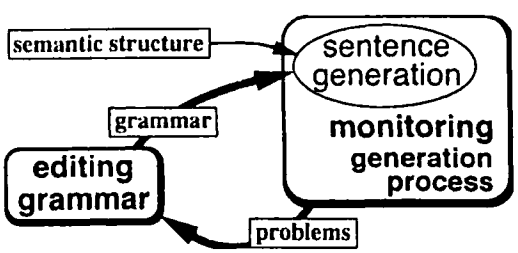

Figure 1: The flow of the grammar development

* Tokunaga Lab. Dept. of Conputer Science
Tokyo Institute of Technology
2-12-1 Ookayama Meguro Tokyo 152 JAPAN
E-mail: kumano@cs.titech.ac.jp

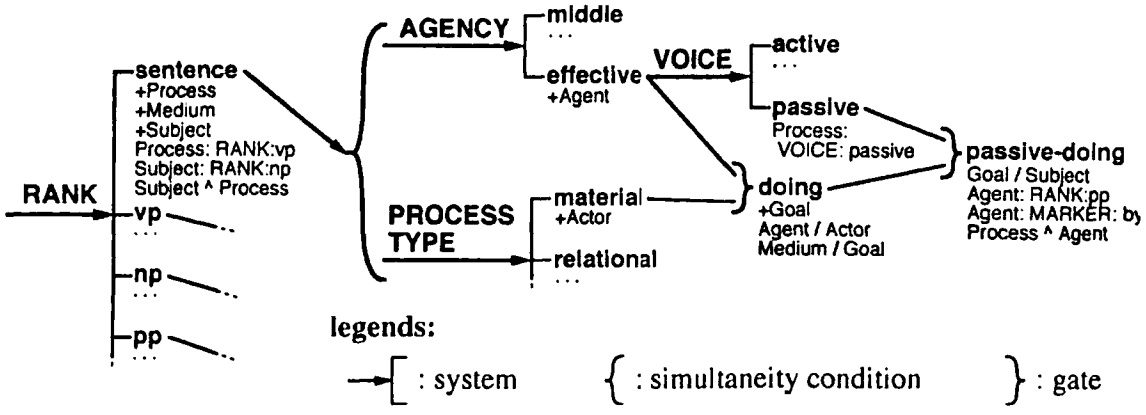

Figure 2: A system network

work for implementing SFGs. Section 3 explains about each component of GENESYS. Section 4 shows a worked example, and section 5 summarizes the paper.

2 Systemic Functional Grammar and Functional Unification Grammar

2.1 Systemic functional grammar

A SFG is a description of the lexicogrammatical stratum in the systemic functional theory. It declaratively describes the correspondences between potential meanings and their language expressions. The potential meanings are typologically organized according to a typology of a set of grammatical features. In this typology, each feature is related with some other features in terms of the subsumption and co-occurrence relations. These relations are usually represented by a system network.

Figure 2 shows a simple example of a system network. A system network consists of several kinds of basic elements: systems, simultaneity conditions, gates and realization statements.

A system represents a dimension of the space of potential meanings, such as AGENCY and PROCESS TYPE, which has a set of alternative grammatical features as its values. A simultaneity condition denotes more than one systems, i.e. dimensions, are orthogonal. A horizontal link denotes a subsumption relation between two grammatical features; for example, passive is subsumed by effective and doing is subsumed by material. Note that doing is subsumed by both effective and material. This is represented by a gate[1].

Each grammatical feature is associated with a set of realization statements. A realization statement is a syntactic (structural) constraint upon the corresponding language expression. For example, +Agent means that there exists a constituent functioning as the agent of the process. Agent/Actor, which is referred to by the term *conflation* in SFG, means that the Agent and Actor are realized as a single constituent. A statement such as Subject: RANK:np, which is referred to by *preselection*, denotes a constraint on a co-occurrence relation between a feature of the current constituent and that of its daughter. Subject: RANK:np, for example, means that the feature np must be chosen in the RANK system for the constituent Subject if the current constituent has the feature sentence. Finally, Subject ^ Process denotes a constraint on ordering constituents, meaning that Process immediately follows Subject.

[1] Originally, a system network has two types of gates: *and* gates and *or* gates. The current version of GENESYS, however, allows only *and* gates because the information represented by an *or* gate can also be represented by introducing another simultaneity condition, and we believe that the latter would be even better from the theoretical point of view.

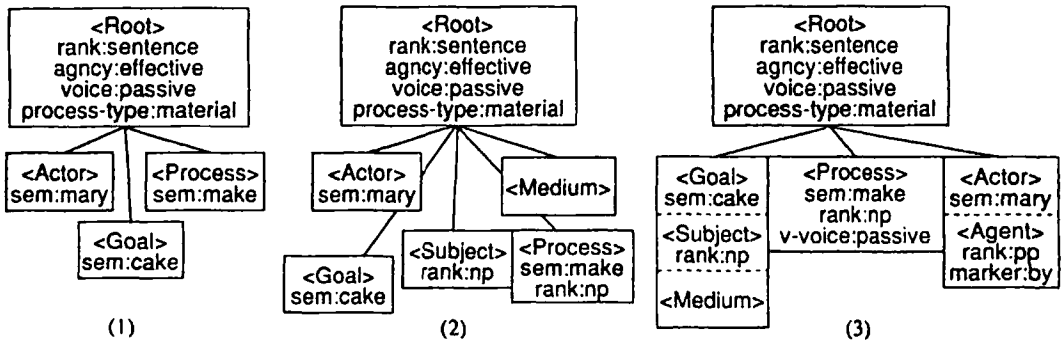

Figure 3: An example of generation process

It may be worth pointing out here that a system network includes both the grammatical and lexical knowledge. As shown in figure 2, grammatical features are primarily organized in terms of the RANK system. Lexical features are organized in the parts of the lowest ranks: noun, verb, etc.

Although, as mentioned above, a system network is a purely declarative description of a language resource, we could employ it as a set of rules for surface generation. For example, given a set of grammatical features representing the semantic meaning of a sentence to generate, the generator would refer to the system network to accumulate a set of constraints on the structure of that sentence. This process may be as follows. Let an example input be a structure as shown in figure 3 (1), where <Actor>, <Goal> and <Process> are the constituents of the sentence denoted by <Root>. The generator first traverses the system network to accumulate the constraints on the structure of the level of <Root>. Since the feature to choose in the RANK system is sentence, the constraints such as +Process and +Medium come into consideration. This makes the structure like in figure 3 (2), where the new constituents, <Medium> and <Subject>, are introduced and the order of <Subject> and <Process> is decided. After traversing the overall network, the structure will be like in figure 3 (3). Then, the generator traverses the system network for each of the constituents recursively. When this process is completed, the resultant structure will be a fully specified sentence.

The generation process is to enrich an input partial specification of a sentence to generate a full specification. This can be partially done by the knowledge described in the realization statements. But note that if the feature passive had not been specified in the input in the above example, the choice on the voice would have been arbitrary as long as the generator had had no knowledge but that in the system network. Strategies for grammatical/lexical choices are out of the scope of system networks. Then, why are we making efforts to develop them? Our answer would be that we believe exploring the knowledge as in system networks is the first but essential step toward exploring the knowledge of strategies for grammatical/lexical choices. It seems a vain attempt to design a decision scheme without any considerably clarified knowledge about the organization of the choice points on which those decisions will be made.

2.2 Functional unification grammar

A system network is a purely linguistic description of lexico-grammar, which is completely independent of the ways to implement it for computational NLP systems. As mentioned in section 1, however, when developing a grammar, the user may want to actually implement and use it for testing.

There have been several proposed methods to implement system networks for NLP. Some of them are to translate the description of a system network into constraint-based formalisms such as functional unifica-

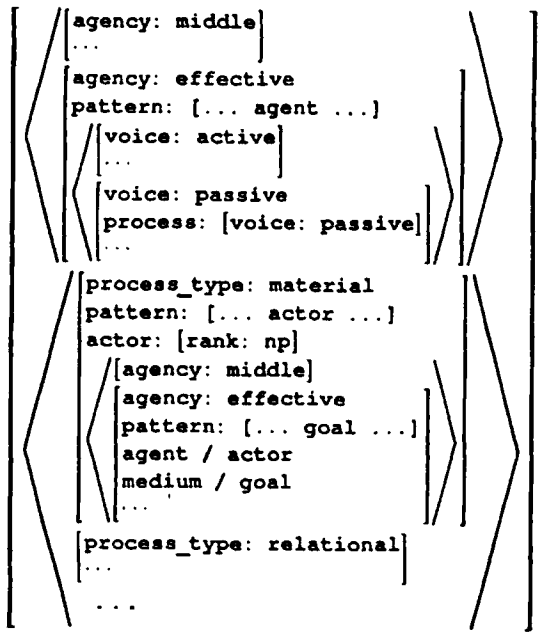

Figure 4: A part of the translation of the system network in figure 2

tion grammar (FUG) [6], typed feature structure (TFS) [2] and the knowledge representation language LOOM [7]. The knowledge represented in those formalisms can be easily consulted by general unification-based deduction systems. Furthermore, the declarativeness of the system network description is preserved in those formalisms. GENESYS adopted FUG for the internal representation of system networks.

The data structure handled in FUG is called a functional description (FD). A FD consists of a set of attribute-value pairs and disjunctions of FDs. A value is either an atomic value or a FD. In the FUG framework, both grammars and inputs are represented in FDs. In this paper we call them grammar functional descriptions (GFDs) and working functional descriptions (WFDs) respectively. When generating a sentence from an input WFD, the generator unifies it with the GFD to enrich it. The unification proceeds recursively with respect to the constituents in the WFD.

2.3 Translation of SFG into FUG

The algorithm translating SFG into FUG is based on Kasper's [6]. Figure 4 shows the translation of the AGENCY, VOICE and PROCESS TYPE systems in figure 2. As Kasper showed, a system of a system network can be translated into a disjunction of FDs, where each FD corresponds to an alternative in that system. A simultaneity condition of systems is naturally represented by a list of disjunctions. Translation of realization statements also follows Kasper's method. For example, a *preselection* constraint is translated into an attribute-value pair; e.g., Actor: RANK:pp into `actor: [rank:pp]`. A conflation of constituents is realized by unifying the FDs corresponding those constituents. In figure 4, `agent/actor` and `medium/goal` denote conflations. The existence of a constituent can be represented by a reserved attribute `pattern`; for example, +Agent is translated into `pattern: [...agent...]`. "..." in `pattern` attributes denotes an arbitrary sequence of constituents. A constraint on ordering constituents is also translated into a `pattern` attribute.

A gate requires more complicated translation. As pointed out in several papers [3, 6], gates cannot be straightforwardly represented in the FUG notation. Kasper modified the original FUG notation to represent gates. Elhadad, on the other hand, introduced the notion of *type* into the attribute-value representation, which turned out to be close to TFS. In the current implementation of GENESYS a gate constraint is simply represented by an additional disjunction. In figure 4, for example, the gate doing is translated the disjunction embedded in the FD corresponding to material.

3 The components

Figure 5 illustrates the components of GENESYS and its interactions with the user. In this environment, the user develops grammars in the following cycle.

1. **Editing a grammar:** The user draws a system network on the system network editor. The specifications of the network are translated into the FUG formalism.

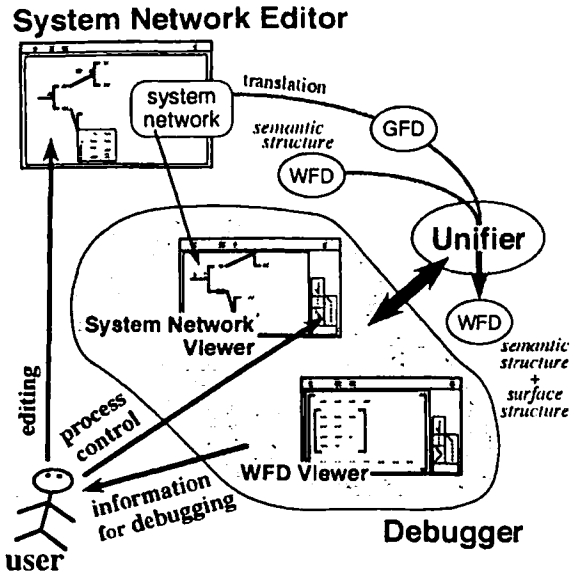

Figure 5: The system overview

2. **Detecting problems:** The user examines the grammar by running the unifier. Observing the generation process in detail, the user would detect the problems, if any, in the grammar. If some problems are detected, the user goes back to the editing phase to modify the grammar.

The system network editor and the debugger are tightly coupled and the user can freely move from one to the other. This is one of the significant features of GENESYS.

The basic components of GENESYS are implemented in SICStus Prolog and the graphical user interface is implemented in Tcl/Tk [11]. We used the ProTcl[2] library to make an interface between Tcl/Tk and Prolog.

3.1 The system network editor

The system network editor is a tool for defining system networks. As mentioned in the previous section, a system network is a typology of grammatical features; therefore, what is important in developing a system network is to grasp the topology of the overall network. The system network editor provides a GUI on which the user can efficiently define the topological specifications of a system network.

[2] ProTcl is a freeware developed by Micha Meier (micha@ecrc.de)

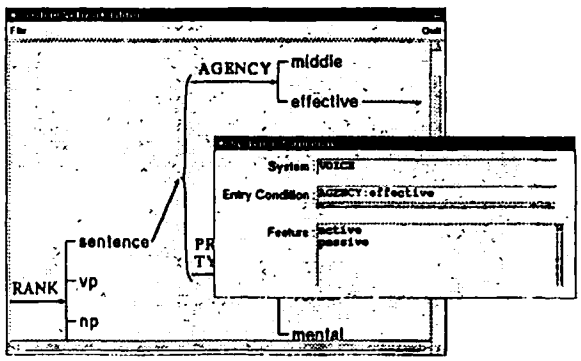

Figure 6: The system network editor

An elements of a system network is either a system, a feature, a simultaneity condition or a gate. To draw a network, the user first specifies the locations of systems, simultaneity conditions and gates on the canvas through mouse operations. For each of them, the user specifies its name[3] and its entry condition on the dialog window. The entry condition of an element is a conjunction of its mother element(s). Further, for each system, the user specifies its alternative features and their realization statements[4]. Figure 6 shows a snapshot where the user is defining a system named VOICE. Its entry condition has been defined to be effective of the AGENCY system. Entry conditions can be specified by means of either keyboard typing or mouse operations.

When constructing the network, the user can utilize the following functions as well.

- It is assumed that the names of the systems are all identical. When this constraint is violated, the editor gives a warning.
- The user can open a window for writing comments at an arbitrary place. Using this function, the user would write down example sentences for each feature.

The definition of a network is saved in the Prolog database, which is then translated into the FUG formalism. The translator was implemented in Prolog.

[3] For simultaneity conditions and gates, their names are not always necessary to be specified.

[4] Realization statements can be also associated with gates

```
agency(middle,Path,WFD) :-
  ...
agency(effective,Path,WFD) :-
  unify(Path,WFD,pattern:[dots,agent,dots]),
  attribute_value(Path,WFD,voice,Value),
  voice(Value,Path,WFD).

voice(active,Path,WFD) :-
  ...
voice(passive,Path,WFD) :-
  unify(Path,WFD,process:[voice:passive]),
  ...

...
```

Figure 7: Precompilation of FUG into Prolog rules

3.2 The unifier

The unifier unifies an input FD with a GFD. An input FD is assumed to be a partial specification of the grammatical features of a sentence to generate. The unification process enriches an input FD to produce a full specification of the output sentence. An actual sentence is generated by linearizing lexical specifications of the output FD.

If the input is under-specified, more than one alternatives will be valid. Unification by the unifier proceeds in the depth first fashion. At each choice point, i.e. in each system, the alternatives are preferred in the static order described in the system network.

Our implementation of the unifier is similar to that of the PFUF system implemented by Fasciano et al. [4] in the following respects.

- The system precompiles the descriptions of a GFD into a set of Prolog rules. Since Prolog rules for every predicate are disjunctive, a disjunction of FDs can be naturally realized as a set of Prolog rules. For example, the FDs corresponding to the AGENCY and VOICE systems, shown in figure 4, are precompiled into the predicates as shown in figure 7.

- As shown in the example, the first argument of each predicate functions as its index. These indices make double-hashing of SICStus Prolog quite effective.

3.3 The debugger

Unexpected failures of unification are usually good clues to the locations of the problems in the grammar. The GENESYS debugger has two major functions that facilitate monitoring the generation process:

- controls the trace of the generation process, and

- display graphically of the generation process' intermediate states.

3.3.1 Control of the generation process

Prolog interpreters provide debugging environments in which the user traces resolution processes. Similarly, in GENESYS, the user can trace generation processes by the mouse operations on the debugger windows. The information of the process is presented by the system network viewer. The debugger accepts the following commands.

- **creep:** to proceed to the next primitive step. A primitive step roughly corresponds to unification of an attribute.

- **skip:** to skip the process of traversing all the parts subsumed by the current system or alternative. For example, given the skip command when the unifier enters the PROCESS TYPE system in figure 2, the unifier continues traversing the whole part under the current system, such as doing and doing-material.

- **undo:** to go back to the latest state.

- **leap:** to proceed the whole generation process.

3.3.2 The WFD viewer

The WFD viewer graphically shows the intermediate state of WFDs (figure 8). It has several optional functions that facilitate information retrieval from WFDs.

- WFDs are usually much larger than the size of the window. The window automatically focuses on the constituent

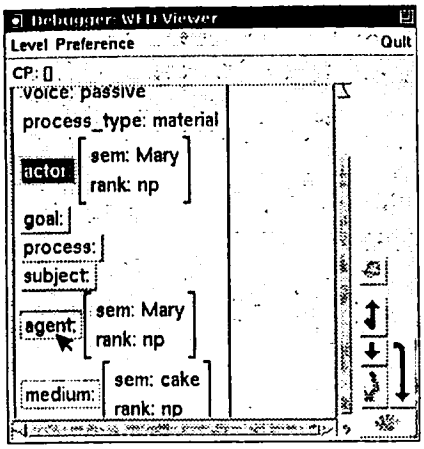

Figure 8: The WFD viewer

where unification is proceeding so as to follow the user's attention. The path of the current constituent is shown on the top; e.g. CP: [ ].

- The user can close FDs that seem to be unnecessary for debugging. The values of goal, process and subject are examples of closed FDs.

- WFDs tend to include a number of conflation relations between their constituents. The WFD viewer shows which constituents are conflated. Figure 8 is a snapshot where the viewer shows that agent is conflated with actor. While the mouse cursor is pointing at a feature, the feature conflated with it is kept highlightened.

3.3.3 The system network viewer

The system network viewer displays the progress of traversing the network (figure 9). For each constituent the user can open a window, which provides information such as:

- the path of the constituent,

- the chosen elements, i.e. systems, features, etc., and

- the realization statements associated with the chosen features.

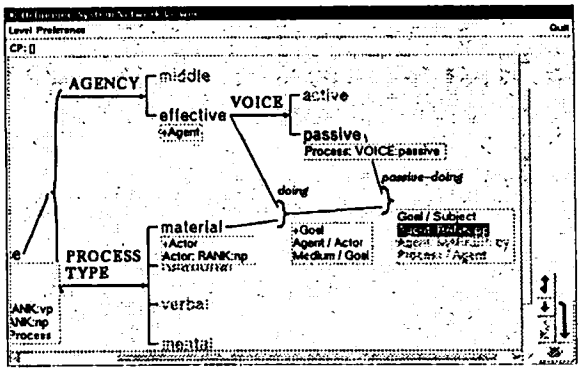

Figure 9: The system network viewer

4 Detecting problems: an example

In this section, we will give an example of detecting problems of a developed grammar. We will use the grammar given in the previous section.

Let us go back to the state shown in figure 9. It is the snapshot where the unifier is about to merge the constraint Agent: RANK:pp (highlitened by reversed characters) with those already in the current WFD. At this time, the features effective, passive, material and doing have already been traversed as shown the figure 9 and the associated constraints have been accumulated in the WFD as shown in figure 8.

Note that in the current WFD the value of rank in agent is np, while the system network is to impose the constraint that it must be pp. This inconsistency causes a failure of traversing and triggers backtracking. If this failure is unexpected for the user, its cause should be identified.

A good way to look for the cause is to explore when the existing constraint rank:np was imposed. This is easily done by employing the WFD viewer and the system network viewer. As mentioned above, the WFD viewer facilitates retrieving the existing constraints. And the system network viewer highlightens the chosen features and the realization statements associated with them. In this example, the window in figure 8 shows that agent is conflated with actor, and the window in fig-

ure 9 shows that the constraint rank:np was imposed when the unifier passed the feature material. Based on these pieces of information, the user would make modifications to the current grammar such as deletion of the statement Actor: RANK:np.

5 Conclusion

In this paper we have reported on the GENESYS system, which provides an integrated environment for developing systemic functional grammars. In this environment, the user constructs a system network with the editor designed for this purpose, and examines it by running the surface generation module, i.e. the unifier. The user can easily monitor the generation process through the GUI of GENESYS.

There have been several reports about systems that support development of systemic grammars: HyperGrammar[10], the Kyoto University systemic grammar development facility[1], and so forth. Compared with those systems, a distinctive feature of GENESYS is that its components cooperatively support the user by means of the GUI.

We might be able to refine the functions of the debugger. It would be more helpful, for example, if the debugger could give the information about when each constraint had been imposed on surface structures.

Acknowledgments

We would like to thank Prof. C. Matthiessen and his research group for useful discussions and comments on SFG.

References

[1] J. A. Bateman. The Kyoto University systemic grammar development facility: A user's guide. Technical report, Kyoto University, 1987.

[2] J. A. Bateman. The nondirectional representation of systemic functional grammars and semantics as typed feature structure. In *Proceedings of the International Conference on Computational Linguistics*, 1992.

[3] M. Elhadad. FUF: the Universal Unifier – user manual, version 5.0. Technical Report CUCS-038-91, Columbia University, 1991.

[4] M. Fasciano and G. Lapalme. A Prolog implementation of the Functional Unification Grammar formalism. In *Proceedings of International Workshop on Natural Language Understanding and Logic Programming*, pp. 37–45, 1994.

[5] M. A. K. Halliday and R. Hassan. *Language, context, and text: Aspects of language in a social-semiotic perspective.* Deakin University Press, 1985.

[6] R. Kasper. Systemic Grammar and Functional Unification Grammar. In *Systemic Functional Approaches to Discourse*, chapter 9, pp. 176–199. Ablex, 1988.

[7] R. Kasper and M. O'Donnell. Representing the Nigel grammar and semantics in LOOM. Technical report, USC/Information Science Institute, 1990.

[8] W. C. Mann. An overview of the Nigel text generation grammar. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics*, pp. 79–84, 1983.

[9] W. C. Mann. An overview of the Penman text generation system. In *Proceedings of the National Conference on Artificial Intelligence*, pp. 261–265, 1983.

[10] C. Nesbitt. *Construing linguistic resources: consumer perspectives.* PhD thesis, University of Sydney, 1994.

[11] John K. Ousterhout. *Tcl and the Tk Toolkit.* Addison-Weasley, 1993.