SESAM - Simulating Software Projects

J. Ludewig, Th. Bassler, M. Deininger, K. Schneider, J. Schwille
Software Engineering Group, Dept. of Computer Science, University of Stuttgart, FRG
sesam@informatik.uni-stuttgart.de

Abstract

Teaching software engineering as well as researching in this area is very tedious due to the length and costliness of software projects. SESAM therefore is designed as a simulator for software projects, allowing students to gain reality-like experiences in project management and researchers to evaluate hypotheses on the mechanisms influencing software projects. This paper focuses on the basic assumptions for SESAM, its building blocks and the way hypotheses are affecting the simulation.

After a short description of the requirements for SESAM we introduce objects, attributes, actions, relationships between objects and hypotheses as its basic concepts. We present attributed graph grammars as a means for representing hypotheses. Finally we position our project with respect to related work, and we show its present state and future development.

1 Introduction

1.1 SESAM: A simulator

The process of software development has not yet been fully described and explained. There is no comprehensive and generally accepted model for the process of software development (Sommerville, [7], p. 6). Models exist only for some classes of software projects (e. g. waterfall model, rapid prototyping). The lack of a reliable basis complicates any research attempt in the area of software engineering. Software engineering knowledge consists of a great number of tiny fragments; the glue for these fragments, an overall model, has not yet been found.

What impact has this deficit in theory on the education of new software engineers? Upon graduation, they first have to gather experiences in different project tasks (software development, design, systems analysis etc.), before they are able (and trusted) to lead a software project. University education can shorten this "training on the job", but can never replace it.

SESAM ("Software Engineering Simulation by Animated Models") is intended to support both software engineering researchers and teachers. SESAM is a tool for simulating software projects. Its basic concept is borrowed from adventure games with the player leading a fictitious software project. The goal of the game is to successfully carry out and finish the project. During the game, the player will be confronted with complicating events: staff members resign, important tools are delivered late or with severe bugs, the client changes the requirements, and so on. Time is passing and money is spent, with no way for the player to cheat. There is no predefined path through the game—the player has to find his or her own way to project completion. It is left to him or her how to assign the workload of the project to the staff members. Figure 1 shows a prototype of the player's world (that is, the user interface). At the end of the project the game is rated. Strong and weak points of the player's project management are indicated—based on a scale that has been set by the model builder, along with other parameters of the game.

1.2 Aspects of SESAM

SESAM users belong to one of two groups. One group—the model builders, who experiment using the parameters provided by SESAM—aim at a better understanding and explanation of the various aspects of the software development process. The other group—the players—wish to gain experience in project management. Typical players are students, whereas the model builders are researchers.

Which gains can these groups expect from SESAM? For the model builder, there are the following aspects:

- SESAM contains a collection of precisely defined hypotheses about the software development process, whereas in the software engineering literature rules and causal correlations (hypotheses) generally are stated rather vaguely and ambiguously.
- Assumptions can be validated using SESAM simulation. This applies especially to the collection of hypotheses mentioned above.

For a player, there are different benefits:

- He or she can undergo reality-like project management experiences at low cost (simulation is inexpensive), in short time (a game proceeds much faster than reality) and at no risk (a failed SESAM project does no dam-

0-8186-2830-8/92 $03.00 © 1992 IEEE

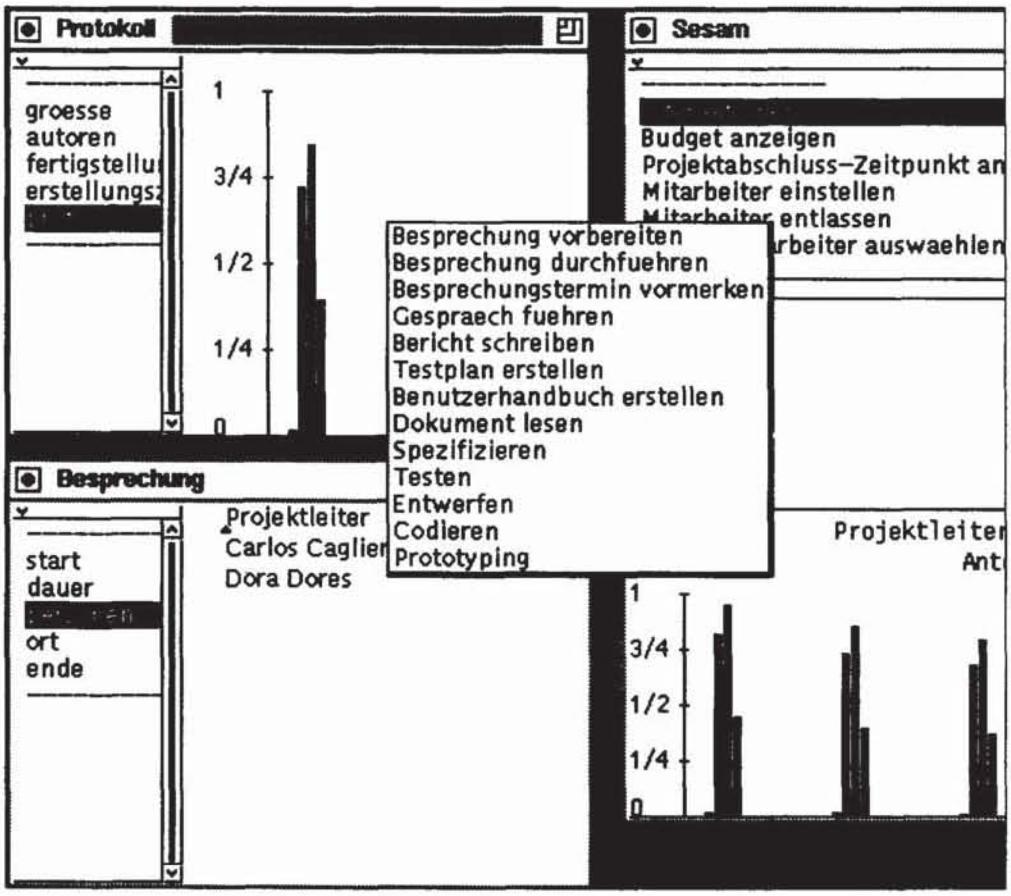

***Fig. 1* A game snapshot from SESAM:**
To the top left a window shows the minutes of a meeting (in the aspect "profile"), below another meeting is just in progress (here showing the people involved). The player has opened a pop-up menu for executing interactions.

age, while a failed project in reality might do immense damage).

- SESAM addresses the human instinct of play. In playing with SESAM, the student experiences the effect of his or her decisions. Learning by experiences is intuitive and therefore more efficient than teaching project management in terms of "good advice" during a course.
- SESAM is a forecasting tool. Using SESAM, various alternatives for a on-going project can be evaluated. In real life, there is no way to roll back time in order to alter a decision of the past.

1.3 Requirements for SESAM

From the aspects described above, we can conclude the following requirements for SESAM:

- SESAM has to provide the principal elements of a software project (staff, time, budget etc.), together with operations on these elements. Every important decision of a software project must be modelled in SESAM.
- Games have to proceed close to reality, i. e., if the player makes the same decisions during the game as were made in a real project, the results of the game must be comparable to the results of the project.
- The user interface has to be attractive and must facilitate the use of SESAM. SESAM has to be self-explanatory.
- Games must be repeatable, so that the player can try different alternatives of managing the same project.
- SESAM must give reasons for the rating of the player's game.
- SESAM must provide easy-to-use building blocks for the construction of models for the software development process.
- The model builder must be able to parameterize the building blocks, using a mechanism much like the effort multipliers of COCOMO [2].

2 Assumptions, concepts and roles

2.1 Basic assumptions

When building SESAM we assume that it is possible to model the software development process using objects, relationships between objects, actions and hypotheses, where objects and relationships can be seen as nodes and edges of a graph. *Objects* have *attributes* to represent their individual properties. The player can change *relationships*—i. e., the edges of the graph—by *actions*. Using *hypotheses*, it is possible to define changes for the structure of the network as well as for attributes of objects.

Objects, attributes, relationships, and actions will be discussed in the remainder of this chapter. Hypotheses and, based thereon, the modelling of regularities in software engineering are presented in chapter 3.

2.2 Building blocks

The objects to be found in a real life software project include the people involved, the documents to be used and produced, and all necessary operational reserves. *People* may be clients, project managers, software developers and so on. *Documents* comprise all of the software (i. e., specifications, design, code, testing plan, documentation etc.), but also any related contracts, standards, books. *Operational reserves* are, amongst others, budget, offices, computers or tools.

However, objects are not sufficient by themselves; they must have individual properties. Properties are stored in attributes, whose values may range from rough indications like "the design has a high degree of complexity" to exact quantities like "programmer X has a productivity of 2.3 LOC/h". Some examples for objects and attributes are:

- a person "project staff member" having the attributes
  - age (in years),
  - education ("BS", "MS", "PhD", ...),
  - design experience (measured by the number of projects he or she was involved in as a designer),
  - design skill (a number on a scale from 0 to 10);
- a document "high level design" with its attributes
  - size (number of characters),
  - complexity ("low", "standard", "high"),
  - quality ("low", "acceptable", "high");
- a "CASE tool" with its attributes
  - purchase-price (in currency units),
  - design method supported ("structured design", "object oriented design", ...).

Objects may be connected by relationships, which are symmetric or asymmetric (i. e. the corresponding edges

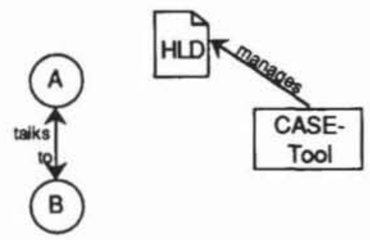

***Fig. 2* Effect of actions and hypotheses: The starting situation.**

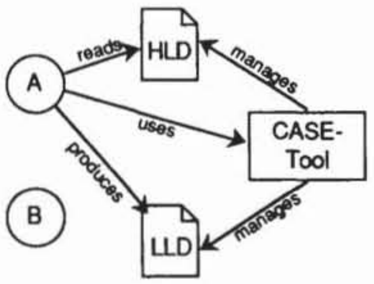

***Fig. 3* Effect of actions and hypotheses: The resulting situation after execution of the actions "terminate meeting" and "prepare low level design using CASE tool".**

are non-directed or directed, respectively). In the course of the software development process, relationships exist mainly as relationships in communications and interactions or as organizational relationships. Consider for instance:

- "reads": a staff member reads a high level design document,
- "manages": a CASE tool manages a high level design document,
- "uses": a staff member uses a CASE tool,
- "produces": a staff member produces a low level design document,
- "talks to": a possibly symmetric relationship—two staff members are talking to each other.

A distinct partial graph of the network of objects and relationships is called a *situation*. During a SESAM simulation the state of the network will be changed. This is achieved by actions and hypotheses. Both actions and hypotheses apply to a particular situation. They change relationships in that situation, i. e., establish a relationship between objects or remove it. They may also generate new objects. Furthermore, hypotheses may change attributes of objects in the situation considered. Actions are triggered by the player, who wishes to influence the proceeding of the game. Hypotheses are triggered by the simulator and without any intervention by the player, if there is a situation that matches the preconditions of the hypothesis. Changes effected by a hypothesis are determined by the (assumed) rules applicable to the current situation. We will clarify this by an example:

> Starting situation: The project is in its design stage; high level design (HLD) has been successfully completed. It has been stored via a CASE tool, which will be used throughout the project. Low level design (LLD) has not yet started. Project members A and B are in a meeting (i. e. are talking to each other). This state is shown in figure 2.
>
> The player now wishes to assign person A to LLD. A shall use the HLD document and the CASE tool. To this end the player executes the action "terminate meeting", which deletes the relationship "talks to" that connected A and B. A is now free and can work on LLD. The action "low level design using CASE tool" generates a new document "LLD" and establishes the necessary relationships as shown in figure 3.
>
> Based on this situation a hypothesis on the effect of using CASE tools for LLD fires. It changes the attributes of the objects involved, following the assumptions stated in the hypothesis. This might mean that the size of the LLD document becomes four times the size of the HLD document, and, because this is A's first project as a software designer, the quality attribute is set to "acceptable" although the quality of the HLD document was "high".

A more detailed discussion of hypotheses and their influence on the elements of our software engineering simulation follows in chapter 3.

2.3 Player, model builder and developer

Three roles are to be considered in SESAM—player, model builder and developer. The player gets a project generated by SESAM and has to use his or her acting alternatives as a project manager to successfully complete the project. He or she may apply actions to objects and thus change the network of relationships; the player may not, however, modify object attributes or relationships directly. Depending on the state of the network SESAM identifies situations and their matching hypotheses. This leads to changes of object attributes and relationships.

Consequently, the "project manager's" task is to select an appropriate action for the actual state of the project from the given set, and to have it executed. This generates a new state of the simulated project, which may trigger some hypotheses, again changing the state of the project. The action-hypotheses cycle will be repeated until the software product in question is completed.[†] Based on the game history and the final state, the quality of the project will be evaluated.

Objects and their attributes, relationships, and the mechanism to describe actions and hypotheses together form a construction kit. The model builder takes the elements from the kit to determine the actions the player may use, and to build the hypotheses to be triggered by the simulator. The set of hypotheses and actions make up the model of the software development process, which the model builder wants to control the game. Besides describing hypotheses and actions, the model builder generates the starting situation the player will face in his or her game, and sets up the rules for the final evaluation of the game.

It is the developer's task to provide the construction kit for the model builder and the mechanism of simulation for the player.

Summarizing the roles in SESAM: the developer produces the basic concepts, the model builder takes them to construct hypotheses and actions and to provide a starting situation for the game, the player uses the simulation environment and the starting situation to simulate a software project.

3 Hypotheses on software projects

3.1 What are hypotheses?

Objects and relationships build up the static structure of our software project model. But simulation of a software project with its complex internal interrelationships requires modelling its dynamic structure, too—how does the project proceed, how does the software product change?

Some reasons for changes in the project state[*] are quite obvious (e. g. the monthly payment of salary at a fixed date decreases the remaining budget) and may be hardwired within objects or the simulator. However, there are a large number of other interrelationships that influence the project state and thus the project evolution itself. In software engineering literature many of these are referenced, e. g., Sommerville ([7], p. 43) cites from a study stating that "the size of an organization correlates negatively with job satisfaction and productivity. It correlates positively with absenteeism and staff turnover." In contrast to the obvious reasons for changes of the project state mentioned above, such statements at first have the nature of a hypothesis—they are unproven scientific assumptions.

Only after validating these hypotheses empirically—or proving them in any other way—they may be taken for certain and built firmly into the model. In some cases, a

[†] Of course no real software product is produced in SESAM, just the model of a product that would have been produced, if one had taken the same decisions and executed the same actions in a real project.

[*] "Project state" means the whole of states of all objects involved in the project.

different wording or a variation of a hypothesis may prove to be more reliable, comprehensive or simply more correct. It is for instance not at all obvious, whether there exists a relation between the hypothesis above on correlation of job satisfaction and size of an organization, and the well known "Brooks's law": "Adding manpower to a late software project makes it later" ([3], p. 25). Do both hypotheses explain a common phenomenon? Do they just stress different aspects or are they in fact independent? Maybe one is a special case of the other? A hypothesis may also turn out to be obsolete, trivial or plain wrong.

Therefore we treat such statements with caution and separate them from "well proven" parts of the model. In SESAM surmises on interrelationships which are not proven are called what they are—*hypotheses*. We suppose that the "treasure of software engineering knowledge" mostly consists of such observations, surmises and conclusions, so they have to be represented as such in our model. But hypotheses are not always as explicitly stated and easily recognized to be surmises of interrelationships as the ones above. Sometimes they are contained implicitly in rules of etiquette and practices of software engineering—Structured Programming, Structured Analysis or object oriented methods are not used just for fun, but because they come with the promise of "improvement". Wherever, beyond this promise, there is no concrete statement of what kind of improvement is to be expected, scepticism is indicated. We hope to clarify such cases by building SESAM. SESAM will provide a facility for gathering hypotheses from a variety of sources and for studying their interaction.

3.2 Hypotheses are vague

SESAM will provide quantitative statements on the development and success of the software project simulated. To achieve this objective, all parts of the simulation model must be unambiguous, precise and quantifiable—even hypotheses. Therefore, for the purpose of simulation, each of the following questions must be answered for every hypothesis:

- Which situation must exist for the hypothesis to be applicable?
- Who and what is important for the hypothesis, i. e., who is involved, who or what will be influenced?
- Which consequences arise from accepting the hypothesis as an expression of a real life interrelationship?

Unfortunately, hypotheses mostly are vague or not stated explicitly at all. (E. g., what is the promise of structured programming? If stated explicitly, new hypotheses might show up.) Sometimes the author *is not able* to indicate the consequences of a hypotheses in full detail, because his or her observations are of only qualitative nature. In such a case, it might indeed be the goal of simulation to find a more precise wording for the hypothesis by rating and varying its parameters—or to unmask it as inconsistent or untenable.

3.3 Making hypotheses precise

The problem of finding an adequate representation for hypotheses places us in a dilemma:

- On the one hand, statements are to be formulated in accordance with their application domain—the software development process and its internal interrelationships.
- On the other hand, a formal, computable representation is indispensable.

Ignoring one of these two demands leads to unpleasant consequences:

- Too informal a representation is of no use for the purpose of simulation—a computer cannot evaluate it.
- Too formal a representation renders *human* handling of hypotheses difficult. It is hard to recognize whether a set of formal expressions correlates with the hypothesis it is derived from or whether some semantics have been added or taken away. Are precision and certainty pretended without a need for the simulation? What exactly implies this set of expressions if we translate it back to the language and manner of thinking of a software project?

It is necessary to fulfil both demands to avoid a Gordian knot of assumptions and suppositions that is impossible to validate. Therefore we decided to represent each hypothesis at three levels of formalization.

The most informal level is a simple citation—every hypothesis is stored in its original wording. The citation is indispensable for future validation, as the representations at more formal levels need a baseline to which they can be compared. This is the only way to avoid an inadvertent change in meaning. Of course, the reference for the citation has to be stored, too—this is not only a question of scientific honesty, but also facilitates later checking of the context of the hypothesis. Sometimes the traded aphorisms—called hypotheses—can be fully understood only in their proper context!

The intermediate level uses a formatted representation. The statement is split in its main components in order to remove any grammatical wrapping. The aspects represented in this format are shown in figure 4, its use is illustrated in figure 5.

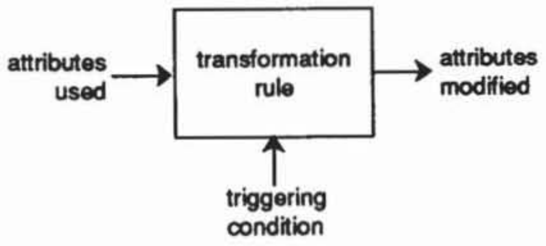

Fig. 4 **Formatted representation of hypotheses.**

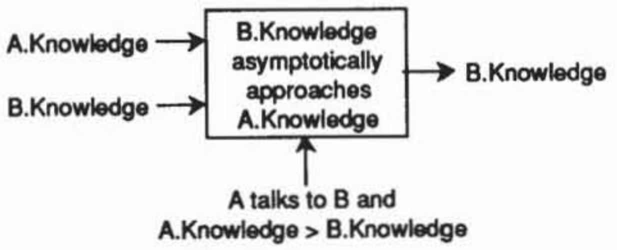
A.Knowledge
B.Knowledge
B.Knowledge asymptotically approaches A.Knowledge
B.Knowledge
A talks to B and A.Knowledge > B.Knowledge

Fig. 5 **Formatted representation of the hypothesis: „If A talks to B and A's knowledge exceeds B's, then B's knowledge will asymptotically approach A's."**

Wherever possible, we identify objects as well as their attributes that are considered in the statement. Apart from these we collect those objects and attributes which are influenced by the interrelationship stated. A *triggering condition* determines which circumstances must exist for the hypothesis to "fire", i. e., for which situation it fits. Such circumstances include the project state, certain events, a specified point of time, or any combination of these. The interrelation of the objects involved is described as a *transformation rule* (e. g. linear or exponential correlation, or just a qualitative description). It is important at this point not to introduce one's own assumptions, parameter ratings or attempts at precision, which are not fully covered by the hypothesis.

The most formal level includes detailing the effect of the hypothesis in full detail, establishing parameter values, and resolving all inconsistencies that may have been discovered at the intermediate level. This implies decisions on all aspects that cannot be taken directly from the hypothesis—and to document them! Finally the hypothesis is represented by one or more productions of an Attributed Graph Grammar (AGG).

The state of a simulated project is represented as an attributed graph. As explained in chapter 2.1, this graph consists of objects (nodes) and their relationships (edges). Hypotheses change the project state and thus the graph. Starting situation and resulting situation of a hypothesis are partial graphs of the project state graph. In AGGs the corresponding transition is represented directly as a graph production. The following paragraphs present the basic concepts behind AGGs; a more detailed discussion is given in [4] and [6].

Attributed Graph Grammars are a generalization of Chomsky grammars. The latter deal with sequences of symbols, to which the following operations are applied:

1 Identify a part of the sequence of symbols that matches the left side of the grammar production.
2 Substitute the subsequence of symbols identified in 1 with the right side of the production.
3 In attributed grammars, attributes of all symbols contained in the production (both left and right side) may be modified.

Graph Grammars deal with graphs consisting of nodes and edges instead of sequences of symbols, to which analogous operations apply:

1 Identify a part of the graph that matches the left side of the graph grammar production (i. e., for which there is an isomorphic mapping to the left side of the production).
2 Substitute the partial graph identified in 1 with the right side of the production (i. e., cut out that partial graph and insert the graph on the right side of the production in its place).
3 In an AGG, attributes of all nodes and edges contained in the production (right side only!) may be modified.

Note that, in step 1, for AGGs attributes of nodes or edges may also be used in the identification of a matching partial graph (i. e. the search is not restricted to structural aspects of the graph).

Hypotheses are formally represented by graph grammar productions. The left side of a production—the precondition of a hypothesis—shows the situation that causes the hypothesis to be applied, i. e., the project state triggering the hypothesis and the objects and relationships to be identified. Many hypotheses will not affect the structure (step 2), whereas almost all of them will change object attributes. If no substitution of partial graphs is necessary, step 2 is omitted; however, there are cases which require changes to the graph, e. g., if communication relationships have to be established or removed.

Two examples illustrate the difference. Figure 6 shows the changing of communication relationships. The starting situation is a meeting of three staff members, one of whom "has a headache". The production applied to this situation, however, does not state anything about meetings, just about the relationship "A and B are talking to each other". This relationship will be disrupted, if A has a headache. The result of two applications of this same production leads to a situation, in which the person with a

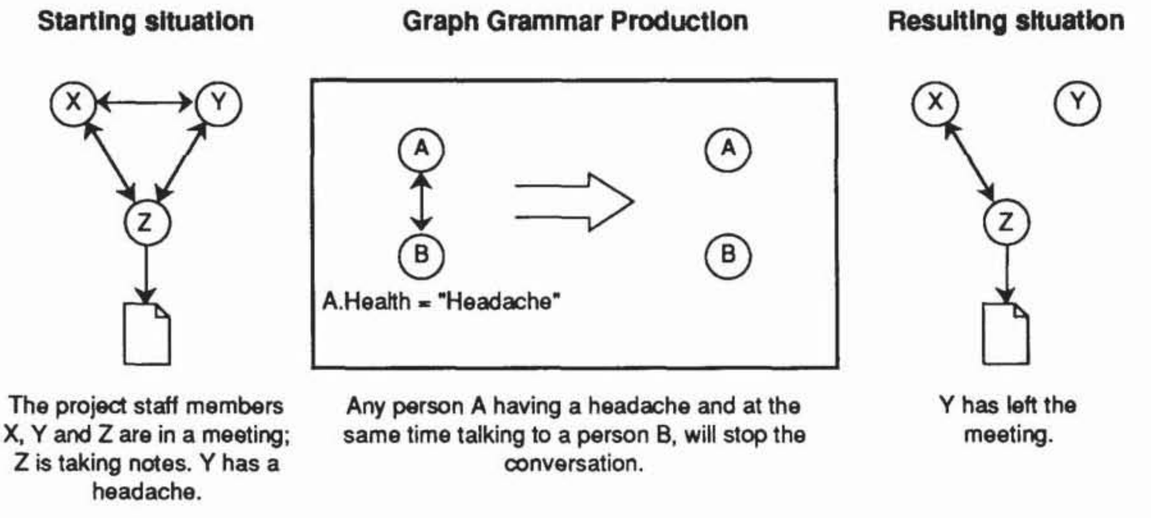
Starting situation
Graph Grammar Production
Resulting situation
X
Y
Z
A
B
A.Health = "Headache"
A
B
X
Y
Z
The project staff members X, Y and Z are in a meeting; Z is taking notes. Y has a headache.
Any person A having a headache and at the same time talking to a person B, will stop the conversation.
Y has left the meeting.

Fig. 6 **Changing (communication) relationships using Attributed Graph Grammars.**

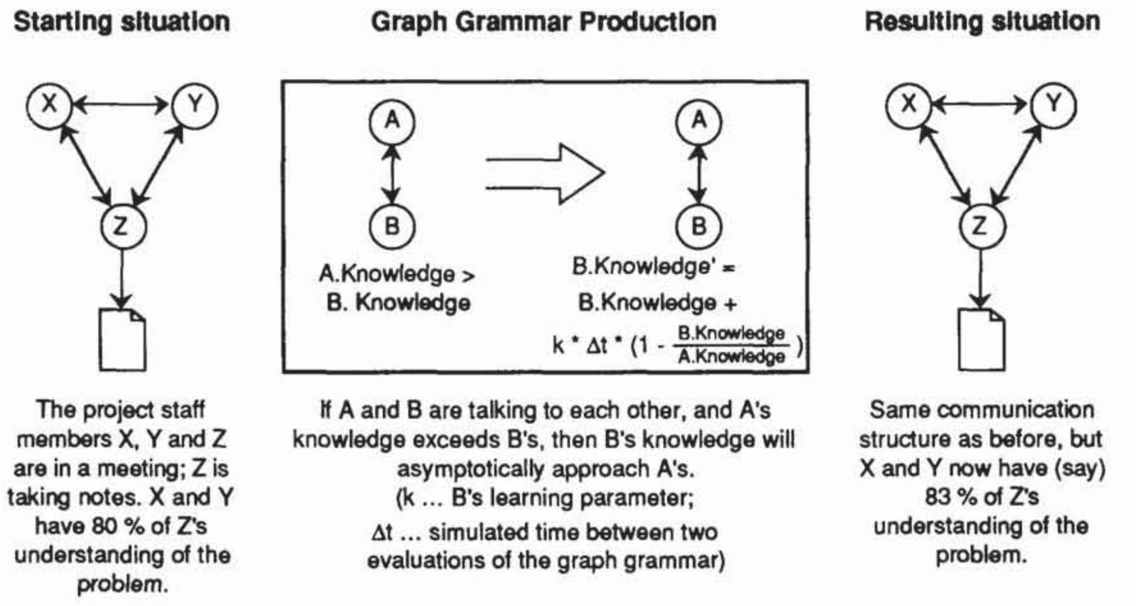
Starting situation
Graph Grammar Production
Resulting situation
X
Y
Z
A
B
A.Knowledge > B. Knowledge
A
B
B.Knowledge' = B.Knowledge + k * Δt * (1 - B.Knowledge / A.Knowledge)
X
Y
Z
The project staff members X, Y and Z are in a meeting; Z is taking notes. X and Y have 80 % of Z's understanding of the problem.
If A and B are talking to each other, and A's knowledge exceeds B's, then B's knowledge will asymptotically approach A's. (k ... B's learning parameter; Δt ... simulated time between two evaluations of the graph grammar)
Same communication structure as before, but X and Y now have (say) 83 % of Z's understanding of the problem.

Fig. 7 **Changing object attributes using Attributed Graph Grammars.**

headache is no longer in a communication relationship with the others—to be interpreted as "left the meeting".

The other example (figure 7) shows the mere changing of object attributes. The hypothesis of asymptotically approaching knowledges, which has already been shown in figure 5, is represented here as a production of an AGG. It applies to a similar situation as in the preceding example (this time there is no headache involved). The result is an increase of knowledge that depends on the duration of the meeting (more precisely, on the simulated time that has passed since the last evaluation of the graph grammar) and the individual learning capability (which is assumed equal for person X and Y in this example).

Apart from the situation itself, events and the simulated time may have an impact on the applicability of a hypothesis/production (i.e., may be part of the triggering condition of a hypothesis).

Attributed Graph Grammars provide an easy way of representing hypotheses formally, but nevertheless in an intuitive way. They allow to do this in terms of the application domain—an immediate graphical representation of software project situations.

The formally defined semantics of AGGs [4] permit an automatic generation of production rules for a rule-based inference tool from graph grammar productions, or to have graph grammar productions interpreted right away.

We see this method of stepwise formalization of hypotheses as a promising way to a more precise and comprehensive notion of the presently vague and not interrelated suppositions of software engineering.

4 Present state and future development

4.1 Related work

Simulating software projects is not a new idea. COCOMO [2] shows how development effort and development time correlate with program size and other influencing parameters. The COCOMO model can (and must) be calibrated by setting these influencing parameters, the so-called effort multipliers. We adopted this idea for SESAM. McKeeman [5] reports a tutoring program for the training of software developers. Using the program, developers learn how to conduct a review. This program has the nature of a game, much like SESAM. Abdel-Hamid [1] describes a simulation model for the software development process that is based on System Dynamics. As with SESAM, this model allows conclusions about software projects. However, all of these approaches are restricted to certain aspects of software development, whereas SESAM is based on a comprehensive approach. Depending on the hypotheses and other elements used by the model builder different aspects may be investigated.

4.2 Present state of SESAM

So far two prototypes of SESAM have been developed. The first one was based on COCOMO and turned out not to be extensible. The second prototype already implements most of the concepts presented in this paper. The examples for the user interface given in chapter 1 were prepared using this prototype. At the same time, a collection of hypotheses from available software engineering literature was initiated, resulting in a list of 242 hypotheses. We are currently working on an implementation of the simulator kernel for SESAM, to be completed gradually to a fully operational system. The implementation environment is Smalltalk-80 on UNIX workstations.

4.3 What remains to be done?

The second prototype we have available now is used to illustrate and validate our concepts. Many aspects are not yet considered in this prototype. Following is a list of topics we will address next, ordered chronologically:

- The collection of hypotheses must be separated from the SESAM simulator. To this end, we have to implement a procedure to convert hypotheses from natural language to productions of an Attributed Graph Grammar.
- We need to develop a model for the software development process, implement it using SESAM and validate it. This model determines the actions available to a player. Furthermore we will identify shortcomings of SESAM while developing the process model, and we will be able to specify the necessary improvements and extensions.
- At the end of a SESAM game, the project history must be rated and the player must be given a helpful foundation of the rating.
- During the game, the player will be able to access a software engineering data base giving descriptions of methods and techniques for software engineering. He or she shall have the chance to learn about possible alternatives before making a decision.

5 Bibliography

[1] Abdel-Hamid, T.K., "Investigating the Cost/Schedule Trade-Off in Software Development," *IEEE Software*, vol. 7, no. 1, pp. 97-105, January 1990.

[2] Boehm, B.W., *Software Engineering Economics*, Englewood Cliffs, New Jersey: Prentice Hall, 1981.

[3] Brooks, F. P., *The Mythical Man Month*, Reading/Mass.: Addison Wesley, 1975.

[4] Göttler, H., *Graphgrammatiken in der Softwaretechnik*, Berlin: Springer, 1988.

[5] McKeeman, W.M., "Graduation Talk at Wang Institute," IEEE Computer, vol. 22, no. 5, pp. 78-80, May 1989.

[6] Nagl, M., *Graph-Grammatiken—Theorie, Implementierung, Anwendungen*, Braunschweig: Vieweg, 1979.

[7] Sommerville, I., *Software Engineering*, Wokingham: Addison-Wesley, 3rd Edition, 1989.