Behavior Abstractions to support Reverse Engineering[*]

Dominik Rauner-Reithmayer and Roland T. Mittermeir
Institut für Informatik-Systeme, Universität Klagenfurt, Österreich
{dominik, mittermeir}@ifi.uni-klu.ac.at

Abstract

Most approaches for recovering objects from procedural code are exclusively based on static information. These approaches have the advantage to build on information easily available. However, with code that is not built around very strong and ubiquitous data structures, substantial portions of this code cannot be clearly assigned to one of the objects the object-oriented system is made of.

Here, we discuss an approach where the uncertainties that necessarily appear with purely data-structure driven approaches can be reduced by establishing in addition to the static object model a dynamic object model. The merits of the approach extend object recovery though. We consider the creation of event-state diagrams also useful for software redocumentation as well as for general program understanding.

keywords: object recovery, program understanding, redocumentation, software reuse

1. Motivation

One of the key question that links software maintenance, software reverse engineering, and software renovation is to uncover and finally understand the principles used by the original developers of the legacy software at hand. Here we approach the issue from the background of the *CORET*-project, a project aiming at the conversion of procedural legacy code to object-oriented code.

The transformation from the procedural to object-oriented code is meant to be not only on a syntactical level, but in order to enhance the evolvability of the renovated code, semantically coherent objects are to be formed.

In order to achieve this semantic coherence and to arrive at software with structure conformant to the standards of well designed modern object-oriented software, a coarse object-oriented application model is developed in a conventional forward engineering manner. This object-oriented application model (OOAM) defines the target architecture and serves as reference point, against which the procedural legacy code is mapped. The ensuing transformation process has to transform and regroup the contents of the various legacy procedures in a semantics preserving manner into the new structure (see Figure 1) [8, 14]. To achieve this aim,

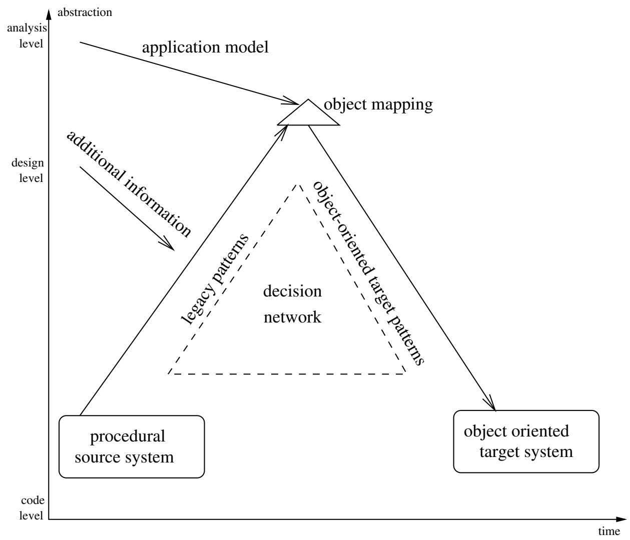

Figure 1. CORET Structure

one first has to identify object candidates in the legacy code and match them to the classes identified in the forwardly developed OOAM. Such candidates can be found in the basic data structures of the legacy code. Various strategies exist to do so [2, 3, 4, 5, 11, 13, 18]. These are relatively successful. However, there remain problems due to the fact that the basic conception of the old, procedural software and the new object-oriented structure is different and also due to the fact that different levels of abstraction and different naming strategies are followed [7]. Therefore, the relationship between forwardly defined and reversely identified objects can initially only have the quality of a proposal. Whether this proposal will conform to a well-designed object-oriented structure can be seen only, when it is possible to suitably group the "executable" part (body) of the legacy source code

* This work is funded by the Austrian Science Fund (FWF), grant Nr. P 11.340-ÖMA.

"around" those candidate objects in the form of semantically coherent methods.

Since one has to assume that the procedural legacy code has been structured according to criteria that are quite different from those one uses when developping object-oriented code, this "grouping code around objects" is a non trivial task. In general, one cannot take procedures readily available and simply redefine them, so that they conform to the syntax of methods. The procedures found in the legacy code will relate to several candidate objects. Hence they have to be split in such a way that single minded methods result. This has to be done by chunking this "ececutable" part into independently comprehensible pieces [1] and assigning those to the candidate objects in such a way that finally the resulting object oriented system is behaviourally equivalent to the old legacy system [15].

As this chunking depends on the set of object candidates identified in the previous step, the performance of this process depends on the quality of the original object identification. As mentioned, various approaches have been described to identify object candidates in legacy code [4, 3, 2, 18, 5, 11, 13]. In spite of their differences, however, they all focus just on static information i.e. on information provided in the declarations part of the legacy code. Method construction, and hence the assignment of the "executable" portion of the legacy code, however, has to do with the dynamic properties of objects, i.e. their behavior. Here, we present an approach to uncover the behavior of "objects" hidden in non-object-oriented legacy code. Although we discuss this from the perspective of object recovery, the ideas behind this approach seem also to be adequate for program understanding of non-object-oriented code in general.

In the rest of this paper, we first discuss briefly object-oriented modelling. Then we explain how a dynamic object model can be recovered from classical procedural legacy code and described in OMT terms. Performing such recoveries, one realizes that the direct approach described in section 3.1 reaches its limits, when the implementer of the legacy code decided to encode static aspects in procedural interdependencies. These aspects, referred to as indirect state changes, are sketched in section 3.2. Having thus obtained a complete and coherent dynamic model, integration issues and issues of how to deal with different abstractions on the source code and in the high level OOAM are discussed. Before concluding, the paper gives an outlook on how the dynamic model can also be useful in the constructive phases of renovation projects.

2. Object-Oriented Modelling

To describe object-orientation in a nutshell, one generally refers to the concepts of state spaces, of state encapsulation, message passing and inheritance. Object oriented modelling techniques cover these in different ways. We discuss them here on the background of OMT, the Object-oriented Modelling Technique [17].

In OMT, a system is described in terms of three different "models". The *static model* defines the state space in terms of the various attributes constituting state space dimensions, and the methods (names of methods and their arguments) encapsulating this state space. The *dynamic model* shows under which conditions (events) changes in the actual state occur and which sequences of states (and state transitions) are permissible. The *functional model*, complemented by scenarios, finally, shows how certain application functions can be achieved by sequences of method invocations. Thus, the three models can be considered as different views on parts of the same system. Certain redundancies one can identify among these models are important to check for internal model consistency.

In this context, the dynamic model has particularly the role to show, under which conditions certain state transitions might be performed. As the state space of OMT-objects is usually complex, the dynamic model will also show, which state dimensions can be modified independently. In this capacity, it becomes an important device for reviewing the static model. E.g., if there is no interaction among some state dimensions, the question might be raised, why these aspects pertain to the same object. If the state transition diagram is not a connected graph, the modeller knows that the model is either incomplete or contains information pertaining to different objects or at least to objects considered from different role-perspectives.

Nonconnected state transition diagrams are never shown in textbook examples, as they are inconsistent. In real analysis situations, information elicited from different subjects might quite often lead to such situations though. They have to be resolved later by the modeller. In object recovery, the problem is to identify code that "belongs" to object candidates. As this identification and mapping problem is plagued with uncertainties and has to proceed from partial information, the same problems can and will occur.

The approach we are describing rests on this very principle: A well defined state space is described not only by its dimensions (attributes) but as well by the coherence of its state transition, the information captured in OMT's dynamic model. If, for a set of candidate objects, no consistent dynamic models can be derived, this set is apparently dynamically inconsistent. If though, consistent dynamic models can be derived from the legacy code for all candidate objects, a basis for chopping up this legacy code and regrouping the chops into methods is given.

How such a dynamic model can be derived is explained in the next section.

3. Dynamics on the Code Level

3.1. Basic Idea

Like other researchers dealing with object recovery, we consider the overall state space of a system represented by the variables declared within this system. Hence, the state space of an object is defined by the variables bound to the respective candidate object. Hence, state transitions are modifications of the value of defined variables. Events, triggering such state transitions are all those programming constructs that potentially lead to changes of the value of the respective variables. On an elementary level, this can be done in imperative programming languages only by an assignment statement.

In actual code, the assignment performing a state change need not be directly visible. It can be contained in whatever form deeply nested in some procedure. Hence, procedure invocations have at least the potential to lead to state transitions if the variable in question is in the scope of this procedure. For the further discussion, one has to note though, that the actual invocation of a procedure (and likewise the execution of an assignment) can be identified only at run time. On the basis of the source code, we can only identify procedure calls and assignment statments that, depending on the context in which they appear, might lead to an invocation.

To simplify the discussion, we will refer to the variable (or variables) capturing the state of an object as *state-variable*. This leads to the definition:

> A call of a procedure that defines (modifies) or uses the state-variable constitutes an *event* on the respective object.

This definition implies that we consider state changes only on the level of procedures. The occurence of an assignment statment itself within the body of a procedure is not referred to as event, since it is too fine grained, i.e. it has no counterpart on the level of the OOAM. As consequence of this decision, we cannot say that each occurrence of a definition of the state-variable leads necessarily to a state transition. Whether such a state transition will occur depends on the context of the respective assignment. With context, we refer here to structual considerations as much as to aspects of the actual values variables might hold. We refrain from an in depth discussion of this issue though, since it seems fair to assume, that whenever a variable is (re-)defined, this redefinition has the semantics of a potential value change, hence a state change.

For reasons of simplification and for pragmatic reasons explained in section 4.2, we assume in the rest of this paper that definitions (modifications) of the state-variable principally occur within procedures. Thus the definition of events as given above and the definition of events as defined in OMT can be considered equivalent. Therefore, the following categorization of events can be made:

- *State changing event:* On each possible execution path through the procedure, at least one attribute of the state-variable is defined.
- *State preserving event:* On none of all possible execution pathes an attribute of the state-variable is defined, i.e. the variable is only used.
- *Complex event:* There are execution pathes on which the state-variable is defined, and others that leave the state-variable without (re-)definition.

We will see later, that the occurrence of a complex event will always lead to a complex state, i.e. a state consisting of more than one substates and their generalization. Hence, the choice of this term.

3.1.1 Primitives

Independently from the event classification based on its effects, one can classify events based on how they are identified within the source code:

- *creation:* The initialization of the state-variable usually happens in an initialization procedure. However, as initialization (and deletion) are key aspects to obtain a consistent dynamic model, we will relax our definition for these two events in so far, as we consider initialization (and deletion) activities as events irrespective of the specific syntactic form in which they are made. In an OMT diagram, the creation will lead to a welldefined start of a state transition diagram (see Fig. 2).

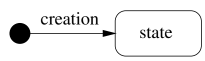

Figure 2. Object creation

- *deletion:* Is seen in the source code only, if such operations as 'dispose' are used. Otherwise, it is implicit when the extent of the state-variable is finally left. It leads to a terminating node in the OMT diagram (see Fig. 3).

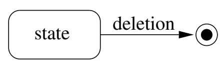

Figure 3. Object deletion

- *basic event:* This is either a state changing or state preserving elementary event. It is identified in the source code whenever a procedure call that directly and singularly uses or defines the state-variable appears. In the

OMT diagram it is represented as single arc. In case of a state changing event, this arc will connect different states. In case of a state preserving event, it will loop back to the same state (see Fig. 4). To identify the code pertaining to an event, a one-to-one correspondence between event names and procedure names is required.

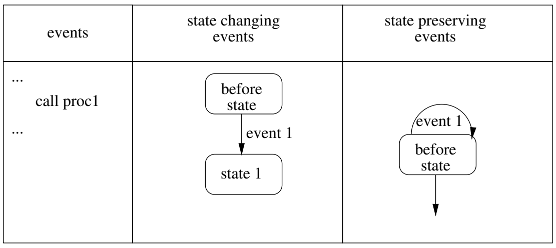

Figure 4. Basic event

3.1.2 Models for Control Structures

Here, we consider initially just control structures over state changing events or over state preserving events.

- *sequence of events:* This is identified, if within a compound statement a series of events on the same state variable occur. The effect is, that the state-variable will repetitively be defined or used. In an OMT diagram, the sequence of state changing events is represented as a chain of state transitions connected by the respective events, while state preserving events are represented as loops on the same state (see Fig. 5).

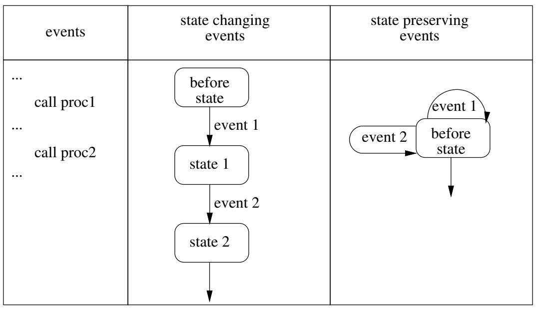

Figure 5. Sequence of events

- *alternative occurrence of events:* This is identified, if the event(s) occur within the branch of an alternation construct (IF, CASE, ..). Fig. 6 shows, that an alternative involving state changing events always leads to a generalization of states. If in all alternative branches state changes occur, the final state of the alternative will be the generalization of the (final) states resulting from the branches. If a branch without state changing event exists, the state reached on entry into the alternative construct is part of the generalization.

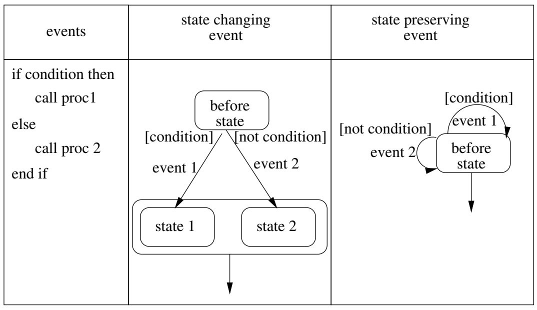

Figure 6. Alternative occurrence of events

- *iterative occurrence of events:* This is identified, if an event occurs within a repetitive construct (WHILE, FOR, recursion, ...). With repetitive constructs, there is always some repetition- (loop-)invariant. This invariant state is also the final state of the repetitive construct. Whether the before-state is in the same generalization or not depends on the particular semantics of the repetitive construct (see Fig. 7 and Fig. 8). One might note that the WHILE-construct with state changing events builds actually a link to complex events, since, in case the while-condition is initially false, before-state and after-state are in the same class and hence generalized to a higher level state comprising both (primitive) states.

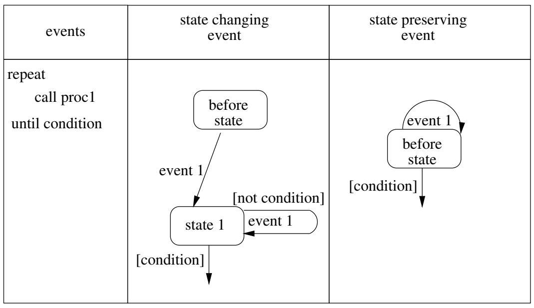

Figure 7. Iterative occurrence of events: REPEAT

3.1.3 Complex Events

Complex events result from control structures involving alternations with branches leading to state changes and other branches that leave the initial state unchanged. In the simplest version, such situations will occur with a one-sided IF-THEN-construct or with an IF-THEN-ELSE-construct where one branch is state changing and the other one state

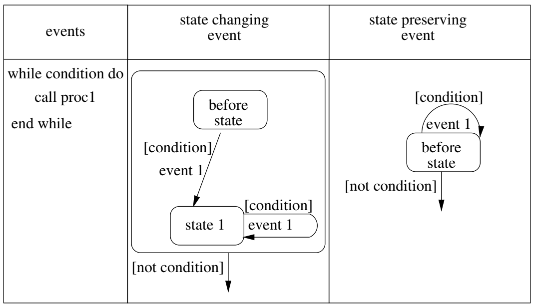

Figure 8. Iterative occurrence of events: WHILE

preserving. If, in an IF-THEN-ELSE only one branch contains the state variable, it might be considered from this perspective as a one-sided IF-THEN-construct. The mapping of these situations to OMT-diagrams is shown in Fig. 9 and Fig. 10.

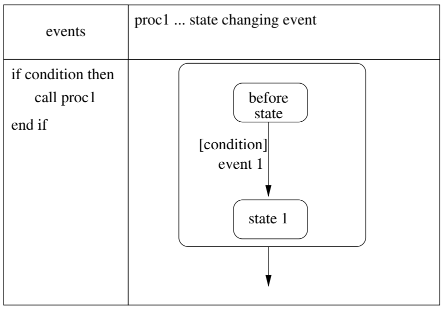

Figure 9. Complex Event resulting from IF-THEN

One might note, that in both cases the before-state of the IF and its final state are to be generalized to a higher level state. From this perspective, the situation is similar to the model of a WHILE-construct. With the WHILE, the semantics of this generalization is due to the loop invariant. Here, the generalization encompassing before state and final state seems to be rather ad-hoc.

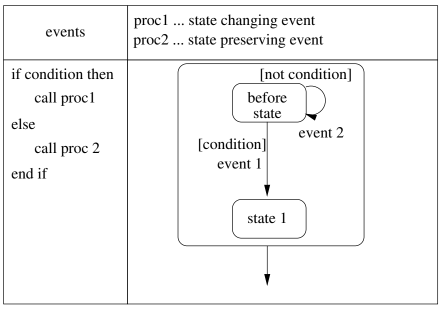

Figure 10. Complex Event resulting from different usage of State Variable

Fig. 11 shows how such complex events are modelled on higher level of abstractions. Assuming a sequence of calls to procedures proc1, proc-c, and proc3, where proc1 and proc3 are state changing events, while proc-c represents a complex event. As in this case, before- and final state of proc-c are generalized to the same state, the complex event event-c vanishes at the higher level of abstraction. It is hidden in this generalized state.

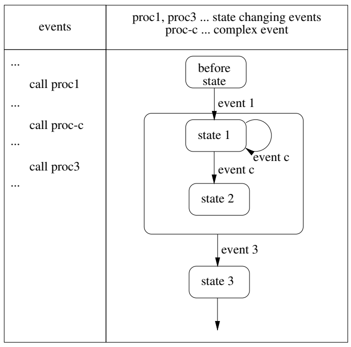

Figure 11. Sequence with Complex Event

3.1.4 Nesting

Here, two situations are discussed. One deals with calls, where the respective event occurs only at a lower level of abstraction, the other one deals with raising the level of abstraction of the generated OMT-model to raise it to the level of an analysis- or design document.

- *indirect occurrence of event:* This is identified, when one reaches a procedure call such that the called procedure itself does not use or define the state-variable. However, the called procedure (directly or indirectly) calls another procedure that performs such use or modification. For the OMT model, only those procedures are relevant, in which direct events occur. Procedures that might be called on the path to such procedures leading to direct events do not figure in the OMT diagram. They are relevant though in the process of code assignment to methods.

- *composite event:* This is an aggregation of a strongly connected subgraph of a dynamic model. It is not directly identified in the source code, but an abstraction needed to relate "low level events" identified in the source code to "high level events" expressed in the OOAM. Since the state diagrams defined above are all single entry-single exit structures, the subnet to be collapsed into a composite event can always be considered as lattice structure. The composit event thus leads from the before state in this lattice to its final state. One might note that, due to the before-final-generalizations of complex events, intermediate states/events can occur (e.g. with: IF cond THEN BEGIN proc1; proc2;

proc3 END;). These intermediate states are lined up on a path starting at the generalization and leading back to this generalization. Hence, it seems adequate to drop also this path in the model expressed on a higher level of abstraction.

3.2 Indirect State Changes

The state transition model as defined above is based on explicit state changes of the object in question. Certainly, the majority of state transitions will occur in the way expressed above, however several cases of indirect state transitions have to be considered and identified by means of a slightly more involved analysis of the source code.

An example should demonstrate the class of problems we are referring to: Consider a system that distinguishes between active and inactive business partners. To avoid redundancies and to optimize access time, active and inactive business partners are kept in different files. Whenever the state of a partner changes from active to inactive (or vice versa) its business record is just moved from one file to the other. In this case, which is representative for situations in which the categorization of a set is realized by physical partitioning, the event one can identify from the source code pertains to the respective partition (removal of an active partner and subsequent insertion of inactive partner). The "partner record" itself is not changed, as its state of activity is not represented explicitly.

This case cannot be found completely by the mechanisms described in section 3.1. However, a few conditions have to be met by the legacy application to ensure that it dealt with such partitions correctly. These are that removal and insertion need to be done in some form of transaction. Hence, some physical or logical closeness of the two operations can be assumed. Further, it needs to be ascertained, that the operation never leads to duplications. Hence, the removal operation has to be explicit. Whether this is done by some form of explicit dispose operation or by some index or pointer operation is immaterial. What matters, is that a pair of complementary operations in both containers need to exist and that the programmer will have followed some standardized pattern to perform such tricky operations without risk of faults. These patterns will be either recognized by a human [9], or could also be recognized by some pattern matcher, if such patterns are already known by the system. State preserving events will serve as starting points in the search of such a pattern matcher, as all of these indirect state changes we identified so far require at least one read operation of the state-variable.

The example just discussed is just one example where state information is not represented explicitly by means of a state-variable. It is hidden in the interaction between the dynamics of execution and some partial state representation. On first glance, one might consider such interactions as a limitation to our approch. At hindsight, one recognizes though, that only the consideration of the dynamic model will help to identify such implicitly represented state information. To identify it in reverse engineering ventures is important though, since the OOAM, which is developed on the basis of the application domain and in ignorance of particular implementation tricks used long times ago, will contain such transitions that cannot be uncovered from analyzing just declarations.

To show the breadth of the spectrum of this interaction between static and dynamic aspects we give another example. Here, naive observation will show a host of unmotivated state transitions, or at least state transitions at an unmotivated part of the program. To demonstrate this class of problems, we use again an example:

Assume somebody implements – based on the transaction frequency – a set by means of a multi-set, ensuring that upon deletion all equivalent instances (instances representing the same element) are removed and that upon counting duplicates are also removed. Thus, some form of "dynamic garbage collection" is performed on this set. Here, apparently not each insert operation (identified on first glance as state changing event) will actually be state changing from the OOAM perspective. Delete operations are state changing on both, the code level and the OOAM perspective. Count operations though, will obviously be state preserving on the OOAM perspective, but appear to be state changing from the implementation perspective.

Again, such effects will lead initially to wrong reversely generated dynamic models. The problems in these models can be identified though and hence an appropriate mapping to the OOAM can be obtained. One problem invariant common to all these cases is, that the programmer of the legacy code had to have (at least implicitly) some transaction model in mind. This transaction model will provide patterns and clues to uncover such situations. An inventory of such situations will improve the recovery process.

4. Bridging Levels of Abstraction

4.1. Hierarchies of Events

In toy-examples, one might aim for direct mappings between what is given on the OOAM level and what has been recovered as object model from legacy code. In realistic cases though, the OOAM and the object model directly recovered from code will be separated by several levels of abstraction, a problem alluded to already several times in the previous discussion. Here we give some hints on how one can deal with those problems.

Focussing on the different levels of abstraction between code and the OOAM one has also to consider that the procedural legacy code and the new OOAM are developed according to different development paradigms. This has the consequence, that the dynamic model one obtains directly from the code, is not necessarily a coherent and strictly connected graph, a prerequisite for a formally correct dynamic model. Different strings of event sequences need to be connected and states where such connections might be made have to be identified.

This problem seems analogous to problems dealt with in the database community in the realm of schema integration. For integrating object-oriented schemata, several approaches that extend concepts from relational schema integration exist. These approaches focus on the integration of the static part of the schemata involved. Surveying the literature dealing with the integraton of dynamic aspects of an object-oriented schema also leads only to fewer works. One of the most advanced among them is H. Frank's dissertation [6]. It can serve as a reference point for our work, but one has to note, that Frank's work considers data bases, while we consider software systems. Hence, when he considers aggregations and relationships between models on different levels of abstraction, state aggregations are obtained. This allows him to stay fully conformant with the modelling primitives provided by OMT. We, on the contrary, focussing on the "executional part" of a program, have to consider events as elements that can be expressed at different levels of abstraction and that therefore need to have the potential of being *composite events*, a concept not forseen in OMT and generally rejected in process oriented literature. In spite of these differences, the subtle event categorization and the formal model, defining the conditions under which events can be considered either equivalent or a generalization of some other event make Frank's work an important basis for our considerations.

4.2. Model Matching

To understand matching of the dynamic model obtained from the code level to the dynamic model contained in the OOAM, we have to consider different naming conventions and different levels of granularity. On the OOAM level, events are considered equivalent to method invocations. To relate these to events seen on the source code level, we decided to define only procedure invocations as events. This has the advantage that we are not concerned with the lowest level of granularity one can see in source code, the assignment statement. Further, we have a chance to label the events by some more or less speaking identifier. The linguistic problem involved with matching those are not further discussed in this paper. It is similar to the linguistic problem one has between state-variable and object designator, an issue dealt with in [7].

In contrast to the static similarity checks, model matching in the dynamic model can use much richer structural information, since two nontrivial graphs are to be matched against each other. Whether this matching will be driven by state (node) comparisons or by event (edge) comparisons is theoretically irrelevant. Practically, though, one has to consider that both, state and event information is plagued with some noise. Thus, even after having reached the adequate level of abstraction, chances for incomplete matches exist (of course the reasons for such imperfections have to be further investigated). Whether one uses the state space or the event sequences to perform the match is a tactical decision, depending on which seems to be more important for the application being reverse engineered. One might have the heuristic, that the aspect of primary concern is more elaborated on both levels. Hence the clues for an initial match are easier to be found. Once such clues have been found, one can progress by raising the level of abstraction of the reversely generated dynamic models by operations similar to those described in [6].

5. Outlook

So far, we described the usage of generating an object-oriented dynamic model from legacy source code only from the perspective of improving the selection of candidate objects. As such, creating such a model can be also seen as a general device to support program understanding. Notably, a person not versed in implementation tricks, common some decades ago, might see connections one would not identify with conventional control- or data flow analysis or slicing, as the dynamic model will establish relationships among portions of code not directly related by these techniques.

The aspect, that the dynamic model relates portions of code that is in no other close relationship in the non-object-oriented legacy system, can be used also at a later phase of our program renovation effort, during the construction of the object-oriented target system.

A full discussion of target system construction would be beyond the scope of this paper. We will refer here only to two basically different strategies:

1. *Recreation of the target system from legacy code:* This concept has been the leading concept behind *COREM* [12]. The advantage of this approach is a certain degree of confidence that "it is still the old code, just structured in a better way". If the legacy code is already at least nicely procedurally structured, one might hope that big chunks of code can be moved this way. Here, the dynamic model will be an important indicator, where exactly those portions can be found in the old code.

2. *Development of the target system with new technology:* This avenue is opened up, when a very strong object-oriented model can be recovered from the legacy sys-

tem. Such a model has to contain more than just static information. With static and dynamic information though, the model seems to be strong enough, to recognize them as design patterns [10, 16]. Once those have been fully identified, the new object-oriented system can at least partly be built from code contained in some pattern library. In doing so, several questions to ascertain semantic equivalence remain still to be answered.

6. Conclusion

An approach to extract an object-oriented dynamic model from classical legacy code has been presented. The merits of this approach lie in object recognition and in general program understanding per se. In addition to these aspects, that have been the key considerations leading to this work, the dynamic model will play also an important role in the generative steps of software renovation.

References

[1] Ilene Burnstein, Abdul Mirza, Katherine Roberson, Floyd Saner, and Abdallah Roberson. Knowledge engineering for automated program recognition and fault localization. In *Proceedings of the* 8^{th} *International Conference on Software Engineering and Knowledge Engineering SEKE' 96*, pages 85–91, Lake Tahoe (Nevada), June 1996.

[2] G. Canfora and A.Cimitile. An improved algorithm for identifying objects in code. *Software Practice and Experience*, 26(1):25–48, January 1996.

[3] G. Canfora, A. Cimitile, and G. A. Di Lucca. Recovering a conceptual data model from cobol code. In *Proceedings of the* 8^{th} *International Conference on Software Engineering and Knowledge Engineering SEKE' 96*, pages 442–449, Lake Tahoe (Nevada), June 1996.

[4] G. Canfora, Aniello Cimitile, M. Munro, and C.J. Taylor. Extracting abstract data types from c programs: A case study. In *Proceedings of the International Conference on Software Maintenance 1993*, pages 200–209. IEEE Computer Society Press, September 1993.

[5] Doris L. Carver. Reverse engineering procedural code for object recovery. In *Proceedings of the* 8^{th} *International Conference on Software Engineering and Knowledge Engineering SEKE' 96*, pages 442–449, Lake Tahoe (Nevada), June 1996.

[6] Heinz Frank. *View Integration für objektorientierte Datenbanken*. DISDBIS Bd. 32. infix, 1997.

[7] Harald Gall and Johannes Weidl. Object-model driven abstraction-to-code mapping. Technical Report TUV-1841-97-23, Technical University of Vienna, December 1997.

[8] Harald C. Gall, René R. Klösch, and Roland T. Mittermeier. Long-term information systems evolution via object-oriented re-architecturing. In 6^{th} *European Software Engineering Conference, ESEC '95*, 1995.

[9] Harald C. Gall, René R. Klösch, and Roland T. Mittermeier. Pattern-driven reverse engineering. In *Proceedings of the* 7^{th} *International Conference on Software Engineering and Knowledge Engineering SEKE' 95*, Rockville, Maryland, June 1995.

[10] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Pattern: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[11] I. Jacobson and F. Lindström. Re-engineering of old systems to an object-oriented architecture. In *OOPSLA*, pages 340–350, 1991.

[12] René R. Klösch and Harald C. Gall. *Objektorientiertes Reverse Engineering*. Springer–Verlag, Berlin, Heidelberg, 1995.

[13] S. Liu and N. Wilde. Identifying objects in a conventional procedural language: an example of data design recovery. In *Proceedings of the International Conference on Software Maintenance 1990*, pages 266–271. IEEE Computer Society Press, 1990.

[14] Roland T. Mittermeir, René R. Klösch, and Harald C. Gall. Using domain knowledge to improve reverse engineering. *International Journal of Software Engineering and Knowledge Engineering*, 6(33), 1996.

[15] Roland T. Mittermeir and Dominik Rauner-Reithmayer. Applying concepts of soft computing to software re(verse)-engineering. In *Proceedings of the Workshop on Migration Strategies for Legacy Systems, at ICSE'97*, Boston, USA, May 1997.

[16] Wolfgang Pree. *Design patterns for object-oriented software development*. ACM Press, 1994.

[17] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen. *Object-Oriented Modeling and Design*. Prentice Hall, Englewood Cliffs, New Jersey, 1991.

[18] Harry M. Sneed. Migration of procedural oriented cobol programs in an object-oriented architecture. In *Proceedings of the International Conference on Software Maintenance 1992*, pages 105–112, Orlando, Florida, November 1992.