# Recovering Behavioral Design Models from Execution Traces

Abdelwahab Hamou-Lhadj, Edna Braun, Daniel Amyot, and Timothy Lethbridge
*University of Ottawa*
*SITE, 800 King Edward Avenue*
*Ottawa, Ontario, K1N 6N5 Canada*
*{ahamou, ebraun, damyot, tcl}@site.uottawa.ca*

## Abstract

*Recovering behavioral design models from execution traces is not an easy task due to the sheer size of typical traces. In this paper, we describe a novel technique for achieving this. Our approach is based on filtering traces by distinguishing the utility components from the ones that implement high-level concepts. In the paper, we first define the concept of utilities; then we present an algorithm based on fan-in analysis that can be used for the detection of utilities. To represent the high-level behavioral models, we explore the Use Case Map (UCM) notation, which is a language used to describe and understand emergent behavior of complex and dynamic systems. Finally, we test the validity of our approach on an object-oriented system called TConfig.*

## 1. Introduction

Dynamic analysis consists of understanding the behavior of a software system by analyzing the data generated from executing its features.

Lately, there have been an increasing number of tools for analyzing traces generated from object-oriented (OO) systems [4, 5, 9, 12, 14, 18, 22]. This growing interest has been driven by the fact that OO concepts such as polymorphism and dynamic binding complicate the process of merely applying static analysis of the source code to understand such systems [19].

To deal with the sheer size of typical traces, most existing tools rely on interactive features such as enabling the hiding of specific components, facilitating search of the traces, detecting patterns of execution, etc. The problem is that it is totally up to the analyst to combine all these features in order to manipulate the trace and reach the desired level of abstraction. This is usually hard to accomplish.

In this paper, we propose an approach for recovering behavioral design models from execution traces based on the removal of utility components. For this purpose, we describe an algorithm based on fan-in analysis for the detection of utilities.

To represent the resulting behavioral design models, we selected the Use Case Map (UCM) notation instead of the UML sequence diagram that has been extensively applied in this context. UCMs are part of the ITU-T family of languages for describing functional requirements and high-level designs [1, 6]. UCMs focus on causal sequences of responsibilities and abstract from message exchanges. The motivation behind using UCMs is further presented in Section 2.3.

To validate our approach, we analyzed an execution trace of an OO system called TConfig [20, 21]. The results were validated by the main designer of TConfig.

The rest of this paper is organized as follows: the next section describes our approach, which includes a brief discussion of the traces used in this paper, the detection of utilities based on fan-in analysis, and a mapping between trace components and UCM elements. In Section 3, we present the results of analyzing the execution trace of TConfig.

## 2. Approach

Figure 1 illustrates the approach described it this paper for recovering UCMs from execution traces. The main steps are:

1. We generate the execution traces that correspond to the software features under study.
2. We filter the traces by removing low-level implementation details such as utilities. In section 2.2, we discuss how we detect utilities.
3. We extract UCMs from the resulting traces. In section 2.3, we discuss how trace elements can be mapped to UCM elements.
4. We validate the results with the original developers of the systems.

The validation step might lead to further filtering of the trace if the software engineer judges that the trace still contains too much detail.

## 2.1. Traces of Method Calls

To reproduce the execution of an object-oriented system, one needs to collect at least the events related to object construction and destruction, plus method entry and exit [3]. Traces, once generated, are usually saved in text files. A trace file contains a sequence of lines in which each line represents an event. An example of this representation is given by Richner and Ducasse in [12]. Each line records: The class of the sender, the identity of the sender, the class of the receiver, the identity of the receiver, and the method invoked.

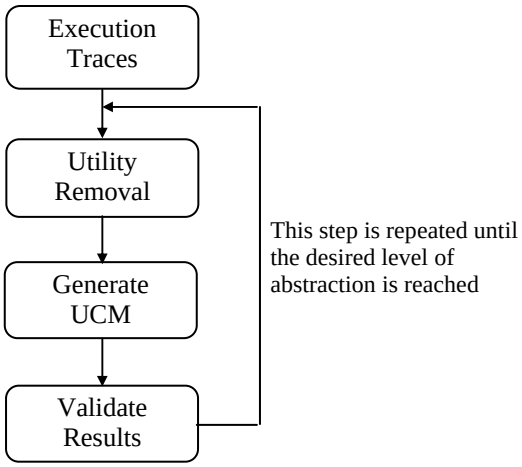


**Figure 1. Approach for recovering UCMs**

However, one of the main characteristics of the UCM notation consists of showing guard conditions. Therefore, in addition to method entry and exit, we also keep track of the conditions that are executed.

Figure 2 shows an example of a sample trace where specific objects are substituted by their class type – the term trace of class interactions would be more appropriate in this case. The figure shows an object of the class *Screen* that calls its *init* method, which in turn creates an instance of the class *Shape* and calls its *update* method. The *update* method calls the *draw* and *refresh* methods if the *fitsScreen* condition is evaluated to true.

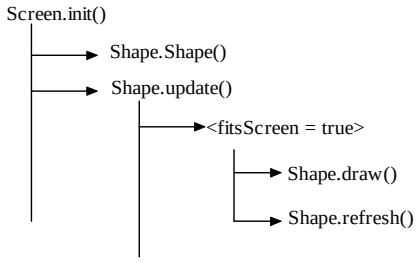


**Figure 2. Trace of method calls with guard conditions**

## 2.2. Detecting Utilities

In our previous work, we studied the concept of utility components and how they differ from the other system components [7]. This study was conducted at QNX Software Systems (the company that supports our research) and involved more than twenty software engineers. Based on the results of this study, we define a utility as: *Any element of a program designed for the convenience of the designer and implementer and intended to be accessed from multiple places within a certain scope of the program*.

Many utilities will be designed to be reused in multiple programs; this definition does not preclude that, but does not require it. Also the definition allows a utility to be a method, class, package or some other element, and to be accessed from a scope that could be as narrow as a class or as wide as the entire system. A key to the definition is that a utility will be accessed from an unknown number of places, not just one. The definition allows such things as accessing methods to be considered utilities, and does not require utilities to be grouped in any way, although it does not preclude that.

In order to detect utility components, we have developed an algorithm that is based on fan-in analysis. Although most of the concepts presented here can easily apply to detecting various types of utilities (e.g., utility methods, packages, etc.), the rest of this subsection focuses on detecting utility *classes* for simplicity reasons.

The fan-in analysis technique is based on the exploration of the class dependency graph built from static analysis of the system. It is used to extract the classes that have a large number of incoming edges (i.e., many dependents). Computing fan-in is a typical way for achieving this as it has already been shown in areas such as software clustering [11, 16]. However, there is a need to adjust this metric in order to consider the scope of a utility. Obviously, using the same threshold for detecting system-scope utilities as well as utilities that belong to specific subsystems would be ineffective.

The class dependency graph is a directed graph where the nodes are the system's classes and the edges represent a dependency relationship among the classes as shown in Figure 3. Building a complete class dependency graph may require parsing the source code (or bytecode files). There are several types of static dependencies that may exist between two given classes including method calls, generalization, realization, etc. Additionally, the edges might be weighted to represent the number of dependencies that exist between two given classes.

To measure the extent to which a particular class can be considered a utility, we suggest the following utilityhood metric:

Given a class C and the following sets:

- S = Set of classes considered in the analysis
- IN = A subset of S that consists of the classes that depend on C (fan-in).

We define the utilityhood metric, U, of the class C as:

$$U = |IN| / (|S|-1)$$

S is used to represent the scope considered in the computation of U. For example, if we are looking for system-level utilities, then S will contain all the system's classes. However, if the search for utilities is restricted to a particular package, then S can be designated to contain the classes of this particular package only.

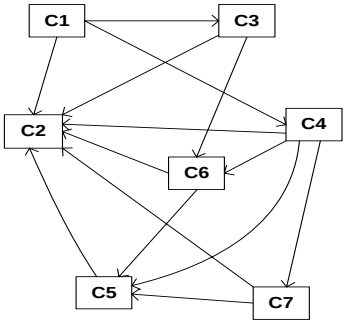


**Figure 3. Class dependency graph**

U ranges from 0 to 1. 0 indicates that the class has no incoming edges. If the scope is the entire system then the class that scores 0 must contain the entry point of the system (or it is an unreachable class). If U is equal to 1 then the class is called by all other classes of S which is a strong indicator that it is a utility class. Note that self dependencies are not considered which explains |S| - 1.

Given the utilityhood metric and the class dependency graph, the algorithm for detecting utility classes is rather straightforward. The steps of the algorithm can be summarized as follows:

1. Set utility_set (the set of utility classes) to empty
2. For every class C of the set S: Compute U
3. Identify the classes that have a U value greater than or equal to a threshold D (that we will discuss later) and add them to utility_set.
4. The classes that are in utility_set are the candidate utility classes.

Step 1 simply creates an empty set that will contain the candidate utility classes. Step 2 computes the utilityhood metric (U) for every class of the set S. Step 3 identifies the classes with a U value greater than or equal to a certain threshold D. Suitable values of D need to be determined by conducting experiments with different systems. We anticipate that each system might have its own threshold, and that software engineers exploring systems will dynamically change D in order to vary the amount of detail displayed. The final step (Step 4) of the algorithm outputs the resulting utility classes.

Table 1 shows the result of computing |IN| and U for every class of the class dependency graph of Figure 3. S contains all the classes that appear in the graph. We used the standard deviation to easily spot the classes with a U value that deviates significantly from the other values. To standardize the results, the z-score was used [15]. The classes that have a large and positive Z value are possible utilities. Note that the components that have a negative Z value are the ones that have a very low number of incoming edges, which discounts them from being candidate utilities.

For example, Table 1 clearly shows that the class C2 has a positive Z value: its U value deviates with 2 standard deviations from the mean, since it has a much larger number of incoming edges (fan-in) compared to other classes. This strongly suggests it is a utility class. However, the class C1 deviates with 1 standard deviation from the mean on the negative side, which strongly suggests it is not a utility. In this case C1 has 0 incoming edges.

**Table 1. Example of applying fan-in analysis**

| | **\|IN\|** | **U** | **Z** |
|---|---|---|---|
| C1 | 0 | 0.00 | -1.00 |
| **C2** | **6** | **1.00** | **2.00** |
| C3 | 1 | 0.17 | -0.50 |
| C4 | 1 | 0.17 | -0.50 |
| C5 | 3 | 0.50 | 0.50 |
| C6 | 2 | 0.33 | 0.00 |
| C7 | 1 | 0.17 | -0.50 |
| | **MEAN** | **0.33** | **0** |
| | **STDEV** | **0.33** | **1** |

However, U (and its corresponding Z value) is not the only parameter that needs to be considered for efficient detection of all utilities. It is important to consider how a redefinition of the set S can be used to detect utilities that exist in scopes narrower that the entire system. For example, suppose that the classes C4, C5, C6 and C7 of Figure 3 belong to the same package P and that we want to detect possible utility classes that exist within P.

Table 2 shows the result of computing |IN| given that the redefined S contains the classes of the package P only. The class C5 has a large fan-in compared to the other classes of the package P. This is also indicated by its Z-score.

It is clear that we need to conduct empirical studies to determine an appropriate threshold that will clearly

distinguish utilities from the other system's components. However, even if such a threshold exists we will still need to allow enough flexibility so as the analyst can adjust the amount of information contained in the traces according to his or her needs. This is because, what might be a utility for one maintenance task might not be for another task.

One shortcoming of this approach is when it is applied to systems that have a poor design. The problem is that the scope of the components might be hard to determine since it may not be clearly reflected in the source code. For example, some classes might be placed in the wrong packages. For this purpose, there is a need to investigate techniques that are independent from the scope attribute. We leave this point as future work.

**Table 2. Fan-in analysis applied to a specific package**

| | **\|IN\|** | **U** | **Z** |
|---|---|---|---|
| C4 | 0 | 0.00 | -0.99 |
| **C5** | **3** | **1.00** | **1.39** |
| C6 | 1 | 0.33 | -0.20 |
| C7 | 1 | 0.33 | -0.20 |
| | **MEAN** | **0.42** | **0** |
| | **STDEV** | **0.42** | **1** |

## 2.3. Representing Traces Using UCMs

Use Case Maps [6] allow one to model system behavior in terms of causal flows of *responsibilities*, which are activities that can be allocated to system components.

We chose to use UCMs since they are a rich requirements-level notation for showing at a glance the various control-flow possibilities in a system. Unlike UML 1.x sequence diagrams, UCMs abstract from inter-component communication to focus on the business logic. Like activity diagrams, they can integrate many scenarios with operators for looping and for forking and joining alternative or concurrent paths. Complex maps can also be decomposed into sub-maps (with *stubs*).

UCMs can also represent the system architecture in a 2-dimensional way, with components containing sub-components (in a way more understandable than what can be achieved with UML 2.0 swimlanes). Moreover, the UCM notation has special operators for describing timers and for creating and manipulating objects (*dynamic responsibilities*). Additionally, stubs may also contain multiple sub-maps. This allows for flexible integration and exploration of scenarios that have overlapping parts.

A first attempt at using UCMs for program understanding is presented in [2]. The authors based their approach on the static generation of scenarios from manually tagged elements in the code, which is more cumbersome and less prone to automation than the dynamic approach suggested here.

Table 3 shows how we map traces to the various UCM concepts. The case study described in the next section will provide illustrations of typical UCMs for a single trace.

**Table 3. Mapping from traces to UCMs**

| **Trace element** | **UCM element** |
|---|---|
| Package | Component (Agent), shown as a rectangle with thick border. |
| Class | Component (Team), shown as a rectangle with narrow border. |
| Object | Component (Object), shown as a rounded-corner rectangle. |
| Thread | Component (Process), shown as a parallelogram. |
| Beginning / End of trace | Start point (circle) / End point (bar) (also used as connectors for linking sub-scenarios to the parent stub) |
| Instruction | Responsibility (shown as a X on a path) |
| Block of 3 or more instructions in the same class/object | Stub (diamond) with the name of the first instruction that is not a constructor. This stub contains a *plug-in* (another sub-map) showing the sub-sequence with one responsibility per instruction. |
| Constructor | Dynamic responsibility (arrow with +) |
| Destructor | Dynamic responsibility (arrow with -) |
| Repeated instruction | Responsibility with repetition count (number between curly brackets) |
| Repeated sequence | Loop (with loop count between curly brackets) |
| Condition | Condition (between square brackets) |
| Non-continuous repetition | Plug-in map corresponding to the repeated sequence. A stub (with repetition count if necessary) using this plug-in is inserted in the path each time this repetition occurs. |

# 3. Case Study

We analyzed an execution trace generated from an object-oriented system called TConfig (ver. 2.1) [20, 21]. TConfig is a Java application used to generate the minimum number of test configurations covering component interactions of degree *n*, where *n* is defined by the user. It uses advanced mathematical concepts such as fields and Latin squares. TConfig contains 4 packages, 29

classes, and 407 methods. The size of TConfig is 6.56 KLOC.

### 3.1. Collecting the Traces

We used our own instrumentation tool based on the BIT framework [10] to insert probes at the entry and exit points of each system's non-private methods and branch of every condition statement. Constructors are treated in the same way as regular methods. Although the system comes with a GUI, we deliberately ignored the GUI package to avoid encumbering the traces.

TConfig's GUI supports various parameters which allow to choose the number of components (and their names), the number of values for each component (and their names), the coverage degree (*n*), and the heuristic to be used for generating test configurations (recursive block or IPO). We decided to analyze the most feature-rich set of options: named collection of three components with two values for each component, pairwise interactions ($n = 2$), and the recursive block heuristic. We refer to this feature as the *named-recursive feature* and we refer to the trace that corresponds to it as the *named-recursive trace*.

The trace was generated as the system was running, and was saved in a text file containing the following information:

- Thread name
- Full class name (e.g., base.Value)
- Method name
- Condition if it is a condition statement
- A nesting level that maintains the order of calls

We noticed that the tool uses only one thread, which made us ignore the thread information.

The initial statistics about the collected trace are shown in table 4. The metrics used in the table are described in what follows:

- N = initial size of the trace
- $N_{acc}$ = size of the trace after removing accessing methods. For this purpose, we used the set and get naming convention to detect accessing methods
- $R_{acc} = 1 - N_{acc} /N$

The initial size, N, of the named-recursive trace is 1029 which includes the method invocations as well as the conditions that were executed. Obviously, this does not reflect the size of typical and most interesting traces, which can easily go beyond hundreds of thousands of invocations. We deliberately choose a small system for this preliminary study to confirm the idea that filtering based on the detection of utilities is a promising approach for future trace analysis techniques. Future work should focus on analyzing large traces and investigating more sophisticated utility detection techniques.

The removal of accessing methods results in a trace that contains 203 invocations and conditions as represented by $N_{acc}$. This represents a reduction ratio of 80%.

**Table 4. Statistics of the named-recursive trace**

| Trace | N | $N_{acc}$ | $R_{acc}$ |
|---|---|---|---|
| named-recursive | 1029 | 203 | 80% |

### 3.2. Processing the trace

We used fan-in analysis to detect potential utilities that can be removed from the named-recursive trace to recover the corresponding Use Case Map. For this purpose, we proceeded according to two phases. The first phase consists of detecting system-scope utilities that we simply refer to as *global utilities*. The second phase aims to improve the result of the first phase by removing utilities that may exist in specific packages. We refer to these utilities as *local utilities*.

The result of each phase is presented to the main designer of TConfig in the form of a UCM. The objective is to evaluate whether the resulting UCM conveys the main behavioral aspects of the traced software feature or not. In other words, we want to know if the UCM had enough information that will allow a designer to understand the feature at a high level without diving into the details. The expert's comments are discussed in the Section 3.4.

**First phase: Detecting global utilities**

Table 5 shows the results of applying fan-in analysis to TConfig classes in order to detect system-scope utilities (i.e., S is set to contain all the system's classes). The table is sorted according to the descending order of Z.

Due to the absence of a utility threshold, we are left with the only option of removing iteratively the classes that have a large Z positive value. The trace resulting after each iteration will need to be turned into a UCM and presented to the expert for validation. The problem with this approach is that we might end up having successive iterations that result in traces that do not differ a lot from each other. Validating each of these traces will certainly be inefficient. To overcome this problem, we use the gain in terms of size reached after the removal of a particular class (or set of classes if they have identical Z value) to help us decide when we need to produce the UCM. In other words, if the removal of one class does not result in an important reduction of the trace size then we proceed with the removal of additional classes before producing the corresponding UCM. This will prevent us from

having several UCMs that are only slightly different from each other.

We applied this technique to the classes of TConfig system. The first iteration is concerned with the removal of the 'base.ParameterSet' class since it is the one that has the highest Z value (3.04). However, the reduction ratio attained after removing this class is $R_{ps}$ = 7% (the raw number of invocations and conditions is $N_{ps}$ = 188) as shown in Table 6.

**Table 5. Applying fan-in analysis to TConfig system**

| Class | \|IN\| | U | Z |
|---|---|---|---|
| **base.ParameterSet** | **8** | **0.29** | **3.04** |
| **base.ConfigurationSet** | **5** | **0.18** | **1.40** |
| **gui.ItemListDialog** | **5** | **0.18** | **1.40** |
| **recursive.Messages** | **5** | **0.18** | **1.40** |
| recursive.PolyMod | 4 | 0.14 | 0.85 |
| recursive.Utility | 4 | 0.14 | 0.85 |
| base.Generator | 3 | 0.11 | 0.30 |
| base.Messages | 3 | 0.11 | 0.30 |
| base.Parameter | 3 | 0.11 | 0.30 |
| base.Value | 3 | 0.11 | 0.30 |
| gui.TConfigUI | 3 | 0.11 | 0.30 |
| recursive.IntMod | 3 | 0.11 | 0.30 |
| recursive.RemainderPoly | 3 | 0.11 | 0.30 |
| base.Document | 2 | 0.07 | -0.25 |
| gui.ParmSetDialog | 2 | 0.07 | -0.25 |
| gui.ValuesDifferDialog | 2 | 0.07 | -0.25 |
| gui.ValuesSameDialog | 2 | 0.07 | -0.25 |
| ipo.InteractionElement | 2 | 0.07 | -0.25 |
| ipo.TestValue | 2 | 0.07 | -0.25 |
| recursive.Field | 2 | 0.07 | -0.25 |
| gui.ParameterDialog | 1 | 0.04 | -0.79 |
| ipo.IPOGenerator | 1 | 0.04 | -0.79 |
| recursive.LatinSquares | 1 | 0.04 | -0.79 |
| recursive.OrthogonalArray | 1 | 0.04 | -0.79 |
| recursive.RecursiveGenerator | 1 | 0.04 | -0.79 |
| gui.TConfig | 0 | 0.00 | -1.34 |
| gui.TConfigApplet | 0 | 0.00 | -1.34 |
| ipo.IESet | 0 | 0.00 | -1.34 |
| recursive.GFPolynomial | 0 | 0.00 | -1.34 |
| | **MEAN** | **0.09** | **0** |
| | **STDEV** | **0.07** | **1** |

**Table 6. Results after removing system-scope utilities**

| Trace | $N_{acc}$ | $N_{ps}$ | $R_{ps}$ | $N_{g\text{-}util}$ | $R_{g\text{-}util}$ |
|---|---|---|---|---|---|
| named-recursive | 203 | 188 | 7% | 99 | 51% |

We did not attempt to represent the corresponding UCM simply because it will look similar to the original trace. On the other hand, the removal of the 'base.ParameterSet' in addition to the classes that have the second highest Z score namely, 'base.ParameterSet', 'base.ConfigurationSet', 'gui.ItemListDialog', and 'recursive.Messages' results in a reduction ratio of $R_{g\text{-}util}$ = 51% (the number of invocations and conditions $N_{g\text{-}util}$ = 99) as shown in Table 6. This high reduction ratio is a strong indicator that many low-level details have been removed from the original trace. Therefore, we have decided to transform the resulting trace into a UCM that is shown in Figure 4.

**Second phase: Detecting local utilities**

The named-recursive trace invokes methods of two packages which are 'base' and 'recursive'. We applied fan-in analysis for each of these packages.

Table 7 shows the result of applying fan-in analysis to the 'recursive' package. The class 'recursive.Messages' was already removed during the detection of global utilities. Similar to the detection of global utilities, we have decided to consider the classes 'recursive.PolyMod' and 'recursive.Utility' as candidate utilities. However, we notice from the result of removing global utilities as illustrated in Figure 4 that the classes 'recursive.IntMod' and 'recursive.RemainderPoly' are behind many interactions. We therefore assume that these are low-level interactions and decide to add the two classes to the utility set as well. This assumption will need to be verified by the expert when validating the resulting UCM.

**Table 7. Applying fan-in analysis to the 'recursive' package**

| Class | \|IN\| | U | Z |
|---|---|---|---|
| **recursive.Messages** | **5** | **0.56** | **1.53** |
| **recursive.PolyMod** | **4** | **0.44** | **0.96** |
| **recursive.Utility** | **4** | **0.44** | **0.96** |
| **recursive.IntMod** | **3** | **0.33** | **0.40** |
| **recursive.RemainderPoly** | **3** | **0.33** | **0.40** |
| recursive.Field | 2 | 0.22 | -0.17 |
| recursive.LatinSquares | 1 | 0.11 | -0.74 |
| recursive.OrthogonalArray | 1 | 0.11 | -0.74 |
| recursive.GFPolynomial | 0 | 0.00 | -1.30 |
| recursive.RecursiveGenerator | 0 | 0.00 | -1.30 |
| | **MEAN** | **0.26** | **0** |
| | **STDEV** | **0.20** | **1** |

Table 8 shows the result of applying fan-in analysis to the 'base' package. The class 'base.ConfiugrationSet' was already removed during the detection of global utilities. We next decide to remove the 'base.Messages' class since it is the one that has a high Z value. We also decide to remove the 'base.Value' class in an attempt to better abstract out the interaction between the Parameter class and the Value class.

**Table 8. Applying fan-in analysis to the 'base' package**

| Class | \|IN\| | U | Z |
|---|---|---|---|
| **base.Messages** | **3** | **0.50** | **1.61** |
| **base.ConfigurationSet** | **2** | **0.33** | **0.59** |
| **base.Value** | **2** | **0.33** | **0.59** |
| base.Generator | 1 | 0.17 | -0.44 |
| base.Parameter | 1 | 0.17 | -0.44 |
| base.ParameterSet | 1 | 0.17 | -0.44 |
| base.Document | 0 | 0.00 | -1.46 |
| | **MEAN** | **0.24** | **0** |
| | **STDEV** | **0.16** | **1** |

To summarize, the detection of local utilities has resulted in the removal of six additional classes, which are: 'base.Messages', 'base.Value', 'recursive.PolyMod', 'recursive.IntMod', 'recursive.RemainderPoly', and 'recursive.Utility'. The removal of local utilities from the trace (resulting after the removal of global utilities) results in a trace whose size is $N_{l\text{-}util} = 66$, which represents a reduction ratio of $R_{l\text{-}util} = 38\%$. The resulting UCM is shown in Figure 5.

### 3.3. Generating UCMs

We used a tool called UCMNav [17] to draw and export the UCMs that were recovered from the named-recursive trace (Figures 4 and 5). We carefully followed the mapping rules in Table 3 to reproduce the UCMs that correspond correctly to the content of the traces. Arrows were added to some lengthy path segments to clarify their direction. Additional plug-in UCMs were generated for each stub found in Figures 4 and 5. They include sequences of responsibilities capturing blocks of instructions in the same class. These more detailed views are not shown here for simplicity.

Although UCMNav allows great flexibility for manipulating the various UCMs elements, we hope to be able to generate the UCMs automatically from the traces in future studies. A standard exchange format for representing traces of method calls such as CTF [8] could help here.

### 3.4. Validating the results

We showed the UCM that resulted from the removal of global utilities (Figure 4) to the expert (who was familiar with this notation) to assess whether it was a good high-level representation of the traced scenario or not. In general, the expert was able to trace the main responsibilities of the different components (i.e., classes and packages) with respect to the traced scenario. He also said that the content of the UCM was a good high-level description of the scenario, and he was glad to see the loops used for the creation of parameter values. He appreciated the layout, especially as he could clearly distinguish the input of parameters and the processing with a specific heuristic (recursive generation in this trace). The expert, who is also familiar with UML 1 and UML 2 sequence diagrams and with Message Sequence Charts, mentioned that he did not miss the inter-component interactions (messages) as he was more interested in the flow of activities for understanding (and explaining) this application. The details of the stubs found in the plug-in maps were of limited interest compared to the top-level UCMs found in Figures 4 and 5.

At a certain point the expert thought that the responsibilities confined to the 'OrthogonalArray' class were too abstract and that we might have removed important information. However, after examining the source code, he confirmed that the UCM showed what was actually happening in the code which was different from what he initially thought. We found this last point particularly interesting because it suggests that we can use dynamic analysis to improve existing program comprehension models that are based on static analysis only [13].

However, the expert disagreed with the fact the 'base.ConfigurationSet' class should be completely removed from the UCM. According to him, while most methods of this class can be considered as utilities, he wished that the UCM included two methods of this class: mergeConfigurationsWith and mergeParametersWith. We attribute the absence of these methods to the fact that the analysis is done at the class level rather than the method level. Therefore, a class that contains many utility methods is automatically considered as a utility class although it may contain some methods that are important to the understanding of the scenario. Our approach could easily be adapted to work at the method level, or a combination of class and method level. We leave this for future research.

Finally, the expert pointed out that the classes 'IntPoly', 'RemainderPoly', and 'PolyMod' are his implementation of some kind of a mathematical field, which is used to compute Latin squares.

The expert's feedback with respect to the UCM generated from removing global as well as local utilities (Figure 5) indicates that the model is now much clearer: It represents a better description of the high-level model of the scenario. The expert agreed particularly with the removal of the 'base.Value' class. He said that this class can be inferred from the 'Parameter' class since all it does is create values that are added to the parameters.

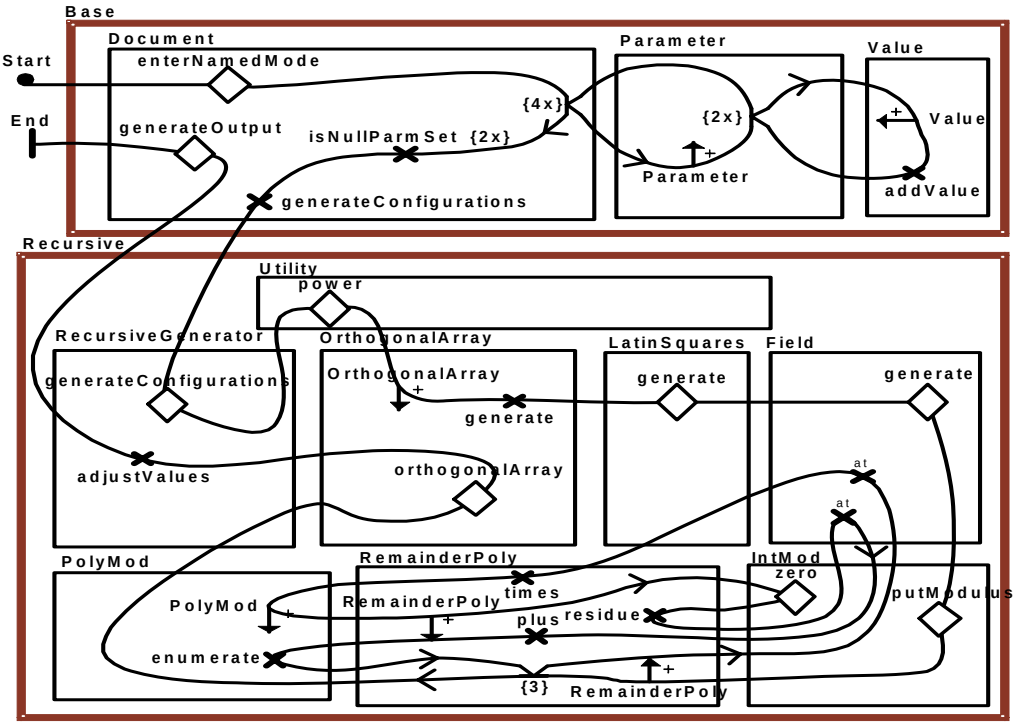


**Figure 4. UCM corresponding to the trace after removing system-scope utilities**

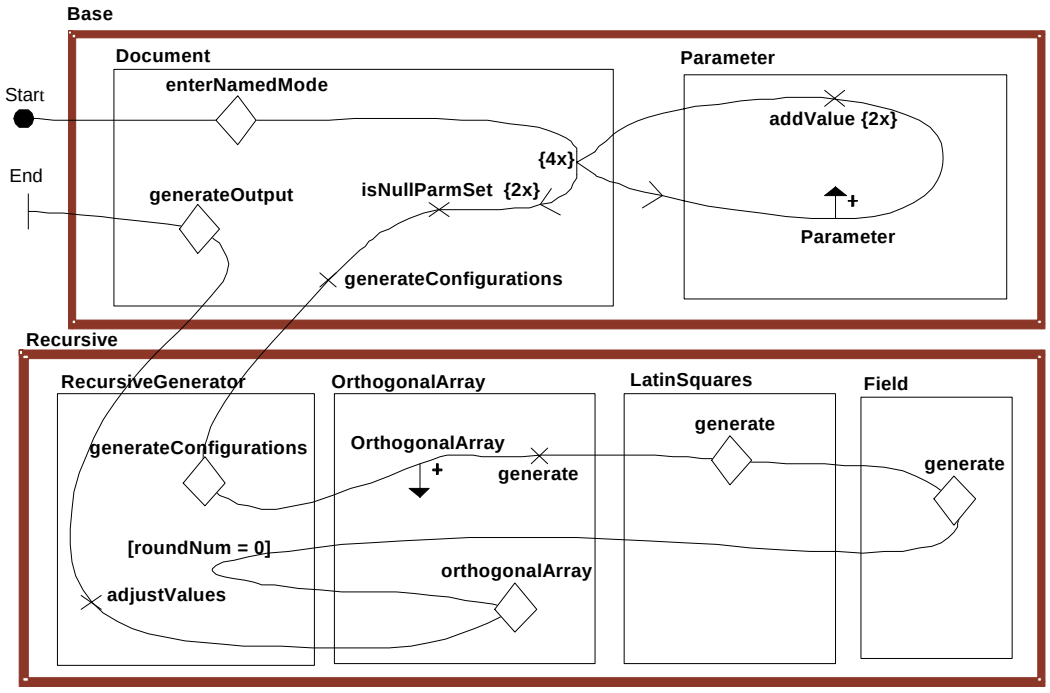


**Figure 5. UCM corresponding to the trace after removing system-scope as well as local utilities**

The expert found that the removal of the 'recursive.Utility' class was definitely a good decision since it only contains a set of static methods that are used by the other classes of the 'recursive' package.

The removal of the classes 'IntMod', 'PolyMod', and 'RemainderPoly' was seen by the expert as a way to hide the details of the implementation of the fields that are used to compute the Latin squares. He was especially glad to see the zigzagging path segments involving these three classes (bottom of Figure 4) disappear as they were not contributing that much to the high-level logic of the application.

## 4. Conclusions and Future Work

In this paper, we discuss how behavioral design models can be build from execution traces using filtering techniques based on the detection of utilities. For this purpose, we presented a precise definition of the concept of utilities and discussed an algorithm that detects them. Our algorithm is based on fan-in analysis and takes two parameters: the component dependency graph and the set of components that are included in the analysis. The latter is used to adjust fan-in metric to detect utilities that exist in a scope narrower than the entire system.

Additionally, we chose to represent the high-level behavioral models using the UCM notation as opposed to the UML sequence diagram that are usually used in this context. UCMs provide a compact and hierarchical view of the main sequences of responsibilities combined with architectural components. They abstract from details related to message exchanges while providing means of visualizing dynamic aspects (such as the creation and destruction of objects) in a static way.

The analysis of a small-sized trace generated from the TConfig system has resulted in two UCMs. The first UCM is generated after the removal of system-scope utilities whereas the second UCM considers the removal of system-scope utilities as well as utilities that belong to specific packages. The two UCMs were presented to the designer of TConfig to validate their content. The designer found the two UCMs very descriptive of the traced scenario although the second UCM represents a much clearer picture of the scenario.

Future work should focus on experimenting with the concept of filtering based on utilities on large traces. We anticipate that there is a need to more advanced utility detection techniques. We also need to adapt our approach to the detection of other types of utilities such as utility methods and packages.

On the visualization side, the automated generation of UCMs from traces represents interesting challenges from a layout point of view. We also need to explore different ways of structuring traces into different levels of parent maps and sub-maps based on different (user-driven) criteria such as the nesting of invocations. A UCM can also combine several scenarios in a single view (with alternative paths and dynamic stubs). This feature could be used to explore more complete design views where similarities and variations in a set of scenarios generated from traces would be emphasized.

Finally, we need to investigate how the utility detection capabilities can be incorporated into a trace analysis tool.

## Acknowledgments

This work was supported financially by NSERC and QNX Software Systems. We are most thankful to Alan Williams for taking the time to review and evaluate the traces and UCMs generated from TConfig.

## References

[1] D. Amyot, "Introduction to the User Requirements Notation: Learning by Example". *Computer Networks*, 42(3), 285-301, 21 June 2003.

[2] D. Amyot, G. Mussbacher, and N. Mansurov, "Understanding Existing Software with Use Case Map Scenarios". In *3rd SDL and MSC Workshop (SAM'02)*, Aberystwyth, U.K., June 2002. LNCS 2599, pp. 124-140.

[3] W. De Pauw, R. Helm, D. Kimelman, J. Vlissides, "Visualizing the Behaviour of Object-Oriented Systems". In *Proc. of the 8th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Washington, DC, 1993, pp. 326-337.

[4] W. De Pauw, E. Jensen, N. Mitchell, G. Sevitsky, J. Vlissides, J. Yang, "Visualizing the Execution of Java Programs". In *Proc. International Seminar on Software Visualization*, Dagstuhl Castle, Wadern, 2002, pp. 151-162.

[5] W. De Pauw, D. Lorenz, J. Vlissides, M. Wegman, "Execution Patterns in Object-Oriented Visualization". In *Proc. of the 4th USENIX Conference on Object-Oriented Technologies and Systems, COOTS*, 1998, pp. 219-234.

[6] ITU-T, *Draft Recommendation Z.152 (Use Case Maps)*, Sept. 2003. http://www.UseCaseMaps.org/urn

[7] A. Hamou-Lhadj, T. C. Lethbridge**,** "Reasoning about the Concept of Utilities". In *Proc. of the 1st ECOOP Workshop on Practical Problems of Programming in the Large*, Oslo, Norway, June 2004

[8] A. Hamou-Lhadj, T. C. Lethbridge**,** "A Metamodel for Dynamic Information Generated from Object-Oriented Systems". In *Proc. of the 1st International Workshop on Meta-models and Schemas for Reverse Engineering (ATEM)*, Victoria, Canada, published by *Electronic Notes in Theoretical Computer Science (ENTCS)*,94: 59-69, 2004.

[9] D. Jerding, S. Rugaber, "Using Visualization for Architecture Localization and Extraction". In *Proc. 4th Working Conference on Reverse Engineering*, Amsterdam, Netherlands, Oct. 1997, pp. 56-65.

[10] H. B. Lee, B. G. Zorn, "BIT: A tool for Instrumenting Java Bytecodes". *USENIX Symposium on Internet Technologies and Systems*, Monterey, California, 1997, pp. 73-82.

[11] H. A. Müller, M. A. Orgun, S. Tilley, J. Uhl, "A Reverse Engineering Approach to Subsystem Structure Identification". *Journal of Software Maintenance: Research and Practice, Vol 5, No 4*, December 1993, pp. 181-204.

[12] T. Richner, S. Ducasse, "Using Dynamic Information for the Iterative Recovery of Collaborations and Roles". In *Proc. of the 18th International Conference on Software Maintenance (ICSM)*, Montréal, Canada, 2002, pp. 34-43.

[13] M.-A.D. Storey, K. Wong, H. A. Müller, "How Do Program Understanding Tools Affect How Programmers Understand Programs?". In *Proc. 4th Working Conference on Reverse Engineering*, Amsterdam, Holland, 1997, pp. 12-21.

[14] T. Systä, "Understanding the Behavior of Java Programs". In *Proc. 7th Working Conference on Reverse Engineering*, Australia, Brisbane, 2000, pp. 214-223.

[15] M. Triola, *Elementary Statistics*, 9th edition, Addison-Wesley, 2003.

[16] V. Tzerpos, R. C. Holt, "ACDC: An Algorithm for Comprehension-Driven Clustering". In *Proc. of the 7th Working Conference on Reverse Engineering*, Brisbane, Australia, November 2000, pp. 258-267.

[17] UCMNav: http://www.usecasemaps.org/tools/ucmnav

[18] R. J. Walker, G. C. Murphy, B. Freeman-Benson, D. Swanson, and J. Isaak, "Visualizing Dynamic Software System Information through High-level Models". In *Proc. of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, British Columbia, Canada, October 1998, pp. 271-283.

[19] N. Wilde, and R. Huitt, "Maintenance Support for Object-Oriented Programs". *Transactions on Software Engineering*, 18(12):1038–1044, Dec. 1992.

[20] A. W. Williams, "Software Component Interaction Testing: Coverage Measurement and Generation of Configurations". *Ph.D. thesis, University of Ottawa*, 2002, http://www.site.uottawa.ca/~awilliam/papers/

[21] A. W. Williams, "TConfig", 2004, http://www.site.uottawa.ca/~awilliam/TConfig.jar

[22] I. Zayour, *Reverse Engineering: A Cognitive Approach, a Case Study and a Tool.* Ph.D. dissertation, University of Ottawa, 2002, http://www.site.uottawa.ca/~tcl/gradtheses/izayour