Set Operations for the Unified Modeling Language

Petri Selonen

Tampere University of Technology, Institute of Software Systems,
P.O. BOX 553, FIN-33101, Tampere, Finland
petri.selonen@tut.fi

Abstract. Software design is an iterative and collaborative process. There is a need for techniques for composing and decomposing the constructed models, and for comparing them against each other to avoid inconsistencies and to promote understanding of the system under design. To address these issues, set operations for the Unified Modeling Language (UML) are discussed. The operations are defined tool-independently using the UML metamodel and are being implemented as a part of a larger framework for manipulating UML models. The operations are put in context with other similar techniques, including composition relationships of Subject-Oriented Design.

1 Introduction

The Unified Modeling Language (UML) [11] has established itself as an industry standard for describing and designing software systems. UML offers different diagram types to view a system from different perspectives, on different levels of abstraction, and at different stages of software development process. While in principle a system design could be described as a monolithic model, this is very rarely the case in practice. Typically, system design is a collaborative effort, performed by several engineers or design teams, each focusing on different viewpoints or concerns. Software engineering processes are often incremental and iterative by nature (e.g. [7], [5]) and produce new models throughout the whole design life cycle. In addition, there often exists a structural mismatch between the specification paradigms across a software development life cycle, caused by a *scattering* and *tangling* effect [3]. Scattering is caused by individual requirements affecting several units of interest in a design model, while tangling is caused by a single unit of interest catering for several different requirements.

While these problems are well recognized, there exists relatively modest tool support addressing them. There is a need for mechanisms for exploiting the dependencies between individual UML models. Such composition and decomposition mechanisms should contribute to reduced complexity and improved comprehensibility of the system under design [17]. While only few UML CASE tools offer model merging capabilities (e.g. Rational XDE [12]), to the knowledge of the author, no tool provides support for performing operations for extracting only the common parts of two models, or only the parts unique to one model.

The mechanism proposed in this paper is *UML set operations*. A UML set operation is a binary operation that, on the basis of two UML diagrams of a particular type, produces a new UML diagram of the same type. Three UML set operations are discussed: union, intersection, and difference. The operations can be used as such, but their true power arises when accompanied with a suitable set of model operations [9], and as a part of a larger framework for manipulating UML models [1]. Because of the connectivity properties of the set operations, they can be combined into more complex expressions (e.g. symmetric difference).

Model merging is supported by the union operation when used for merging the modeling artifacts produced by several designers or design teams. Similarly, the increments produced during a software development process can be integrated into the existing system model. As an example of such a process, consider a use case driven, incremental software process producing an initial base structure model (as a class diagram) and for every use case, a set of sequence diagrams. These sequence diagrams can be transformed into a class diagram [15] and merged together with the existing system model using a union operation. Model merging is also a prerequisite for constructing a framework supporting chains of model manipulation operations. Set operations are a fundamental concept when combining the results of e.g. transformation and projection operations together [9].

Composing and decomposing a model is a key activity when supporting the construction of the models from model fragments, each describing individual concern or a cross-cutting aspect of the system. Such mechanisms are integral to paradigms like Subject-Oriented Design that rely on support for the separation of concerns [2] [17].

Model comprehension is supported when the designer is allowed to create transient views for exploring the parts of the system she is interested in. Designers and stakeholders use different diagram types for exploring the system from different viewpoints or levels of abstraction. Model comparing and slicing can be used as a technique for extracting the parts of a model that contribute to a given viewpoint or stakeholder interest. An example of such a procedure is comparing different versions of a system to promote understanding of model increments. *Visual differencing* can be used as a complementary technique for gaining a better comprehension into the system at hand.

2 Specifying Set Operations

Of the nine different UML diagram types, set operations are defined for the specification level diagrams focusing on the structure of the system under design: class diagram, component diagram, and deployment diagram. In addition, the operations are applicable to specification-level use case diagrams. The remaining diagram types are discussed in Section 4. The origin and purpose of the models define how they can be meaningfully interpreted: the quality of the input operands determine the usefulness and quality of the resulting diagram.

2.1 Definitions

The architecture of UML is based on a four-layer metamodel structure, which consists of the following layers: user objects, models, a metamodel, and a meta-metamodel. The meta-metamodel layer is the infrastructure for a metamodeling architecture and it defines the language for specifying metamodels. The metamodel layer is an instance of the meta-metamodel and defines the language for specifying a model. The user model is an instance of the UML metamodel, consisting of metaclass instances and links between them. The set operations are defined using the standard UML metamodel.

We adopt the definitions for a model, a diagram, and a diagram type from [16]. A model describes a UML metamodel instance, a diagram describes a graphical representation of a model, and a diagram type describes a particular kind of a diagram proposed by the UML Notation Guide. A *model element* refers to a UML metaclass instance, while a *property* of a model element refers to a meta-attribute instance belonging to the model element. The *state* of a model element is defined by the property values.

The elements are divided into *primary elements*, *secondary elements*, and *relationships*. Primary elements (e.g. class, interface) are composites that can contain secondary elements that do not exist on their own (e.g. attributes, operations). The division to primary and secondary elements is drawn along the composition relationships in the UML metamodel. Relationships (e.g. dependency, association) are instances of metaclasses that are derived from the UML Relationship metaclass. The *primary parent elements* of a secondary element e_n refer to the elements in chain e_1, e_2, $\ldots$, e_{n-1}, e_n, where e_1 is a primary element, $e_2 \ldots e_n$ secondary elements, and for each element e_{k+1} ($0 < \mathrm{k} < n$), e_{k+1} belongs to e_k.ownedElements.

The *context of a secondary element* is determined by the parent element that the secondary element belongs to, and by its connecting mandatory meta-association instances (i.e., meta-associations with a lower multiplicity bound greater than zero). The *context of a relationship* is determined by its end model elements. *Reconciliation* of model elements refers to the process of determining the state for a model element representing a modeling concept described by several corresponding model elements.

Figure 1 shows a subset of the UML metamodel for structure diagrams. The metamodel shows the metaclasses, their meta-attributes and meta-associations, and generalization relationships. According to Figure 1, Class is a primary element, Attribute, Operation, Parameter, and AssociationEnd secondary elements, and Generalization and Association relationships. The context of Generalization is determined by its child and parent Classifiers, while the context of Attribute is determined by its parent and type Classifiers. The examples in this paper are presented as instances of the metamodel in Figure 1.

A *set operation* is defined with a signature $\mathrm{D} \times \mathrm{D} \longrightarrow \mathrm{D}$, where D is a UML diagram of a given type. The operations are divided into three phases: (1) establish a correspondence relationship between the model elements of the input diagrams, (2) perform reconciliation on the corresponding model elements, and

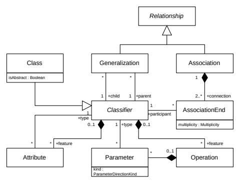

Fig. 1. Subset of a UML metamodel for structure diagrams (Foundation::Core)

(3) perform the actual set operation. Because of the connectivity properties of the set operations, they can be combined into more complex expressions.

2.2 Correspondence Relationship

Given a set operation Op(A,B) with input operands as UML diagrams, a *correspondence relationship* needs to be established between their model elements. If two model elements correspond, they represent the same semantic design concept and are considered as the same element when performing the a set operation (e.g. union, intersection, difference). The correspondence relationship can be a many-to-many relationship, showing candidate elements that could correspond. In the following definition, we resort to the two most important modeling concepts in UML: a metaclass and a name.

Definition 1. *Given models A and B, model elements a belonging to A and b belonging to B are said to* correspond *if they have the same metaclass, the same name, and a corresponding context.*

Definition 2. *Given models A and B, model elements a belonging to A and b belonging to B are said to* uniquely correspond *if they correspond only to each other.*

While all other properties of corresponding model elements might differ from one model to the other, we require that the naming conventions used are consistent. Having unambiguous and well-defined naming conventions is an integral part of good software engineering practices. This becomes evident with large-scale industry software production. When dealing with models synthesized by a reverse engineering process, model elements usually have unique identifiers by default. Unnamed primary elements do not belong to correspondence relationships.

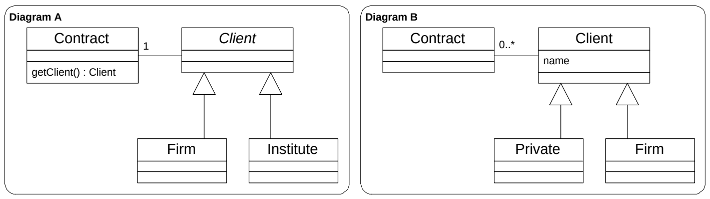

Fig. 2. Example UML diagrams

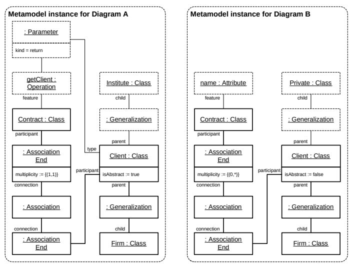

Fig. 3. Metamodel instances of diagrams A and B

Figure 2 shows two example UML class diagrams. While overly simple, they are sufficient for demonstrating the techniques presented in this paper. Figure 3 shows metamodel instances for the diagrams A and B presented in Figure 2. The uniquely corresponding model elements and their links (meta-association instances) are shown in boldface. The interesting properties of model elements are also shown as attribute values.

2.3 Operation Definitions and Reconciliation

While the conventional set-theoretical union and intersection are symmetric by nature, our interpretation of the corresponding UML set operations is asymmetric. This results from the fact that corresponding elements can have different and conflicting states. From a structural point of view, union and intersection do produce symmetric results.

Definition 3. Union(A, B) *contains all model elements from models A and B. Uniquely corresponding elements are merged and their meta-association instances are redirected to the merged model elements.*

Definition 4. Intersection(A, B) *contains only the uniquely corresponding model elements.*

How the potential differences between the states of corresponding model elements should be interpreted depends on their relationship. The default strategy is asymmetry: to favor the first operand over the second one. Section 3 discusses additional reconciliation strategies and directives. For an operation Op(A,B), the participating operands can relate to each other in the following ways:

– Abstraction level. The model at a lower level of abstraction is typically more complete and should be favored over the other.
– Evolutionary phase. The model at a latter stage of evolution is typically more refined.
– Level of completeness. The more complete model could be preferred over a model fraction.

The interpretation of difference in the UML context is somewhat different than the conventional set-theoretical definition. Due to the nature of UML models, difference is defined in two forms: a *preserving difference* and a *destructive difference*. The former preserves all those primary elements whose secondary elements do not have corresponding element in the other diagram. The latter removes all those secondary elements whose primary element has a corresponding element in the other diagram.

Definition 5. Preserving difference(A, B) *contains all model elements in A that do not have a uniquely corresponding model element in B, the primary parent elements of the included secondary elements, and the end elements of the included relationships.*

Definition 6. Destructive difference(A, B) *contains all primary elements in A that do not have a uniquely corresponding element in B, their secondary elements, and the relationships between them.*

Figure 4 shows the results of union, intersection, preserving difference, and destructive difference between example diagrams A and B of Figure 2. The operations are performed on basis of the unique correspondence relationship shown in Figure 3. With union, all model elements are included in the resulting diagram. With intersection, only the common model elements are included. Preserving difference leaves class Contract intact because of the operation getClient(), while destructive difference removes the class. By asymmetry, class Client remains abstract in every diagram, and the multiplicity of the Contract association end is 1.

The operation definitions themselves do not guarantee that the resulting UML diagrams conform to UML's abstract syntax or its well-formedness rules. The former case occurs very frequently in practice with diagrams that are not complete specifications. For example, diagram B in Figure 2 is invalid in the sense that it does not define a type for attribute *name*. An example latter case is a union resulting with circular inheritance hierarchies. The set operations are defined only for those input diagrams that produce well-formed results.

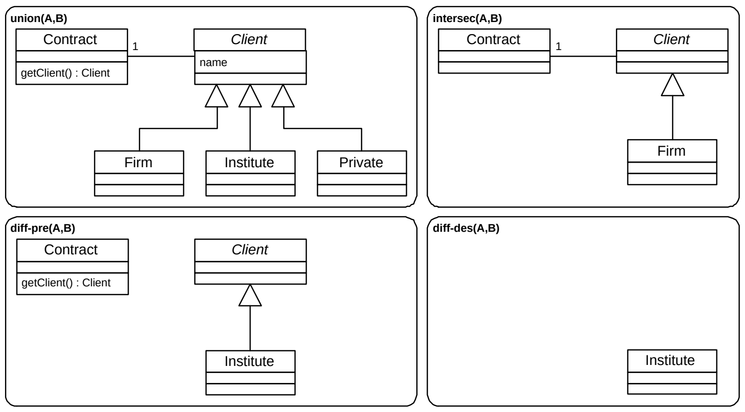

Fig. 4. Example results of union, intersection, preserving difference, and destructive difference

2.4 Past, Present, and Future Implementation

The set operations have been originally implemented and evaluated on top of TED [19], a Nokia proprietary UML CASE tool. The operations were used in a study comparing reverse engineered UML models of a moderate sized Java program Mathaino produced by different reverse engineering tools [8]. The size of the model was around 450 classes, of which the core part selected as input operand was around 100 classes. The results of the study suggest that these techniques can be implemented and used in a meaningful way for real-life software engineering tasks.

Initial prototype versions of the operations have been implemented on top of xUMLi [1], a tool-independent UML processing platform. xUMLi offers an implementation of the UML metamodel conforming to the UML version 1.4, import and export functionality for XMI, Rational Rose [13], and TED. It further offers a visual scripting mechanism for defining tasks for software engineering processes, and an API, supporting COM automation, that allows the model manipulation operations to be written in high-level scripting languages like Python.

3 Extending the Approach

The principles presented in this paper are straightforward and rely heavily on the sensibility of the input diagrams, and on the selected software engineering process to produce meaningful and unique names for identifiers. This is a reasonable assumption and further supported by the fact that a vast majority of the similar approaches rely on these techniques ([3], pp.58).

It is possible to extend the approach with a set of heuristic rules that pay attention to the patterns present in the diagrams, on the states of the model elements, etc. The simple approach presented in Section 2, relating unnamed relationships with identified end elements, could be extended to handle larger model fragments.

One potential, albeit heavy way to extend the approach is to introduce a set of heuristic rules that would give approximations on the semantic distances between model elements. With a combination of such measures, together with suitable threshold values, it might be an interesting approach for determining counterpart elements. For example, if two unnamed classes would have almost the same set of UML Features (i.e., attributes, operations), the designer could be notified and given a chance to confirm the assumption. Even if the elements would not represent the same semantic concept, this information could nevertheless be useful for increasing the designer's insight of the model, and could possibly indicate a need for restructuring the design (e.g. extracting the shared features into a common parent class).

As always, the possibility for user interaction should be available. Even though the set operations can be performed automatically, the designer should be given a chance to affect how the operations are executed. Analogously to the composition relationship concept of Clarke [3], the designer can define *directives* that can override the default functionality. The directives can be arbitrary filters, OCL constraints, or new strategies appended to the original set operation implementation. The designer can either define direct dependencies, or she could be given a chance to interactively decide which elements should be included in the correspondence relationship and which not.

4 Other Diagram Types

This section discusses how set operations could be defined for the UML diagrams types not covered in Section 2. While by definition it is possible to perform the operations on any two arbitrary metamodel instances, deriving a meaningful correspondence relationship between their model elements becomes much harder when issues like instance identity and behavior are taken into account.

4.1 Interaction Diagrams

Interaction diagrams (i.e., sequence diagrams, collaboration diagrams) describe how a set of instances collaborates via exchanging messages to perform a certain task. Interaction diagrams are not full specifications. Instead, they give examples of how a particular task is performed by the system.

Since an interaction diagram describes its behavior with a set of messages, finding a correspondence relationship between two interaction diagrams becomes essentially an effort on finding corresponding messages. This is clearly not straightforward. Since every message represents communication at a single, unique point in time, how can one decide which messages correspond to, or how they are interleaved with, each other in two arbitrary interaction instances? While the correspondence could be concluded with explicitly defined time markers, state information, or interaction patterns preceding the message, there does not seem to exist a general schema for deriving such a relationship. While in theory UML facilitates the representation of parallel execution in a single sequence diagram, in practice its notation is not well suited for such algorithmic representation of interaction. Thus, merging of a set of sequence diagrams is performed more meaningfully if they are synthesized into a state machine [10].

As a pragmatic viewpoint to interaction diagram integration, if the two interactions can be considered orthogonal, their union (merge) can be meaningfully defined as concatenation, possibly with explicit ordering. This approach can be taken if, for example, the two interactions belong to individual aspects or Themes [4]. Our implementation follows this approach.

4.2 Statechart Diagrams and Activity Diagrams

There exist several ways of composing statechart diagrams. However, name-based techniques are very seldom feasible, since statechart diagrams specify the full behavior of a system or a subsystem under design: they are typically complete and not decomposed into separate parts. When the behavior of a larger system is defined using a set of statechart diagrams, the diagrams are usually subdiagrams that can be composed to concurrent state machines at the higher level. When a statechart diagram is composed of several parts, this is usually done by composing the resulting statechart diagram synthetically. In principle, if all states were named with unique identifiers, the techniques presented in this paper could be applied. Methods for composing statechart diagrams are introduced, for example, by Glinz [6] and Schönberger et al. [14]. Glinz introduces a method for formal composition of scenarios into an integrated, consistent statechart-based model, while Schönberger et al. presents an algorithm for synthesizing UML statechart diagrams from a set of collaboration diagrams.

4.3 Object Diagrams, Deployment Diagrams, and Component Diagrams

Object diagrams are instance level diagrams. With deployment diagrams and component diagrams, there exists two alternative forms: specification level diagrams and instance level diagrams. Only the latter ones are concerned here. It is possible to derive a meaningful correspondence relationship between elements in instance diagrams. The most obvious situation occurs when the instances have dedicated unique names. Instance diagrams typically describe snapshots of the system at some point in time, so it is often not useful to perform set operations on them.

5 Related Work and Discussion

While there exists numerous CASE tools supporting UML , these tools generally fall short in exploiting the dependencies between individual model elements in model fragments. Of the current tools, to the knowledge of the author, Rational XDE [12] provides most support for merging UML models. By default, the XDE model comparing and merging tools assume that the input UML models have a common ancestor, that is, the elements share the same unique IDs in the two or more model files. However, XDE also provides functionality for generating a correspondence relationship between model elements not sharing a common ancestor.

With XDE, the user can select two or more model files to be compared and/or merged together. The results are shown as parallel repository tree structures, where the nodes graphically indicate whether given model elements are recognized, and how they have changed from one model to the next. The potential problems with the nodes, *conflicts*, can be solved automatically using default functionality, or they can be solved by hand. If the user decides to merge the models, the tool generates a new, merged model. While the model is merged, the individual diagrams are not. The function for deriving the correspondence relationship is proprietary and not publicly accessible. Consequently, the user cannot extend the process of generating the correspondence relationship. Furthermore, XDE does not merge diagrams, nor does it give the user a chance to browse the UML model in a way that would show which parts of the model are common to all included models, and which originate from a single model.

With the approach described in this paper, the user can affect the process of deriving the correspondence relationship and select individual (sub)packages or diagrams as starting points for the process. Furthermore, the results from the correspondence relationship can be used not only for merging the models, but also for generating their intersection or difference. The resulting model can be further modified before exporting back to the CASE tool used, if indeed it is exported: the resulting model can also be seen as a transient view to the system, thus being used as means for navigating the design for gaining a better insight.

From the viewpoint of Subject-Oriented Design [2], the UML set operations can be seen as a mechanism for composing the system from, and decomposing it into smaller model fragments, each describing a single aspect or a concern. The research done with Theme/UML [18] comes perhaps closest to the work presented in this paper. The UML set operations can also address the problem with the structural mismatch between the specification of requirements for software systems and the specification of object-oriented systems. This mismatch stems from the fact that there are typically several non-functional units of interest and cross-cutting features present in the requirements of the software that are scattered throughout the object-oriented model. At the same time, individual modeling elements cater for several different features simultaneously, thus leading to tangling. This structural mismatch results in reduced comprehensibility and poorer traceability of design models [17].

Clarke addresses these issues with composition semantics that is an analogous mechanism with the set operations. Clarke provides a very comprehensible study on the merging of the individual UML metamodel elements. One of the main differences of the work presented in this paper and by Clarke is that while Theme/UML extends the UML metamodel with several new concepts, our approach relies solely on the standard UML. With Theme/UML, the user has to explicitly define the composition relationships between individual subjects. The work in this paper relies on a visual scripting language to provide a description of the process that is to be applied. Clarke offers the user the concept of composition patterns, a template mechanism for defining collaboration and behavior. Furthermore, Theme/UML takes a stand on the implementation mechanisms to be used and introduces mechanisms for merging operation calls. On the other hand, the work described in this paper relies on a concrete implementation on a tool-independent model-processing platform, which has been already implemented and integrated with import and export facilities for, e.g., Rational Rose.

This paper presented UML set operations, motivated their use, and placed the approach in context with other work parallel to the one presented here. The operations have been implemented and tested on moderate size case material. The operations are intended to be transferred on top of the xUMLi platform and expected to be evaluated with two large-scale industry case studies during year 2003. In addition to their usage as a part of a larger framework for defining model processing operations related to specific software engineering tasks, these techniques also have potential usage scenarios related to the Subject-Oriented paradigm. Set operations on UML models provide a simple, unifying basis for model composition required for various purposes in UML-based software development.

Acknowledgements The author wishes to thank Tarja Systä, Kai Koskimies, and Jari Peltonen for their valuable comments, and the PRACTISE/UML team at Tampere University of Technology. This research was financially supported by the Academy of Finland.

References

1. Airaksinen, J., Koskimies, K., Koskinen, J., Peltonen, J., Selonen, P., Siikarla, M., Systä, T.: xUMLi: Towards a Tool-Independent UML Processing Platform, in Proc. of NWPER-02, Copenhagen, Denmark (2002)
2. Clarke, S., Harrison, W., Ossher, H., Tarr, P.: Subject-Oriented Design: Towards Improved Alignment of Requirements, Design and Code, In Proc. of OOPSLA'99, Denver, Colorado, USA (1999)
3. Clarke, S.: Composition of Object-Oriented Software Design Models, PhD Dissertation, Dublin City University (2001)
4. Clarke, S.: Extending standard UML with model composition semantics, in Science of Computer Programming, Vol. 44 No. 1, Elsevier Science (2002) 71-100.
5. D'Souza, D., Wills, A.: Objects, Components, and Frameworks with UML, the Catalysis Approach, Addison-Wesley (1999)
6. Glinz, M.: An Integrated Formal Model of Scenarios Based on Statecharts, Proceedings of Software Engineering, Lecture Notes in Computer Science 989, Springer-Verlag (1995) 254-271
7. Jacobson, I., Booch, G., Rumbaugh, J.: The Unified Software Development Process, Addison-Wesley (1999)
8. Kollman, R., Selonen, P., Stroulia, E., Systä, T., Zündorf, A.: A Study on the State of the Art in Tool-Supported UML-Based Static Reverse Engineering, in Proc. of WCRE'02, Richmond, Virginia, USA (2002)
9. Koskinen, J., Peltonen, J., Selonen, P., Systä, T., Koskimies, K.: Towards Tool-Assisted UML Development Environments, In Proc. of SPLST'2001, Szeged, Hungary (2001) 1-15
10. Mäkinen, E., Systä, T.: MAS - An interactive Synthesizer to Support Behavioral Modeling in UML, In Proc. of ICSE 2001, Toronto, Canada (2001) 15-24
11. Object Management Group: OMG Unified Modeling Language Specification Version 1.5 (2003). On-line at http://www.omg.org/uml
12. Rational Software Corporation: Visual Modeling, Design and Development with Rational XDE Home (2003). On-line at http://www.rational.com/products/xde
13. Rational Software Corporation: Rational Rose (2003). On-line at http://www.rational.com
14. Schönberger, S., Keller, R., Khriss, I.: Algorithmic Support for Transformations in Object-Oriented Software Development, Technical Report GELO-83, University of Montreal (1998)
15. Selonen, P., Koskimies, K., Sakkinen, M.: How to Make Apples from Oranges in UML, In Proc. of HICSS'34, Hawaii, USA (2001)

16. Selonen, P., Koskimies, K., Sakkinen, M.: Transformations Between UML Diagrams. Journal of Database Management, 14(3) (2003), pp. 37-55. To appear.
17. Tarr, P., Ossher, H., Harrison, W., Sutton Jr., S. M.: N Degrees of Separation: Multi-Dimensional Separation of Concerns, In Proc. of ICSE'99, Los Angeles, California, USA (1999)
18. Theme/UML: Extending Standard UML with Model Composition Semantics (2003). On-line at http://www.dsg.cs.tcd.ie/∼sclarke/ThemeUML/
19. Wikman, J.: Evolution of a Distributed Repository-Based Architecture, In Proc. of NOSA'98, Blekinge, Sweden (1998)