Supporting End-User Tailoring of Component-Based Software - Checking Integrity of Compositions

Markus Won and Armin B. Cremers

Institute for Computer Science III, University of Bonn
Roemerstrasse 164, 53117 Bonn, Germany
+49 228 73 4506,
{won|abc}@cs.uni-bonn.de

Abstract. Component-based software can be used to build highly tailorable and therefore flexible software systems. To support end-users while tailoring their applications themselves different approaches were discussed. This paper describes an interactive integrity checking concept as a support for end-user tailoring. It is based on the idea that developers can describe the "right" use of their components as well as they can describe properties which belong to specific groups of applications. Those information can be used to check the application composed out of such components during tailoring time. Thus, this leads to higher-quality tailoring and more stable and better fitting tailored applications. Furthermore, the learning of tailoring activities will improve as well as a better understanding for the resulting software can be achieved.

1 Introduction

Most of the software sold nowadays are off-the-shelf products designed to meet the requirements of very different types of users. One way to meet these requirements is to design software that is flexible in such a way that it can be used in very different contexts. This flexibility can be achieved by tailorable design architectures. The idea behind this concept is that every user can tailor his software in the way that it meets his personal working contexts best. In this way the parts of the specification and design can be done during the use. According to the ideas of participatory design users then have better possibilities to form the software in the way they want to.

Component-based architectures were basically developed with the idea of higher reusability in the way that parts of software can be used to build those highly flexible software. The same operations that are used when composing software during development (i.e. choosing components, parameterizing them, binding them together) are then the basis for a tailoring language.

Especially if the components have a GUI representation it is quite easy to design appropriate tailoring languages including a visual representation. Although both - the idea of adapting an application seen as a composition of components and the use of visual tailoring language which supports only very few tailoring mechanisms - are well understood by users, there is a need for further support during tailoring time.

Frese et al. [3] discuss the amount of failures (that take about 12 % of working time) that users do during their work. Especially they present empirical studies that show that not only beginners but especially experienced users do failures. Furthermore they distinguish between different types of failures i.e. those which can be corrected easily and those that cannot even been located directly. What can be learned from this study is that doing failures during use and especially during tailoring of software is normal or even an intended using behavior. Therefore supporting mechanisms which can help finding failures, understanding the composition or testing the current composition in an exploring environment seem to be very useful.

This paper describes a new approach which uses an integrity check not only to ensure the correctness of a working application but support users interactively during tailoring time (as a guidance). An integrity checking mechanism not only has to control the validity of the composition but shows the source of error and also can give hints or correct the composition itself. The idea behind that is not only to assist the end-user tailoring activities but also to stimulate the learning of the tailoring language, the proper use of the components, and of the functionality of the resulting application (by looking behind the scenes).

In the following section the context of this work is described in more detail focusing on tailorable component-based software. After that we will concentrate on the idea of integrity and integrity checking which are both common in the computer science. Here a short overview on the state of the art is given. The following sections then focus on the idea of integrity checking as a support for end-user tailoring.

2 Component-based Tailorability

In the last years, component-based architectures [14] have become quite fashionable in the field of software engineering. A very important property of a component is its reusability and the independent development of components. Therefore, components can be seen as small programs which can exist and run alone but may also be combined with other components. Thus, an application normally consists of several components that are connected with each other. Applications based on component architectures are designed by selecting one or more components and set up connections between them.

Furthermore, applications can be easily enhanced by adding new components to the existing set. Mainly there are three different operations which are used to design applications with component-based architectures.

- adding or removing components,
- changing the parameters of one component, or
- linking two components according to their specified interfaces

These operations can be used during construction time as well as in case of tailoring an existing application [13]. The tailoring language then consists of the same three simple operations. The operands within the tailoring language are the components. According to ease the learning of such a tailoring language it is due to the designer of an component set to choose the right granularity. Thus, beginners start to compose their own applications by using few components which provide for a great amount of functionality whereas more experienced users can combine more components which are smaller. [1] A second step to ease the learning therefore is to allow for a layered architectures [16] which means that several components can be stick together and saved as one larger.

This idea was implemented in the FREEVOLVE platform [13] which is based on the FLEXIBEAN component model [13]. Several visual tailoring environments were developed and evaluated. Most of the users easily understood the concept of component-based architectures as there are several domains in real life which are very similar to that concept. One of the remaining problems is that it is not clear to the users in which situation which component to choose or how the components have to be bound together. Seen from the perspective of the developers the semantics of the components as well as their parameters and interfaces have to become more transparent.

There are some other concepts [9] to create a better understanding of tailoring languages as well as of the resulting applications. In the following those concepts as well as their integration into a search tool based on the ideas of the FREEVOLVE platform are described shortly.

First, Mackay [4] found that the lack of documentation of respective functions is a barrier to tailoring. Manuals and help texts are typical means to describe the functionality of applications. Thus, all simple components within the FREEVOLVE were extended by *descriptions*. Furthermore, help texts can be added to abstract components by the composers. Such an abstract component and the belonging description then can be annotated by other users. So, discussions could support the deeper understanding of an tailoring artifact.

Mackay [4] and Oppermann and Simm [8] found that *experimentation* plays a major role in learning tailoring functions. "Undo function", "experimental data", "neutral mode", etc. are features which support users in carrying out experiments with a system's function. Those functions were integrated into the FREEVOLVE platform.

Furthermore, an exploration mode was added which simulates a work space with experimental data. Users who want to explore the changed functionality of a tailored application can do it by looking at the impacts on the virtual work space when using the application.

Examples provided by other users are an important trigger to tailor [15]. While the FREEVOLVE tailoring environment supports experiments, the ques-

[1] The FREEVOLVE framework (described in the following) allows for hierarchical construction of components. Thus an - so called - abstract component can contain several other components. So, the usage of components can be eased without restrictions to the flexibility.

tion how to support the exploration of compound components or elementary components remains. These artifacts cannot be executed in the environment by themselves. A solution is to exemplify the use of a component by a small characteristic example application.

Checking: Automatic visualized integrity checks can prevent from building pointless application (cf. [17]). Here integrity checks are used not only to prevent from failures but in the way of supporting users when tailoring their applications in the way that making failures and getting corrections on that can be seen as learning. Thus in the following we will concentrate on how integrity checks can be used to ease the learning of a component-based tailoring language.

3 State of the Art - Integrity Control

Integrity checking is widely common within different fields of computer science. In the following we will list the techniques which are used in our concept (see next section):

- **Database integrity:** The domain of information systems is concerned with the consistent structuring and storing of information. During design time scheme transformations can help to create a appropriate data models [12] which prevents from inconsistent data. During runtime information systems maintain the consistent data basis by controlling new input or changes. That can be done by constraints or triggers [12]. Both concepts can be transferred to the component-based software approach. Whereas the design of a database (normalizing the schemes) can be compared to the design of a component model, constraints that ensure certain conditions to the elements within the database, can also be used to control a composition's characteristics. In this paper we focus on integrity conditions that can be checked during tailoring time.
- **Software engineering (design by contract):** The design by contract-concept [5] concentrates on the idea that additional conditions can be added to methods when they are to be invoked. So, a pre-condition can be formulated which has to be fulfilled by the invoking object. On the other hand after the method has ended a post-condition is guaranteed. For Java there are special tools (i.e. [4]) which deal with this approach focusing on better error removal and therefore faster development. Here the idea is to enhance the conditions that have to be fulfilled if an interface should be used during run-time. An similar way to control run-time conditions to the interface usage is described in the following.
- **Software engineering (additional semantic interface descriptions):** Behavior Protocols [10] add additional information to the interface definition by describing how a component can be used (which methods can be called in which order). If the component is used, the caller has to comply with this protocol. Such protocols describe the usage of an interface. As the protocols have to be described during design-time one can assure that interfaces can

be used only conformably to the protocol. Here, very interesting conditions to the interfaces are defined. In our approach (described below) we also try to address the dynamic part of the interface's use. Furthermore, we try to integrate problems which can occur if we do not address the complete flow of events that can be generated by sequent use of interfaces. An similar approach describes Reussner [11]. Here not only two components and the connections between them are investigated. Instead of additionally there is the idea that a component describes what she "needs" as a caller out of its "environment" (surrounding runtime environment, other components, etc.). This information is targeted on the interfaces. An component's input interface then can be set into relation to the output interface. Thus not every condition has to be fulfilled at the input side if the corresponding output port is not used. These dependencies between a component's input and output port can be described by additionally information.

– **Application templates including integrity constraints:** Birngruber and Hof [1] describe a group of applications by a plan. This plan consists of conditions on the use of special components, parameters which are dependent on others, etc. The idea here is that the so called CoPL Generator analyses the plan and builds an application according to those conditions. The composition is done semi-automatically as there is user interaction where decisions can be made. So in the broader sense of the word this can be seen as "user-participated" automatic construction.

In the following we will describe our approach of extending the FREEVOLVE platform by an integrity check. This is done in two steps: First integrity conditions and a describing language have to be designed. After that there has to be integrated an integrity checking mechanism into the platform which interacts with the tailoring end-users marks errors, gives hints, etc. (see above). The design of the interaction between the integrity check and the users should be done with the focus on easy understanding and helping to learn how to tailor by assembling components.

4 Supporting Tailorability by Integrity Checks

This section describes different kinds of integrity conditions. There are all based on the assumption that special meta information on the components and about domains or groups of applications is given or can be added to the component set.

The approach described in the following is based on the tailoring techniques mentioned above. Different techniques mentioned above are to be integrated into a new component integrity model. Figure 1 illustrates this approach. On the left several techniques concerning integrity are listed. They can be matched to different aspects of a composition (single components, whole composition, relation between two components). According to the idea that end-user tailoring should be supported they are matched to the three tailoring operations, that are:

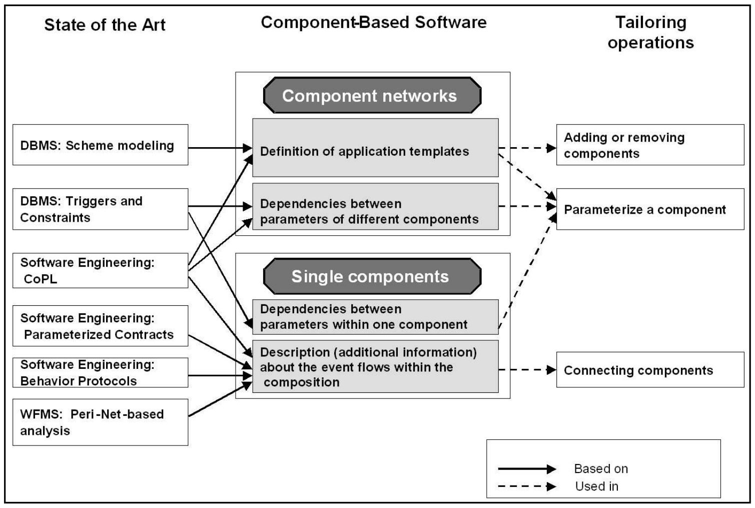

State of the Art
Component-Based Software
Tailoring operations
Component networks
Single components
DBMS: Scheme modeling
DBMS: Triggers and Constraints
Software Engineering: CoPL
Software Engineering: Parameterized Contracts
Software Engineering: Behavior Protocols
WFMS: Peri-Net-based analysis
Definition of application templates
Dependencies between parameters of different components
Dependencies between parameters within one component
Description (additional information) about the event flows within the composition
Adding or removing components
Parameterize a component
Connecting components
Based on
Used in

Fig. 1. Different types of integrity

- **Parameterization of components:** There can be mutual dependencies between different parameters within one component. They can be checked within the component's functionality. Admittedly explicit descriptions of those dependencies not only ensure the correct parameterization but also explains parts of the functionality, and thus, helps learning and understanding the component's semantics. External dependencies between parameters of different components have to be described explicitly. These dependencies restrict the idea of the independence of two or more components. According to application templates (i.e. for a the "class" of movie player applications) such restrictions can be very helpful. Here the idea is, that some components only work well in a set if certain parameters are bound in the right way. Those integrity conditions can be compared to constraint in DBMS. According to this comparison automatic corrections can be seen as triggers where the action is the adjustment itself.
- **Adding/removing components:** Global constraints described in application templates ensure the existence of some components. Furthermore complex dependencies between component's existence (if component A is part of application X then there must be a component B or a component C) can be described.
- **Changing the connections:** In component-based architectures not only two interacting components have to be investigated (according to the integrity of an application) but the whole composition can be seen as a network. Thus as well as workflows [2] they can be seen as directed graphs and therefore analyzed in a similar way [2]. The *Event Flow Integrity* [16] as the heart of the new underlying integrity concept checks the used ports and

the dependencies between them. This concepts is described in detail in the following section.

5 Event Flow Integrity (EFI)

In the following we will evolutionary develop the idea of checking the event flow integrity of a composition.

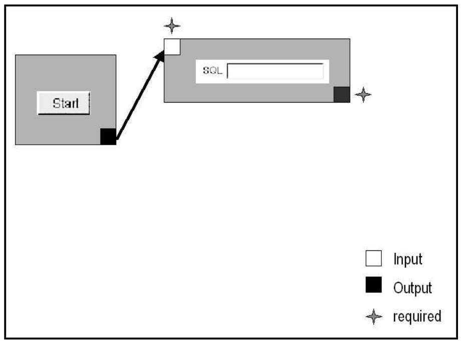


Fig. 2. Simple Approach

First we divide the ports (input and output) of each component into two subsets: The optional ports *may* be used whereas required ports *have* to be connected. Figure 2 illustrates this idea [2]. The search engine has two ports - one input and one output port - which have to be used in a valid configuration. The check algorithm itself is obviously a very simple one. First of all, it has to look whether all needed components are within the composition (i.e. a search tool without the search engine would not make sense) and then it would just have to check if the required ports of all components are in use (are connected to ports of other components).

Therefore the integrity check system would regard the configuration of the application as invalid because the required output port of the search engine is not connected to any other component.

The simple approach has some problems which are discussed in the following. First of all, we could integrate a new (third-party) component into our component set. This new component has no required ports. In our example (cf. fig. 3) we add a switch component (the found objects are divided into two subsets according to a certain condition). The binding between the search engine and the switch component is sufficient to declare the application as valid though we

[2] The example uses a set of components that can be used to compose a search tool. For more details see [16]

still do not have an output component which we wanted to enforce by marking the output port of the search engine as "required".

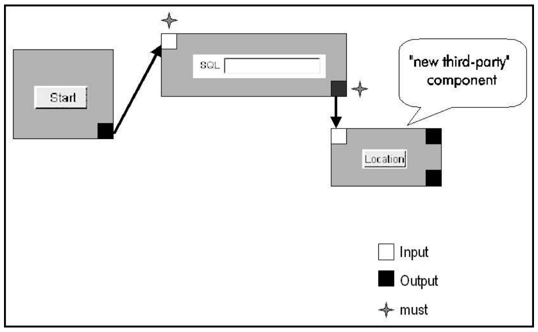


Fig. 3. Integrating new third-party component

This problem could be solved by classifying new components whether they are valid components in the way that they provide the right ports and the ports are marked in the proper way. For instance, this could be done by the administrator of the system. Figure 4 shows the correct version of the example described above. The switch now has two required output ports which are marked by the stars.

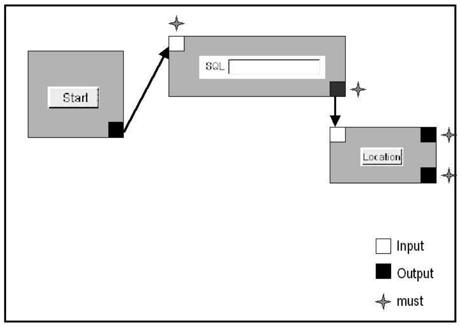


Fig. 4. Extending the markups

But this will not solve our problem in general. What we want is (in our example) that the search engine is always connected to an output component which the filter component definitely is not. More generally spoken, we have components which produce something whereas others consume something. Therefore we can divide our components into producers and consumers of certain events. For example, the search engine is a producer of a search result, whereas every output component (i.e. a output window) would be a consumer. A third category could span all components which are neither producers nor consumers of an event, that is for example the switch component (or a filter) which only transforms an incoming event and forwards it.

In the following we will discuss this approach in more detail. According to that we will present a solution to our problem taking into account the concept of "transitivity of connections".

5.1 Transitivity of Connections

The concept is based on the idea that every component "knows" which messages it produces (that have to be consumed) and which input can be consumed.

For instance, a search engine produces a search result and on the other hand consumes certain input values to specify the search query properly. Thus we may say the search engine is a producer of a search result, and a consumer of such a search result could be any output component (i.e. window, counter, etc.).

An integrity check could then examine every single used component and look for essentially produced outputs or consumed inputs. The search engine then produces a search result and the integrity check now has to look whether there is a component which is directly or transitively connected to the search engine and is able to consume the search result. In the example depicted in Fig. 4 the switch is not a consumer of a search result though it only pipes it in a certain way. Therefore in order to make this application valid we have to add an output component which then has to be connected to one of the output ports of the switch component.

5.2 Problems and Extensions

The concept described above and the resulting algorithm just give a first idea. But there are still some difficulties which are discussed in the following:

– Tokens are not typed (in contrast to the ports).
– There might be components which can produce or consume more than one token of a certain type.
– The basic concept only allows for "required" producers and possible consumers.
– Circles which can occur if components are connected are not taken into account.

Typing the tokens: As we have seen above the ports in the search tool are typed. Thus we have to distinguish between several types of events which can be passed between the components. Therefore we have to have more types of different tokens which is not a problem in general.

Producing or consuming more than one token of a kind: Another point that we have to address is that there might be components which double tokens or others which can consume exactly one or more than one or an infinite number of tokens of the same type. For example, a switch which is used in the search tool described above. A switch can be used to split one search result into two results according to a condition (i.e. there is a switch which divides the search result according to where the single objects have been found). A switch component would have one input port "search result" but two output ports typed as "search result". Another example could be a counter which can be used to count the found objects. This component has one input port "search result" but is able to consume an infinite number of such tokens. Thus the algorithm needs information on how many tokens of what type are in each component at a certain time. This can be done by integrating counters for each type of token into every single instantiated component.

Marking consumers as "required": As described above there are producers which have required ports and possible consumers. Consequently, another important feature is to provide required consumers. These are components which require a certain input to work properly. For instance, as one can see in Fig. 3 the search engine needs at least a signal of a button which triggers the search query.

The concept so far only takes into account required output ports. One way to integrate required input ports is to extend the domain of the token counter with negative numbers. Therefore a required input port had a count of "-1" which means that there must be a token added to get the valid value "0". Furthermore, it could be useful to transmit required inputs as well as required outputs.

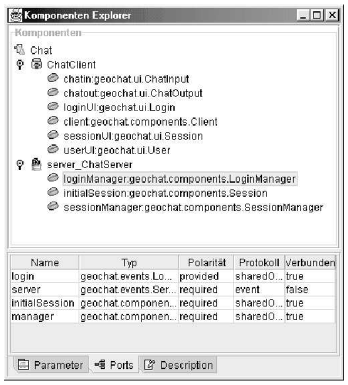
Circuit connections: Finally we have to face the problem that circles can occur if we connect components. Here tokens are passed all along and the would not terminate. Therefore we have to check whether there are circles in our component configuration which can not be solved in the way that the tokens are finally consumed by one of the components within the circle.

5.3 Integrity Information and Checking

The needed information can be described within an component (as additional information, i.e. in JAVABEANS [5] the BEANINFO can provide for the needed information) or explicitly. The second approach allows for more flexibility and takes into account that one component can be used in different contexts and then should take that special environment (domain-specific) into account when defining its integrity information. Changes then can be made easily by the system administrator of an organization or an application developer. The checking algorithm is based on a petri-net-based workflow analysis.

6 Integration into the FREEVOLVE Platform and Visualization

Basically, the FREEVOLVE system provides for an powerful API that allows easy integration of different visual tailoring environments. A new developed one supports different views at the composition (i.e. WYSIWIG, components and their connections, tree view on components, etc.). All views are synchronized with each other. It is designed to support experienced users or administrators (but not especially programmers) when tailoring. Figure 5 shows a screen shot of the first prototype of the tailoring environment. Here all the information about one component are illustrated.



```
Chat
  ChatClient
      chatin:geochat.ui.ChatInput
      chatout:geochat.ui.ChatOutput
      loginUI:geochat.ui.Login
      client:geochat.components.Client
      sessionUI:geochat.ui.Session
      userUI:geochat.ui.User
  server_ChatServer
      loginManager:geochat.components.LoginManager
      initialSession:geochat.components.Session
      sessionManager:geochat.components.SessionManager
```

Name	Typ	Polarität	Protokoll	Verbunden
login	geochat.events.Lo...	provided	sharedO...	true
server	geochat.events.Ser...	required	event	false
initialSession	geochat.componen...	required	sharedO...	true
manager	geochat.componen...	required	sharedO...	true

Fig. 5. Watching the interfaces of one component

The tailoring environment itself provides for an integrity visualization API. It allows to highlight components or pop up messages. The integrity checking module can be integrated as there are interfaces according to the strategy pattern. This allows for flexible redesign of the integrity concept and implementation.

The visualization hints or messages of the integrity checker are based on the idea that the integrity check should mediate between the user's tailoring action and the tailoring environment. Thus every tailoring action and the current composition have to be checked. Eventually (dependent on the integrity strategy's policy) messages are displayed by marking the component or connection which causes an error or displaying help texts (which should be clearly assigned to the defective part of the composition).

7 Conclusions

Flexible component-based applications allows for powerful tailoring possibilities. If tailoring is done by users themselves they have to be supported by appropriate tailoring environments. Mostly this is done by GUIs (i.e. integrated development environments) or by the use of visual languages. Several additional supporting concepts have been developed.

In this paper we present the concept of integrity checking. Here a composition and additional integrity information - belonging to single components or to an application template - can be checked. A tailoring environment which integrates the integrity checking unit helps to test the application and visualizes failures or gives hints how the application can be improved according to the integrity information.

References

1. Birngruber, D.: "A Software Composition Language and Its Implementation", in Perspectives of System Informatics (PSI 2001), vol. LNCS 2244, D. B. Bjorner, M.; Zamulin, A. V., Ed. Novosibirsk, Russland: Springer, 2001, pp. 519-529.
2. Blom, M.: "Semantic Integrity in Program Development", in Department of Computer Science: Karlstad University, 1997.
3. Frese, M., Irmer, C., and Prümper, J.: "Das Konzept Fehlermanagement: Eine Strategie des Umgangs mit Handlungsfehlern in der Mensch-Computer-Interaktion", in Software für die Arbeit von morgen, M. Frese et. al., Ed. Berlin: Springer Verlag, 1991, pp. 241-252.
4. Griffiths, A.: "Introducing JUnit", 2001, http://www.octopull.demon.co.uk/java/Introducing_JUnit.html.
5. JavaSoft: "JavaBeans 1.0 API Specification". Mountain View, California: SUN Microsystems, 1997.
6. Mackay, W. E.: "Users and customizable Software: A Co-Adaptive Phenomenon", Boston (MA): MIT, 1990.
7. Meyer, B.: "Eifel: A Language and Environment for Software Engeneering", Journal of Systems and Software, 1988.
8. Oppermann, R. and Simm, H.: "Adaptability: User-Initiated Individualization", in Adaptive User Support - Ergonomic Design of Manually and Automatically Adaptable Software, R. Oppermann, Ed. Hillsdale, New Jersey: Lawrence Erlbaum Ass, 1994.
9. Paul, H.: "Exploratives Agieren: Ein Beitrag zur ergonmischen Gestaltung interaktiver Systeme", in Europäische Hochschulschriften, Reihe 41: Informatik, vol. 16, P. Lang, Ed. Frankfurt am Main, 1995.
10. Plásil, F. V., S.; Besta, M.: "Behavior Protocols", Dep. of SW Engineering, Charles University, Prague, Technical Report, No: 2000/7, August 2000
11. Reussner, R. H.: "The use of parameterised contracts for architecting systems with software components", in: Proceedings of Sixth International Workshop on Component-Oriented Programming (WCOP'01), 2001.
12. Silberschatz, A., Korth, H., and Sudarshan, S.: "Database System Concepts", Osborne McGraw-Hill, 2001.
13. Stiemerling, O.: "The Evolve Project", in Institute for Computer Science III, University of Bonn, 2000.
14. Szyperski, C.: "Component Software - Beyond object-orientated programming". New York: ACM Press, 1998.
15. Won, M.: "Komponentenbasierte Anpassbarkeit - Anwendung auf ein Suchtool für Groupware", in Institute for Computer Science III, University of Bonn, 1998.
16. Won, M.: "Checking integrity of component-based architectures", in: Proceedings of CSCW 2000, Workshop on Component-Based Groupware, Philadelphia, 2000.
17. Wulf, V.: "Let's see your Search-Tool! - Collaborative use of Tailored Artifacts in Groupware", in: Proceedings of GROUP '99, ACM-Press, pp. 50-60, 199