Product Configuration View to Software Product Families

Tomi Männistö[1], Timo Soininen, Reijo Sulonen
Helsinki University of Technology
Software Business and Engineering Institute (SoberIT)
P.O. Box 9600
FIN-02015 HUT, Finland
+358 9 4511
Tomi.Mannisto@hut.fi, Timo.Soininen@hut.fi, Reijo.Sulonen@hut.fi

ABSTRACT
Development and management of software product families is an emerging and important area of software engineering, whereas product configuration of traditional, i.e., mechanical and electronic, product families has a slightly longer history as a specific area of research and business. This paper describes work in progress on the management of configurable software product families. Solutions to modelling and managing such software products are sought from experiences with traditional products.

Keywords
software product family, configuration, evolution

1 INTRODUCTION

Product configuration and product data management have been successfully applied to traditional, i.e., mechanical and electronic, product families both in research and practice. In software engineering, product families have also become an important area of research as the number of variants of a software product has increased because of various requirements from different markets, hardware platforms, customer specific individualisation, and so on.

The goal of the research outlined in this paper is to manage software product families on the basis of a compact model that describes the variants of the family. In this, we will utilise the lessons learned from modelling traditional configurable products.

Software product families with a large number of variants resemble traditional configurable products in many respects. Therefore, we believe that it is worth applying the results from the area of traditional products to that of software product families. Although the transfer of results from software research to the field of traditional products is also of importance, we do not address it here.

For both traditional configurable products and software product families, it is important to be able to design new variants for a product family in order to meet new emerging requirements. On the other hand, one must also manage the large set of variants and be able to select and produce a correct individual for a particular need from the set of all variants.

This paper focuses on the latter problem. This viewpoint means viewing a software product as a configurable product family that potentially includes a very large number of variants. Such a software engineering paradigm becomes relevant, for example, when software is embedded in a configurable product and the software must adapt to the hardware configuration. If the available memory is limited, the loaded software cannot simply include all possible variability and dynamically adapt to the hardware. Examples of such products are tomorrow's mobile terminals. Similar strategies are also currently sought for enterprise resource planning (ERP) systems that often are large configurable information system packages [11].

We are not primarily interested here in the process for designing a product family or a new variant for a product family — we assume them to be separate problems from the modelling of the software product family. In addition, we do not discuss here other means of software adaptation or adaptability, such as support for customisation by end users, software agents or self-configurable software. However, we believe that a configuration model and the concepts discussed in this paper may also be used for modelling the variability in some of those areas.

In the following, we first describe the field of product configuration of traditional products. Thereafter, we discuss modelling software product families with ideas and solutions imported from traditional products. Finally, we

[1]Visiting Nokia Research Center, Burlington, MA, USA

give some conclusions on the possibilities of importing results from one area to the other and outline further work in this direction.

2 CONFIGURABLE TRADITIONAL PRODUCTS

A *configurable product*, or a *product family*, is such that each *product individual* is adapted to the *requirements* of a particular customer order on the basis of a predefined *configuration model*, which describes the set of legal *product variants* [18,20]. Because of the combinatorial explosion, the number of legal variants of a traditional configurable product is typically large enough, on the order of thousands or more, so that listing them one by one is infeasible. A specification of a product individual, i.e., a *configuration*, is produced based on the configuration model and particular customer requirements in a *configuration task*. The configuration task is assumed to be routine, that is, the generation of the product individual does not involve creative design or design of new components.

Configurable products make a clear separation between the process of designing a product family and the process of generating a product individual according to the product configuration model. This places configurable products in between mass-products and one-of-a-kind products by enabling customer specific adaptation without losing all the economical benefits of mass-products.

Knowledge based systems for configuration tasks, product *configurators*, have recently become an important application of artificial intelligence techniques for companies selling products adapted to customer needs [4]. The purpose of a configurator is to allow managing the configuration models and support the configuration task. Product configuration tasks and configurators have been investigated for at least two decades [14]. Several approaches have defined specific configuration domain oriented conceptual foundations. These include the three main conceptualisations of configuration knowledge as resource balancing [8], product structure [3] and connections within a product [15,21]. Next, we first give an overview to a combined conceptualisation of configuration knowledge synthesising these approaches, which we have reported in detail in [20,21]. Then we briefly review an approach to modelling evolution of traditional configurable products, which we have reported in [12,13]. The latter work builds on the ideas of supporting design object versioning in the field of product design [10].

Configuration modelling concepts

We believe that structured modelling is essential in keeping product configuration models understandable. Accordingly, the models discussed in this section are based on explicit description of structural information in an object-oriented manner [13,21]. We present the concepts for configuration modelling by means of slightly extended UML (Unified Modeling Language). The details of the representation are not primarily important here, as we are more interested in the appropriate product family modelling concepts.

The modelling is based on *component types* that form an *is-a hierarchy* (for simplicity, only with single inheritance). Component types may have *property definitions*, such as part, port and resource use and production definitions. Next, these main concepts and their intuitive semantics are very briefly introduced with some examples shown in Figure 1, which shows an imaginary PC product family.

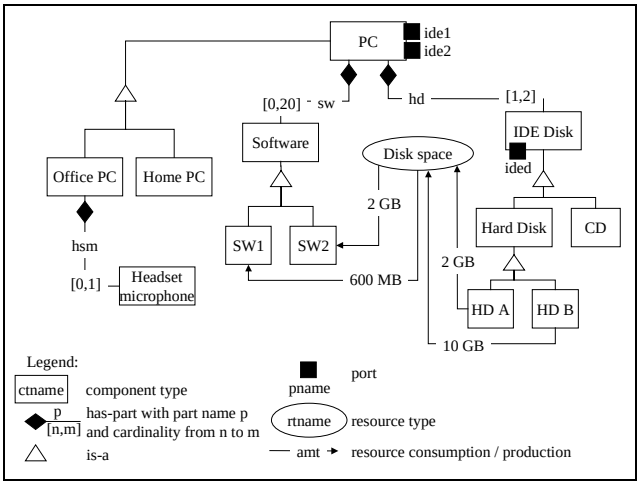

Figure 1. Illustration of configuration concepts by a simplified PC.

Structural composition is modelled by means of *part definitions*. A part definition is augmented with *part name* and *cardinality*, e.g., a PC may have one or two IDE Disks as parts named 'hd'. Zero minimal cardinality represents an *optional part*. In addition, *alternative parts* may be defined. This is not show in the figure, but, for example, if there were more hard disk types, only some of them might be defined as alternatives for a home PC.

Connections between components can be modelled by defining *ports* (representing *interfaces*) for component types, with the idea that in a product individual each port may be *connected* to a port of another component individual. In the figure, there are sample ports for connecting IDE controllers of a PC and IDE disks (for simplicity, *port types* are not shown in the figure).

A configuration model may define *resources*, and component types may define that their individuals *produce* or *use* resources. For example, in the figure hard disks produce disk space, whereas software uses it. In a legal configuration, the production and use of resources should be *satisfied*, e.g., there should be enough disk space to accommodate all software. In addition, a *context* may be defined in which a resource must be satisfied, e.g., the use of a resource may need to be satisfied by production within the same subsystem.

In the is-a hierarchy, the properties of component types are

inherited. When "PC defines that it has part Disk", this means that also its *subtypes* 'Office PC' and 'Home PC' have 'Disk' as part.

The is-a hierarchy also induces variability. That is, when a component type is defined to be a part, this means that any of its subtypes can be a part. E.g., either of the subtypes of 'Disk' is a valid choice as (type of) a part of a PC.

Configuration modelling languages also typically have a mechanism for expressing *conditions* or *constraints* on combining component types, such as *incompatibility* and *requires*. That is, if two component types are incompatible, individuals of both types cannot occur in a legal configuration. When a component type requires another component type, a configuration that contains an individual of the requiring type is legal only if it also contains individual of the required type.

In addition to what is shown in the figure, the configuration model typically also includes some means for expressing the *functions* (or *features*), as seen by the customers. These allow the customer to express his or her interest in more convenient terms than, for example, by selecting particular component types.

Evolution and component internal variation

The long-term management of configuration knowledge has always been a major problem for product configurators. We next briefly describe the concepts for an approach to modelling evolution of product families. A *version* is a concept for capturing the state of a *generic object* at a particular time and the evolution of a generic object is captured by a set of versions [10]. In order to be versions of the same generic object, the versions share something in common, such as an interface or substitutability in use, e.g., so that newer versions can be used in place of older versions. In a configuration model, component types are examples of generic objects.

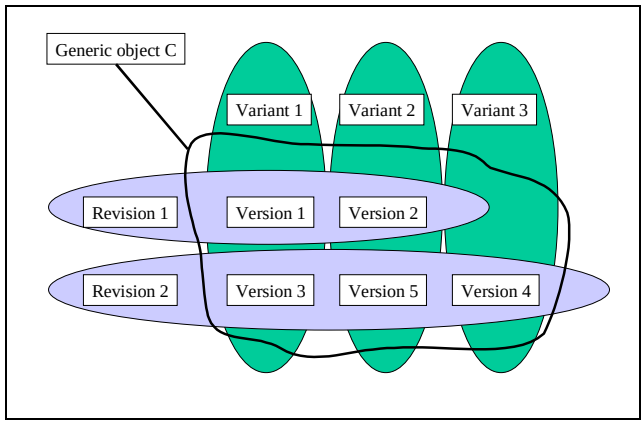

Figure 2. Generic object with versions, variants and revisions.

When variability is also considered, the concept version is divided into concepts *variant* and *revision*, i.e., parallel and consecutive versions of the generic object, respectively. Variants represent coexisting versions, whereas revisions capture the evolution in time.

Concepts 'variant' and 'revision' organise the set of versions, as illustrated in Figure 2. For example, Variant 1 has its own revision history, and the explicit representation of revisions bears the notion that Version 3 and Version 5 are in some sense the corresponding revisions of Variant 1 and Variant 2, respectively. They may, for example, have the same major error fixed or implement the same main functionality.

In addition, there is an ordering between revisions. For example, Revision 2 *succeeds* Revision 1, which means that in Variant 2, Version 5 succeeds Version 2. This ordering of revisions is linear since variants are represented separately.

The generic object also serves as a point of reference to the set of versions. Such *generic references* do not specify the version; it is left open to be bound at an appropriate time. The version binding mechanism enables selecting both the appropriate variant and the revision of a generic component. For example, the selection could be based on a label, such as '1.1.2', a time point or defaults, e.g., the "current revision of default variant".

Variants of a generic object implement what we call *internal variation*, i.e., variation according to which variant of a selected component type is in the configuration. Another form of internal variation is *parameters* of a component type, such as the length of an axis or colour, for which the configuration task must determine appropriate values. This is in contrast to *structural variation*, which captures which component individuals may be selected in a configuration, as discussed in Section 2 in relation to Figure 1.

With traditional products, the revisions typically represent the internal evolution of a generic object, whereas modifications that are visible externally, e.g., changes to the interface, necessitate creation of a new generic object instead of a new revision.

One major issue in supporting evolution with generic objects is how a change, i.e., creation of a new revision, in one component type should affect the other component types. Should its subclasses be affected and what about the wholes in which the component type occurs as a part? That is, how revisions of different entity types may and may not be related. These issues are discussed in more detail in [12] and omitted here for brevity.

Lessons Learned

A lesson learned from traditional configurable products is the essential role of long term management in the success or failure of methods and systems supporting product configuration. One of the earliest examples, the XCON

configurator, was reported to provide work for tens of developers and maintainers of configuration knowledge after a few years of operation [1]. Therefore, we feel that it is also important to address the issue of evolution in modelling of product families.

In order to succeed, one should first be able to describe the variants of a product family in a clear and intuitive manner. With traditional configurable products ad hoc means in modelling the product variety lead to complex models that were difficult to maintain. In addition, mapping the product family to some description language easily destroyed the inherent design structure of the product family. Consequently, such representational mismatch made it difficult to propagate product changes to the product family representation. Furthermore, the description language was typically such that the product experts did not understand it, which required the use of modelling specialists who did not understand the product.

The concepts for modelling product variety of traditional products are generic in the sense that they do not reflect any particular domain area, such as elevators, telecommunication switches, personal computers, etc. In addition, it has been adequate to model the variation of product families at a rather high level of abstraction without kinematics or stress analyses, for instance. Furthermore, the configuration modelling has concentrated on representing the existing variety of product families. Component descriptions are not so generic that one could use the configuration knowledge for discovering new configurations or unforeseen uses of components.

Finding the correct concepts for representing the product families is only the first step in making the management of product families succeed. In addition, the evolution of components and other entities used in the representation should be managed. To achieve this, it is not adequate to only label the versions in separation; one should also capture how the different versions of different entities relate to each other.

3 CONCEPTS FOR CONFIGURABLE SOFTWARE

There are many ways of achieving variety in software. Software variety can be implemented by a 'universal' software product that is a union of the all variants, and may thus behave as any specific variant. An alternative approach is to use preprocessor directives to optionally include pieces of source code. This approach however, easily loses the big picture over the entire product family, as the representation of variation is distributed in the source code. With large software products, a typical adaptation approach is to take an existing variant as a basis and then modify it accordingly [9]. Such "copy, paste and modify" of architectural components easily blurs the original ideas behind the architecture and in consequence deteriorates the overall product architecture [5].

Variability may also be achieved by selecting appropriate components to a family architecture [9,23,24]. In the case of software product families, the architecture can thus be called a *product family software architecture* (PFSA). In fact, there are many different phases in the life cycle of a product individual (to distinguish from the life cycle of product family), e.g., construction of configuration, build and execution, for which different concepts are needed for modelling the product family [16].

We will next discuss the suitability of configuration modelling concepts from the previous section for modelling software product families. We will compare the previous work on software both to the configuration modelling concepts and to our approach.

Functions or Features

Customers view a product differently than the engineers designing it; customers prefer speaking of features that are of value to them, not necessarily about technical details of implementing the features. This means that a configuration model should also capture these features so that customers can utilise them while inputting system requirements.

As there are approaches for modelling software product families on the basis of features [7], it seems that from a modelling perspective, features of software seem to correspond to functions of traditional products.

Our approach differs from most software product line approaches as we put more emphasis on routine creation of individuals for particular customer requirements and separate that process from the evolution processes. As a difference to feature based-approaches, we build the modelling of variety on more structural concepts and consider the customer requirements expressed by features as entities that are mapped on the configuration modelling concepts, e.g., to selections of component types or additional conditions for the legal configurations.

Compositional structure and taxonomy

Compositional structure and taxonomy are very powerful and suitable means for representing configurable products. The decomposition of a system into subsystems is also important for the management of software product families and their configuration knowledge.

For example, work at Philips [24] defines a skeletal architecture that has plug-in components and optional implementations of components (called units in the paper). This provides similar, although simpler, variation for a software product family as optional and alternative parts in a configuration model. Van der Hoek et al. [22] propose a model that has great similarities with some of the concepts discussed above. The optionality proposed in their approach takes the form of structural variety according to the terminology above. The variability of components, on the other hand, is explicitly represented by special variant components, which consist of a set of components with

same interface. For a configuration, one variant is selected from the set of variants of the component according to some control variables. This corresponds to the internal variation of a component, as discussed above.

Our approach uses the organisation of component types in a class hierarchy with structural decomposition, component subtyping and inheritance as a primary basis for modelling product variety. This differs from typical approaches in software product lines that build on connection topology.

Connections and Resources

There are many possibilities in conceptualising connections. The basic idea is to get components connected, which requires a point of connection in components. These are called ports in configuration modelling, similarly as in software architecture domain [see, e.g., 19]. Another issue is whether separate entities are used to represent connections between the components as in architecture description languages or if components and their connections similar to configuration modelling suffice. When separate connectors are used, they also need points of connection, called *roles*.

One approach for expressing the configuration knowledge on the basis of interfaces and their correct connection is that of Koala [23]. The approach makes a distinction between internal variety of a component (they call it internal diversity of a component) and structural variety. For internal variation Koala offers diversity interfaces, which essentially describe the parameters by which the variant of the component (type) can be determined. For structural variety it offers switches that model alternative connections between interfaces.

It seems that the concepts proposed for modelling software architecture and interfaces of software classes or components are rather similar to the concepts proposed for traditional products and it should be easy to utilise results from the latter are to model and manage software product families.

Furthermore, in certain cases, such as method calls to components, it may be adequate to represent the interfaces as resources and only check that all needs are satisfied without making the explicit connections. Module Interconnection Languages (MIL) use resources similar to those described above for matching modules by interfaces of services they provide and need [see, e.g., 19]. In configuration modelling, the satisfaction of resources, however, is typically more complex than simple matching. There may be multiple providers and users of the same resource and thus the amounts produced and used must be calculated, possibly within a context.

Our approach does not initially include separate concepts for modelling connections. Connections can be modelled, at least to some extent, as component types because of their conceptual similarities. However, we are not sure whether that would be an appropriate simplification for software. In fact, modelling of connections is one potential area in which results could be transferred from software architectures to configuration modelling of traditional products. Furthermore, our approach provides a clean way of combining resources and compositional structure, which seems a new and interesting way of representing software product families.

Constraints

Additional constraints describe the conditions on legal configurations that cannot be expressed by other concepts.

For software, additional constraint could specify dynamic behaviour. However, constructing detailed behavioural models is a remarkable effort and may require formalisms that make them computationally infeasible. In configuration modelling of traditional products, the computational complexity has been studied and a working balance between expressive power can be established—e.g., it is not necessary to model the behaviour of the products for configuration and product data management purposes [20].

Our approach is to focus on modelling variety, and thus leave out constructs for modelling detailed behaviour of the software. We intend not to use constraints for directly modelling the variety but as additional conditions on the legal configurations, e.g., for pruning out configurations including component individuals of incompatible component types. However, we plan to provide a formal semantics for the modelling concepts by mapping them to some logic language for validation and generation of configurations, but that language would not be the primary means for representing the configuration model by and to humans. This scheme has been carried out for the concepts described in Section 2 [20].

Evolution

The evolution in Koala is defined to protect the stability of interfaces, which corresponds well with the semantics the generic objects are meant to capture [23]. That is, the new versions may be created for the component as long as the common part described in the generic object, e.g., the interface, remains.

In the approach by van der Hoek et al. [22], variability and revisioning are separated in a manner that is semantically equivalent to generic object of Figure 2. Our proposal is in this respect a superset of their approach, and allows, e.g., the variability to be represented as structural or internal.

Other research in software configuration management has also addressed the orthogonal nature of revisions and variants. An *n*-dimensional grid is one way of representing variants of software [2]. Estublier and Casallas [6] identified the dimensions: historical (i.e., revisions), logical (i.e., variants) and co-operative (i.e., concurrent work intended to be merged) for the version space. VOODOO

system, on the other hand, models versions by a cube that has the dimensions: component, revision and variant [17].

We follow a similar orthogonality principle, but propose a different model for capturing the evolution of software configuration models based on uniform use of generic objects. This makes explicit distinction between evolution and variation of single design objects. In addition, we do not propose composition as one dimension comparable to revisions and variants—compositional structure is modelled by part definitions, which may utilise generic references with a version binding mechanism. Generic objects may be used in modelling the compositional structure of software product families as well as the evolution with respect to taxonomy. However, modelling the evolution of product families still requires further research [12].

4 CONLUSIONS

There are remarkable similarities between the concepts proposed for software architectures or software product families and those used for modelling configurable traditional products. For traditional products, various methods exist that allow modelling product families by essentially using some subset of the concepts of Figure 1, with some approach-specific variations. These concepts for modelling the variety of traditional products seem to suit modelling software product families as well. We do, thus, propose a model-based approach as a means for managing software product families when the number of variants increases. Utilising the concepts from traditional product families and adapting them for representing the architecture and variation of software product families, we believe, would lead to more concise and manageable models. In addition, such models would open a way to using the AI methods developed in the field of product configuration to support the generation of product variants on the basis of the models.

We see the future work in modelling software product families to include the following issues.

In traditional products, kinematics and stress analyses, for example, are abstracted away from configuration models. How about software—What is the appropriate level of abstraction of dynamic behaviour for software product families? Does this change if configuration modelling is extended to capture the dynamic re-configuration of software in the allocation/execution view?

In addition, the conceptualisation of connections requires investigating whether the more complex concepts of components and connectors are needed or if a simpler conceptualisation based on components and their connections through ports is adequate.

Another issue is to consider whether the integration of compositional structure with taxonomy and resources would provide practically feasible product family modelling tools for software engineers and architects.

The incorporation of evolution to product family modelling is also of great importance. However, even though there is much work in modelling evolution in various areas, including design data modelling, product data management, software configuration management, schema evolution of databases and temporal databases, there is still plenty to be done before a mature practice for capturing the evolution of software product families can be defined.

Last but not least, we need to model some real software product families on the basis of the concepts described in this paper. Such experiences are essential in developing the concepts further and in validating their practical feasibility.

ACKNOWLEDGEMENTS

We gratefully acknowledge the financial support from the Academy of Finland (grant 51394). We also thank Alexander Ran for his valuable comments on an earlier draft of the paper.

REFERENCES

1. Barker, V.E., O'Connor, D.E., Expert systems for configuration at Digital: XCON and beyond, *CACM, 32*, 3 (1989), 298-318.
2. Conradi, R., Westfechtel, B., Towards a Uniform Version Model for Software Management, *SCM97, LNCS 1235* (1997), Springer, 1-17.
3. Cunis, R., Günter, A., Syska, I., Peters, H., Bode, H., PLAKON — An approach to domain-independent construction, *IEA/AIE-89* (1989), 866-874.
4. Darr, T., McGuinness, D., Klein, M., Special Issue on Configuration Design. *AI EDAM 12*, (1998).
5. Dikel, D., Kane, D., Ornburn, S., Loftus, W., Wilson, J., Applying software product-line architecture, *Computer, 30*, 8 (1997), 49-61.
6. Estublier, J., Casallas, R., Three dimensional versioning, *ICSE SCM-4 and SCM-5 LNCS 1005*, Estublier, J., ed. (1995), Springer-Verlag, 118-135.
7. Hein, A., Schlick, M., Vinga-Martins, R., Applying feature models in industrial settings, *Software product lines—Experience and research directions*, Donohoe P., ed. (2000), Kluwer Academic Publishers, 47-70.
8. Heinrich, M., Jüngst, W., A resource-based paradigm for the configuring of technical systems from modular components, *IEEE conference on artificial intelligence applications*, (1991), IEEE, 257-264.
9. Karhinen, A., Ran, A., Tallgren, T., Configuring design for reuse, *ICSE'97*, (1997), 701-710.
10. Katz, R.H., Chang, E., Bhateja, R., Version modeling concepts for computer-aided design databases, *SIGMOD*, (1986), 379-386.

11. Kumar, K., van Hillegersberg, J., Enterprise resource planning—experiences and evolution, *CACM, 43,* 4 (2000), 22-26.

12. Männistö, T., A conceptual modelling approach to product families and their evolution. Doctoral thesis. (2000), Helsinki University of Technology.

13. Männistö, T., Peltonen, H., Sulonen, R., View to product configuration knowledge modelling and evolution, *Configuration—papers from the 1996 AAAI Fall Symposium (AAAI technical report FS-96-03)*, Faltings, B., and Freuder, E.C., eds. (1996), 111-118.

14. McDermott, J., R1: a rule-based configurer of computer systems, *Artificial Intelligence, 19,* 1 (1982).

15. Mittal, S., Frayman, F., Towards a generic model of configuration tasks, *IJCAI*, (1989), 1395-1401.

16. Ran, A., ARES conceptual framework for software architecture, *Software Architecture for Product Families*, Jazayeri, M., Ran, A., and van den Linden, F., eds. (2000), Addison Wesley, 1-29.

17. Reichenberger, C., VOODOO A Tool for orthogonal version management, *ICSE SCM-4 and SCM-5 Workshops: LNCS 1005*, Estublier, J., ed. (1995), Springer, 61-79.

18. Sabin, D., Weigel, R., Product configuration Frameworks—A survey, *IEEE intelligent systems & their applications, 13,* 4 (1998), 42-49.

19. Shaw, M., Garlan, D., *Software acrhitecture—Perspectives on an emerging discipline*, Prentice-Hall, 1996.

20. Soininen, T., An approach to knowledge representation and reasoning for product configuration tasks. Doctoral thesis. (2000), Helsinki University of Technology.

21. Soininen, T., Tiihonen, J., Männistö, T., Sulonen, R., Towards a General Ontology of Configuration, *AI EDAM, 12,* 4 (1998), 357-372.

22. van der Hoek, A., Heimbigner, D., Wolf, A.L., Capturing architectural configurability: variants, options, and evolution. (University of Colorado, Dept. of Computer Science, 1999), CU-CS-895-99.

23. van Ommering, R., van den Linden, F., Kramer, J., Magee, J., The Koala component model for consumer electronics software, *Computer,* March (2000), 78-85.

24. Wijnstra, J.G., Supporting diversity with component frameworks as architectural elements, *ICSE00,* (2000), 50-59.