Formal Consistency of Models in Multi-View Modelling

Purandar Bhaduri and R. Venkatesh

TRDDC, Tata Consultancy Services
54 B, Hadapsar Industrial Estate,
Pune 411 013, INDIA.
Email: {pbhaduri,rvenky}@pune.tcs.co.in

Abstract. Meta-model based development offers a promising way of managing the complexity of industrial scale software by describing a system in terms of different 'views'. These views can then be described as instances of a single meta-model. Such views are usually not disjoint and it is essential that they are shown to be consistent. A weakness of meta-modelling tools is the lack of support for describing the behaviour of models, and this is central to demonstrating the consistency of views. We address this problem by combining meta-modelling with formal techniques for stating and verifying behavioural properties. In this paper, we describe a formalization of models and meta-models and show how this leads to automated procedures for consistency checking between views in an industrial software development framework.

1 Introduction

Industrial scale software development relies heavily on the use of models, which help in managing the complexity of problems by separating concerns. Different models often use different description techniques, each suitable for describing a particular aspect of the system. For example, the models used in UML [2] are the *structural views*, e.g. classes, objects and their attributes and relationships, and the *behavioural views*, e.g. sequence and statechart diagrams, depicting inter-object collaboration and intra-object state transitions.

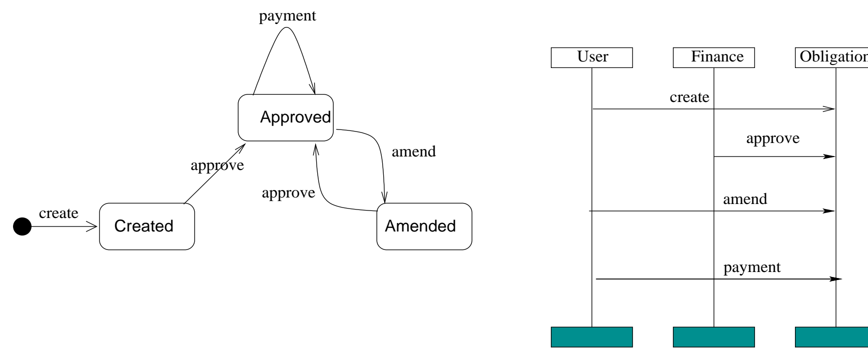
Each of these different models can be treated as an instance of a part of a single *meta-model*. For example, each UML diagram, which is a view of the system, is an instance of the UML meta-model [8].

Different views are usually not disjoint and thus may describe the same property of the system. Common properties must be *consistently defined* to ensure the integrity of the system. For example, two views can be used to describe the dynamic behaviour of a system:

1. A state machine model which describes *all* permissible sequences of events in the life of an object as well as the object's actions in these sequences. This can be considered as a projection of the system traces on a specific object.
2. A Message Sequence Chart which describes one possible time sequence of interactions (send and receive events) among a set of given objects. This describes only a set of system traces between two (usually unspecified) system states restricted to a set of objects.

Example 1. (*Views*) An organization needs a financial and budgeting system for keeping track of expenses for different items. All expenditure must be budgeted for by creating and updating an obligation. A user must first budget expenses through an obligation which must be approved by the finance department. Expenses can be made only against such approved obligations. The amount of an obligation can be amended. All such amendments must be approved afresh.

The above requirements can be interpreted in two possible ways. The first says that no payment can be made against an amended obligation before the amendment is approved, shown by the state machine view in Figure 1. The other is that a payment can be made even if the amendment is approved, provided the amount of the payment was already approved before the amendment. This interpretation is shown by the MSC view in Figure 1



(a) State Machine for an Obligation object (b) An MSC for making a payment

Fig. 1. Two Views of the Budgeting System

In order to deal with the issue of consistency of views, we must have a formal system model as a framework for reasoning about the relationship between the different views. The system model should describe both the static and dynamic aspects of the system. The static aspect comprises the entities in the system along with their relationships at any point of time. This could be taken to be the objects, the values of their attributes and the links between them. The dynamic aspect of the system should capture the transitions of the system from one static view to another. These transitions are caused by operation invocations on objects, which may trigger other invocations in turn.

In this paper we propose a formalization of different models of a system along with the system model to address the problem of compatibility of views. We focus on consistency of dynamic behavioural views only, and ignore issues of static consistency checking. We do not deal with consistency of views with respect to data states, which

will be the subject of a future paper. Here we limit ourselves to the control aspects of dynamic behaviour.

For concreteness, we focus on three different models of a system, comprising the static structure, object interaction and intra-object behaviour views:

1. *Class Diagram* This gives the static view of the system in terms of classes, their attributes and operations, the associations between classes and their properties (e.g. cardinality constraints), invariants (constraints) satisfied by objects of the class, and pre- and post-conditions of operations.
2. *Message Sequence Charts* [6]. This gives the dynamic view of object interaction in terms of exchange of messages (operation calls or signals) to realize a particular functionality.
3. *State Diagrams* This is the dynamic view corresponding to intra-object behaviour in terms of states and transitions of the object in question. Each transition is labelled by a triggering event (invocation of a operation in this object), a guard (a boolean condition) and a resulting action (e.g. a call to a operation in another object), all of which are optional.

These views are not independent as Example 1 shows. The class diagram places constraints on the cardinality of objects and links which any evolution of the system must satisfy as invariants. A class diagram may also impose pre- and post-conditions on the invocation of operations in each object, which must be consistent with the transitions that are enabled in the corresponding states in the state machine. A state diagram for a class specifies the order in which the operations in an object of the class can be invoked and this must conform to the inter-object scenarios presented by a set of MSCs.

Our contribution in this paper is to give a unified model of these notions of views and consistency. We abstract away from concrete views like state machines and MSCs, and instead define a generic notion of a view. In order to do this, we define an abstract model of a system in terms of a labelled transition system with additional structure on the states. The views are projections of this system model which impose certain constraints on the system. Consistency between two views can be checked by using algorithms for intersection of finite automata. These notions can be easily integrated with a model based CASE tool for automatic consistency checking of views.

2 Models and Consistency

To formalize the notions of views and consistency we consider a labelled transition system with additional structure on the states as the system model. A system state is a collection of objects and links between them, where each object is a tuple of object identifier, object state and a set of operations. The event e in a system transition $s \xrightarrow{e} s'$ is either the send or receipt of an asynchronous message or an operation invocation on an object contained in s.

A system trace is a sequence of system states

$$s_0 \xrightarrow{e_0} s_1 \ldots s_i \xrightarrow{e_i} s_{i+1} \ldots$$

such that two successive states are related by a system transition. Note that the state of at most one object is modified by a system transition. We use an interleaving model of concurrency in which a system trace represents a serialization of a given set of events.

A *view* of a system S is a system S' composed of a subset of objects and transitions in S. An example of a system and a possible view is shown in Figure 2, where the view includes objects *Ob1* and *Ob2*, and the transitions labelled by *Ob2.f1* and *Ob1.f3*.

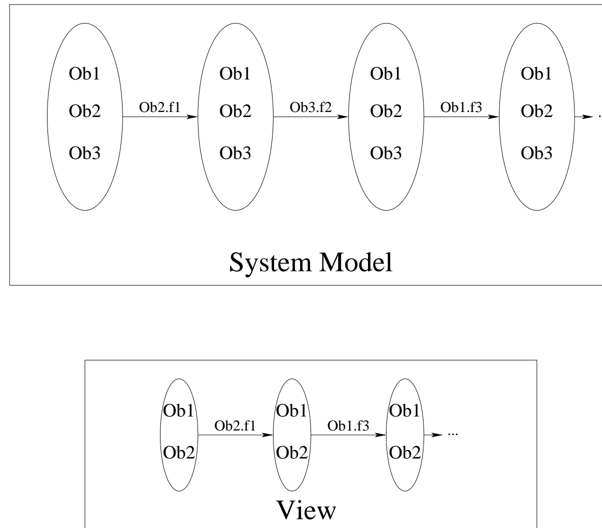


Fig. 2. System Model and Views

In addition to being a subsystem of a system S, a view also places certain constraints on S. These constraints can be expressed as assertions over system states and traces that *may*, *must* and *never* happen in the evolution of the system. For example, the state machine model of an object enumerates the *may* traces (possible sequences), along with the proviso that the list of traces not specified must *never* occur. An MSC model, on the other hand, enumerates a list of *must* traces (mandatory sequences) on a set of objects, while not claiming anything about other traces in the system. We decorate a view with a *mode* which is either *must* or *never* to capture this intuition. Note that we do not need a mode called *may*, because this can be expressed with *must* and *never*. The consistency of two views with different modes can then be expressed as the condition that the *must* and *never* traces do not intersect for the common set of objects. The algorithm for checking consistency is just the familiar one for checking the non-emptiness of the intersection of two finite automata.

Example 2. (*Consistency*) It follows from our formalization of consistency that the views in Figure 1 are not consistent. According to the MSC view the sequence

$$\langle create, approve, amend, payment \rangle$$

must be a valid system trace when restricted to the *Obligation* object. In contrast, the state machine view asserts that the sequence is not a valid trace of *Obligation*.

The notions of view and consistency can be generalized from the specific cases of state machines and MSCs. We can think of a view as a *property* of finite traces of a system. This is tantamount to using a finite state automaton for stating safety properties, with some additional features such as unspecified states and transitions which act as wild cards. Checking a view and a property for consistency then amounts to checking whether the view satisfies the property.

In the rest of the paper, we propose a formalization of these notions of a system model, the different views of a system and their consistency. Our goal is to automate the consistency checking of views in multi-view modelling of object-oriented systems.

Related Work Various approaches to dynamic meta-modelling have been proposed in the literature for defining the behavioural semantics of UML. In [3], dynamic meta-modelling is proposed as a new approach to the operational semantics of behavioural UML diagrams. The system dynamics is specified by means of collaboration diagrams, which are interpreted as graph transformation rules. A similar framework for the formal integration of object-oriented modelling notations using transformation systems has been investigated in [5]. In [4], the authors use Object-Z as the formal meta-language for the definition of UML dynamic semantics. Our dynamic model based on labelled transition systems and the definition of consistency based on intersection of finite traces are conceptually simpler and more amenable to tool support. Our work is closely related to that of [9], where the authors propose a general schema of the semantics of the UML. The different UML diagrams are given individual semantics in a common formal meta-language, which are then composed to get the semantics of the entire model. The formalism used is labelled transition systems with algebraic structure on the states, as supported by the concrete modelling notation CASL-LTL. In this approach, consistency of different UML models is modelled by the consistency i.e., existence of a model, of the resulting algebraic specification. We have taken a simpler approach, as we are primarily interested in automated tool support for analysing consistency.

3 Formalization of System Model, Views and Consistency

In this section we present the technical details of our formal meta-modelling framework. We give formal presentations of the various concepts involved – labelled transitions systems (LTS), system, views, properties expressed by views in terms of traces, *must* and *never* modalities, mapping between views and system model and consistency between views.

3.1 System Model

The consistency of multiple views of a system can be ensured by defining a unified model of the entire system. This model specifies all the possible ways in which the system might evolve over time. It is presented as a labelled transition system with an additional structure on the states.

Definition 1. *A labelled transition system (LTS) is a tuple* $S = \langle States, \longrightarrow, Events, s_0, \mathcal{F}\rangle$*, where* $States$ *is a set of states,* s_0 *is the initial state,* $\mathcal{F}$ *is a set of final states,* $Events$ *is a set of event labels and* $\longrightarrow$ *is the transition relation* $\longrightarrow \subseteq States \times Events \times States$.

Notes

1. An LTS may be *nondeterministic* i.e., two distinct transitions with the same event label may originate at one state.
2. The final states are needed for defining *complete traces* of systems. All consistency conditions and properties are stated in terms of finite traces.
3. An LTS is defined without any structure on the states. In our setting, all the LTS's come equipped with enriched structure defined on the states.

A system is defined as an LTS where the states have internal structure – collection of objects with their local states and links.

Definition 2. *A system is an LTS* $S = \langle States, \longrightarrow, Events, s_0, \mathcal{F}\rangle$*, where a system state* $s \in States$ *is a set of objects* $\{ob \mid ob \in Objects\}$ *and links* $\{\ell \mid \ell \in Links\}$ *between them. Each object is a tuple* $\langle ObjID, ObjState, Operations\rangle$*, of an object identifier, an object (local) state and a set of operations the object offers. At this stage we do not further refine the notion of an object state in terms of values of attributes. The event e in a system transition* $s \xrightarrow{e} s'$ *is either the send or receipt of an asynchronous message or an operation invocation (call event) on an object in* s'*. We assume that in a transition* $s \xrightarrow{e} s'$*, the state of at most one object is modified from* s *to* s'*. All the objects retain their identity and set of operations from* s *to* s'*, unless an object is created or destroyed.*

Example 3. Part of the LTS from the budgeting example above is shown in Figure 3. The state of the *Obligation* object in each state of the LTS is shown in parenthesis at the bottom of the box depicting the object.

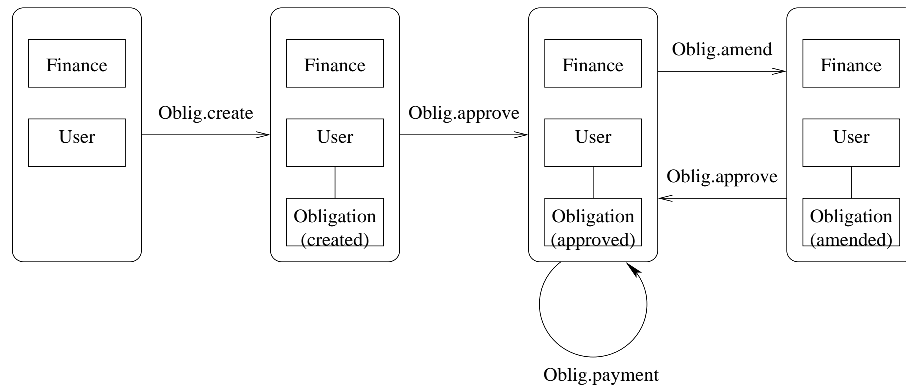


Fig. 3. Labelled Transition System for the Budgeting System

Definition 3. *A system trace is a sequence of system states*

$$t = s_0 \xrightarrow{e_0} s_1 \dots s_i \xrightarrow{e_i} s_{i+1} \dots s_n$$

such that two successive states are related by a system transition and the ending state s_n *is in* $\mathcal{F}$. *By this definition a trace is always finite.*

Note that a trace is a sequence containing both states and transitions. In comparing two traces, we will compare both components. However, in the definition of a view presented below, we do allow a trace to have unspecified states and transitions. When comparing two traces, an unspecified state or transition will match any state or transition.

Definition 4. *The restriction* $t \upharpoonright O$ *of a system trace* t *to a set of objects* O *is obtained by*

1. *removing objects from each state* s_i *of* t *that are not in O,*
2. *erasing the transition labels* e_i *from* $s_i \xrightarrow{e_i} s_{i+1}$ *if* e_i *is not an operation in objects in O, and finally,*
3. *collapsing those states that are related by null transitions into a single state.*

Example 4. In Figure 2, the trace at the bottom is the restriction of the one at the top to the set of objects $\{Ob1, Ob2\}$.

Notes

1. Formally, given a set of objects O, we define an equivalence relation on system states as follows: $s \simeq s'$ if $s \upharpoonright O = s' \upharpoonright O$, where $s \upharpoonright O$ is the system state obtained by removing those objects and their corresponding links from the system state s that are not in the set O. The relation can be read as "s and s' look the same modulo the set of objects in O".
2. The collapsing of states and transitions corresponds to identifying system states by taking the equivalence classes modulo $\simeq$, and deleting the null transitions.

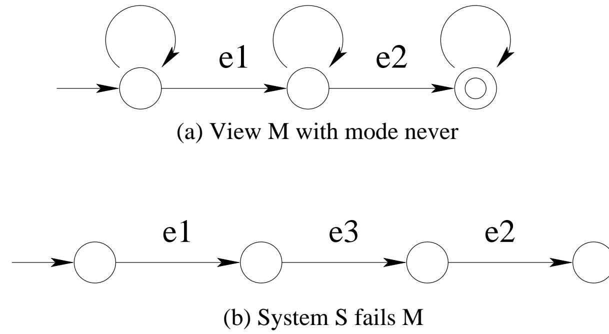
3.2 Views

A view of a system S is an LTS composed of a subset of objects and transitions of S. The transitions that do not affect the objects in a view do not appear in the view. In addition, a view defines constraints on the behaviour of the objects contained in the view. These constraints are expressed as assertions over system states and traces that *must* or *never* happen in the evolution of the system. For example, the state machine model of an object enumerates the possible traces of the object, along with the proviso that the list of traces not specified must *never* occur. An MSC model, on the other hand, enumerates a list of *must* traces (mandatory sequences) on a set of objects, while not claiming anything about other traces in the system. The consistency of two different views follows from the consistency of the assertions over the system contained in these views.

Definition 5. *A view* M *is a pair* $\langle L, mode \rangle$, *where* L *is a finite-state LTS and mode is one of the two modalities must and never.*

Notes A view can have unspecified object states, transition labels and links. The meaning of an unspecified event, state or link is that it will match any event, state or link in comparing two traces.

Example 5. Figure 4 is an example of (a) a *never* view M, that asserts that the event $e2$ will never follow $e1$ on the given object and (b) a system S that fails the view. In this example, the final state is a dead state that captures all forbidden behaviours. Note that in the view the state in the middle can be instantiated to more than one state in the system, since it is unspecified.



(a) View M with mode never

(b) System S fails M

Fig. 4. Example of a *never* view

Definition 6. *A view M is said to be a view of the system S when one of the following conditions hold:*

1. *if M has mode must, then for every trace t of M, there exists a trace t' of S, which when restricted to the objects in M is t, i.e., $t' \upharpoonright objects(M) = t$;*
2. *if M has mode never, then for every trace t of M, there does not exist a trace t' of S, which when restricted to the objects in M is t.*

3.3 Consistency between Views

Definition 7. *Two views $M_1 = \langle L_1, mode_1 \rangle$ and $M_2 = \langle L_2, mode_2 \rangle$ are consistent if*

1. *$mode_1 = mode_2$, or*
2. *(when $mode_1 \neq mode_2$) the traces of M_1 and M_2 when restricted to their common objects do not intersect, i.e., $traces(M_1) \upharpoonright O \cap traces(M_2) \upharpoonright O = \emptyset$, where $O = objects(M_1) \cap objects(M_2)$.*

Notes

1. The consistency between two views is with respect to *safety* properties, i.e., properties whose violation can be detected by considering finite traces.
2. The consistency checking algorithm reduces to checking that the intersection of two FSMs is non-empty.

4 State Machines, MSCs and Properties as Views

In Section 3 we defined a view of a system and consistency of two views at an abstract level. In this section, we provide translations from object state machines and MSCs to the abstract notion of a view. Further, we show how the properties involved in the notion of consistency between views can themselves be captured as views. To check that a system satisfies a property then reduces to checking that the property is a view of the system, in the sense of definition 6. To check that a state machine (respectively MSC) satisfies a property reduces to checking the consistency between the view and the state machine (respectively MSC) in the sense of definition 7.

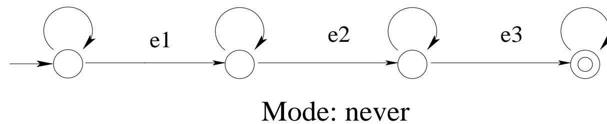
State Machines as Views: Translating object state machines to views is straightforward. A state machine defines *all* possible transitions of an object. No other transition should occur in the system on this object. Thus a state machine presents a *never* view of the system, where the view is obtained from the state machine by adding a dead state as the only final state, and adding all the missing transitions from every state to the dead state. Figure 4 is an example of a state machine treated as a view.

MSCs as Views: An MSC defines an example scenario that must be exhibited by the system. Recall that an MSC defines a partial ordering on the events in a system, where the events are sending and receiving of messages (see [1] for details). The ordering is defined by the ordering of events along a timeline and the fact that a send event precedes the corresponding receive. From this partial order on the events, one obtains a finite LTS in a standard way. The downward closed subsets of the partial order, referred to as *cuts*, represent states of the LTS. There is a transition $s \xrightarrow{e} s'$ if $s \cup e = s'$. We declare all states as final states. Alternatively, if an MSC is used to present a forbidden scenario, then it is a *never* view, where the state containing all the events is the only final state.

Note that in general, a high-level MSC (HMSC) cannot be translated to a finite LTS. For *bounded* HMSCs, a finite state LTS can be obtained, and checking for consistency with a state machine view reduces to model checking of bounded HMSCs, which is co-NP complete (see [1]).

Properties as Views: We have seen how object state machines and MSCs can be translated to views. We can generalize this situation to properties on finite traces, which subsumes both these cases. This way we can specify *safety* properties, properties whose violation can be detected by inspecting all finite traces. We give examples to illustrate how simple safety properties can be expressed using views.

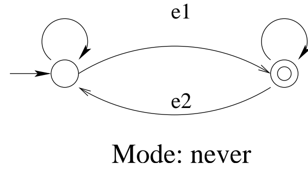
Example 6. Consider the property, "The subsequence of events e_1, e_2, e_3 never occurs in a trace." This is depicted below as a *never* view. Note that the unlabelled transitions



Mode: never

in the view match any transition in the system.

Example 7. "An event e_1 always leads to an event e_2 in a trace." This is again a *never* view, as shown below.



Mode: never

To check whether a view satisfies a property amounts to checking the consistency of the view with the property. We have already seen that this requires checking the emptiness of the intersection of two finite automata.

5 Conclusion

In this paper we have proposed a formal notion of consistency of models commonly used in model based software development. Our underlying semantic framework is labelled transition systems with additional structure on the states to capture the data aspects of the models. The novelty of our approach lies in viewing each model as an LTS with a *must* or *never* modality, that places constraints on the global system model. Consistency checking between models for purely control aspects of behaviour is simply checking non-emptiness of the intersection of two finite automata. We are exploring ways to integrate this notion of consistency and its automated verification in an industrial meta-model based development tool.

The examples presented here all involve determining the consistency of purely control aspects of behaviour described by different models. When the models also involve properties of the data state of the system, such as invariants and pre- and post-conditions for operations, we will need a logical framework for reasoning about consistency. We are currently investigating Lamport's Temporal Logic of Actions (TLA) [7] as a suitable formalism for describing and proving properties of transition systems with both data and control elements. The advantage of using TLA is that it provides a single logical formalism for describing transition systems and formulating their properties, which include both data and control aspects.

Our formalization of the notion of consistency, and its verification in the context of multiple model based development, should be seen as a first step towards an integrated behaviour modelling and analysis framework. This would integrate both structural and behavioural models and aid in forward engineering –refinement, code generation and test case generation, in addition to detecting faults early in the life cycle.

6 Acknowledgements

We thank Prof. Mathai Joseph for his guidance and encouragement throughout the project. Preliminary ideas of this work were presented at an IFIP WG2.3 School on Formal Software Engineering at Pune in January 2002. We thank the participants of the school for their valuable feedback.

References

1. R. Alur and M. Yannakakis. Model Checking of Message Sequence Charts. In *Proceedings of the Tenth International Conference on Concurrency Theory, CONCUR'99*, Lecture Notes in Computer Science. Springer-Verlag, 1999.
2. G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, 1998.
3. G. Engels, J. H. Hausmann, R. Heckel, and S. Sauer. Dynamic meta modeling: A graphical approach to the operational semantics of behavioral diagrams in UML. In A. Evans, S. Kent, and B. Selic, editors, *UML 2000 - The Unified Modeling Language. Advancing the Standard. Third International Conference, York, UK, October 2000, Proceedings*, volume 1939 of *LNCS*, pages 323–337. Springer, 2000.
4. R. Geisler. Precise UML semantics through formal metamodeling. In L. Andrade, A. Moreira, A. Deshpande, and S. Kent, editors, *Proceedings of the OOPSLA'98 Workshop on Formalizing UML. Why? How?*, 1998.
5. M. Große-Rhode. Integrating semantics for object-oriented system models. In F. Orejas, P. G. Spirakis, and J. van Leeuwen, editors, *Proceedings ICALP 2001*, number 2076 in LNCS, pages 40–60. Springer-Verlag, 2001.
6. ITU-T. ITU-T Recommendation Z.120: Message Sequence Chart (MSC), Apr. 1996.
7. L. Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.
8. OMG. The Unified Modeling Language (UML) Specification - Version 1.3, June 1999. Joint submission to the Object Management Group (OMG) `http://www.omg.org/technology/uml/index.htm`.
9. G. Reggio, M. Cerioli, and E. Astesiano. Towards a rigorous semantics of UML supporting its multiview approach. In H. Hussmann, editor, *Fundamental Approaches to Software Engineering. 4th International Conference, FASE 2001 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001 Genova, Italy, April 2-6. 2001 Proceedings*, volume 2029 of *LNCS*, pages 171–186. Springer, 2001.