Journal of Universal Computer Science, vol. 7, no. 1 (2001), 3-18
submitted: 1/9/00, accepted: 13/10/00, appeared: 28/1/01

DisCo Toolset – The New Generation

Timo Aaltonen
(Tampere University of Technology, Finland
timo.aaltonen@tut.fi)

Mika Katara
(Tampere University of Technology, Finland
mika.katara@tut.fi)

Risto Pitkänen
(Tampere University of Technology, Finland
risto.pitkanen@tut.fi
Currently with Nokia Networks)

Abstract: Formal methods have been considered one possible solution to the so-called *software crisis*. Tools are valuable companions to formal methods: they assist in analysis and understanding of formal specifications and enable the use of rigorous techniques in industrial projects. In this paper, an overview of the new DisCo toolset is given. DisCo is a formal specification method for reactive and distributed systems. It focuses on collective behaviour of objects and provides a refinement mechanism that preserves safety properties. The toolset currently includes a compiler, a graphical animation tool, and a scenario tool for representing execution traces as Message Sequence Charts. A prototype verification back-end based on the PVS theorem prover also exists, and a model checking back-end based on Kronos as well as code generation facilities have been planned. In this paper, the operation of the DisCo toolset is illustrated by applying it to an example specification describing a simple cash-point service system.

Key Words: tools, reactive systems, formal specification, real time, animation, TLA
Category: D.2.2, D.2.1

1 Introduction

Formal methods have been considered one possible solution to the so-called *software crisis* arising from the increasing complexity of systems. However, they are not easy to adopt in industrial use. They often require new ways of thinking and some mathematical knowledge. Many difficulties can be overcome by providing appropriate tool support for users.

When talking about formal method tools, people usually first think of verification. Formal proofs are usually complicated and error-prone, and therefore theorem provers such as PVS [ORS92] and model checkers such as Kronos [Yov97] have been developed to assist in or completely automatize them. However, formal proofs are not the only way to analyze a formal specification.

Simulation and animation have proved valuable aids to validation and testing of formal models. They require that the specification has an *operational interpretation*, i.e. that it can be executed in some way.

DisCo [JKSSS90, dis99] is a formal specification method for *reactive* and *distributed* systems. Reactive systems are those that are in constant interaction with their environments. DisCo focuses on collective behaviour of objects and provides a simple refinement mechanism preserving safety properties. Tool support for animation of DisCo specifications has existed since the beginning of the 1990's [Sys91]. An improved version of the DisCo language containing support for real-time specification and a more flexible type system among other new features has been developed during the last few years. Due to technical limitations, including poor portability, of the first tool generation, support for the new language was not added in the old implementation. Instead, a whole new toolset was designed and implemented. This paper describes the new toolset.

The structure of the rest of the paper is as follows. Section 2 introduces the DisCo method and in Section 3 an example specification is introduced. In Section 4, the architecture of the new DisCo toolset and the individual tools are described and their usefulness is discussed. Section 5 discusses related and future work, and Section 6 contains some concluding remarks.

2 DisCo

2.1 The Basics

The basis of the DisCo method is in the *joint action* theory [BKS88, BKS89], which allows describing reactive and distributed systems at a high level of abstraction where implementation-level details are superfluous. Specifications are written in a programming-language-like notation.

The basic building blocks of DisCo specifications are *classes* and *actions*. Additionally *assertions*, *initial conditions* and *relation definitions* can be given. The only way to change the states of *objects* is to execute actions. Actions are atomic units of execution, which consist of *roles*, *parameters*, a *guard* and a *body*. Objects can participate in actions in certain roles. Only the states of the participating objects may change in an action. Parameters are values of basic types or records. Nondeterministic selection of participants and parameter values is restricted only by the boolean-valued guard. If the guard of an action evaluates to true, the action is said to be *enabled*. The body is a parallel assignment clause. The actions are executed in an interleaving manner: the choice of next action to be executed is made nondeterministically among enabled actions. Assertions and initial conditions are quantified expressions over objects. They do not limit the behaviour of the system, thus they need to be verified.

DisCo utilizes *closed world principle*, where objects are always modelled with their assumed environment. This enables constructing an animation tool, where the global state of the system, which determines enabledness of actions, is visualized.

Specifications are generic in the sense that the number of objects need not be fixed. Additionally, their initial states are only restricted by assertions and initial conditions. A generic specification can be *instantiated* by fixing the number of objects and setting their initial states.

2.2 Composition and Refinement in DisCo

Modularity of specifications is unlikely to be the same as modularity of implementation descriptions [Mai00]. DisCo specifications are structured in behavioural units called *layers*, each of which encapsulates a different view of the system being specified. A layer describes how a specification is changed when a new view is introduced. This structuring mechanism is orthogonal to ordinary architectural structuring, in which specifications are divided in architectural units reflecting the implementation-level units. Behavioural structuring — unlike architectural structuring — allows to postpone the definition of interfaces between units, until the collective behaviour of the total system has been captured. This leads to fewer changes to interfaces in the later development phases. For more detailed discussion about the structuring mechanism the reader is referred to [KSM98, KSM99].

Several specifications can be combined to one composite specification. In composition common parts of the specifications being combined are joined together. Certain rules are applied to synchronize some actions and join some sets of classes.

Specifications are refined stepwise towards implementation. At some level of abstraction hardware/software partitioning is made. The refinement mechanism is *superposition* (for more details see [JKSSS90]), which allows introducing and inheriting new classes and augmenting existing ones with new attributes. New roles and parameters can be added to existing actions. Bodies of old actions can be augmented with assignments to new variables. There are two examples of using superposition in an example specification given in Section 3: layer `till` is superimposed by layers `card` and `customer`. Layer `customer` superimposes layer `till` by introducing new class `Customer` and new relation `CustAcc`. Moreover, the layer refines actions `withdraw` and `deposit` originally introduced in `till`. Superposition preserves *safety properties* ("something bad never happens") by construction.

2.3 Formal Semantics and Operational Interpretation

Formal semantics of the DisCo language is given in terms of Temporal Logic of Actions (TLA) [Lam94]. TLA is a linear time logic which describes infinite sequences of states called *behaviours* and their properties. There are an infinite number of *state variables* and in all states of a behaviour each variable has a unique value. Formal semantics makes it possible to verify specifications in a rigorous way.

DisCo specifications have operational interpretation: actions of a specification are considered as operations for the system, which change its state from one to another. This is also essential from animation's point of view.

2.4 Timed Specifications

In terms of TLA, real time was added to the new version of DisCo as follows [KSK99]. It is assumed that each action is executed instantaneously. A global real-valued clock variable Ω, initialized as 0, is introduced to measure time from the beginning of a behaviour. An implicit parameter τ, which represents the

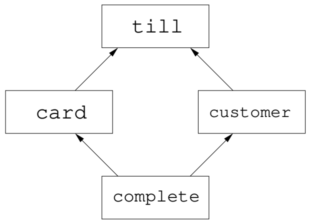

Figure 1: Layers of the cash-point system.

time when an action is executed, is added to each action. Moreover, all guards are implicitly strengthened by the conjunct

$$\Omega \leq \tau \leq min(\Delta) \ ,$$

where Δ denotes a multiset of *deadlines*. Furthermore, conjunct $\Omega' = \tau$ is added to the bodies of all actions.

Minimal separation requirement between actions A and B can be enforced by strengthening the guard of action B by conjunct $\tau \geq \tau_A + d$, where τ_A denotes the most recent execution moment of A. Deadlines are used for bounded response requirements. When a deadline $\tau + d$ is needed for some future action, a conjunct of the form $x' = \Delta_{on}(d)$ is given in the action body. The deadline is added to Δ and stored in a variable x. An implicit conjunct $\tau \leq min(\Delta)$ in all guards prevents advancing Ω beyond this deadline, until some action has removed the deadline with $\Delta_{off}(x)$. Type time, a synonym type of real, can be used in timed specifications. In the initial state, Δ can contain some initial deadlines.

Actions are not executed because time passes, but the passing of time is noticed as a consequence of executing an action. This may lead to *Zeno behaviours*, where time is prevented to grow beyond any bound. However, it is not harmful for a specification to allow such behaviours as long as every prefix of a behaviour can be extended to an infinite one allowing time to grow beyond any bound [AL94].

3 Example Specification

In this section a simple specification of a cash-point service system is given as an example. The system consists of four kinds of entities: *accounts*, *tills*, *cash cards* and *customers*. Cards may be inserted and ejected to and from tills. Money may be withdrawn from accounts using tills. Furthermore, money may be deposited directly to accounts.

The specification consists of four layers: `till`, `card`, `customer` and `complete`. Layers `card` and `customer` refine layer `till`, and layer `complete` refines the composition of `card` and `customer` as depicted in Figure 1.

Layer `till` (see below) defines the most abstract view to the system. Classes `Account` and `Till` are introduced. Assertion `balanceOK` states that balance of all accounts is always non-negative. Withdrawal is possible only from a till.

Action `withdraw` has two roles: `acc` (of class `Account`) and `till` (of class `Till`) and one parameter `amount` of type integer. It may be executed if the withdrawn amount is positive and there exists enough money on the account. An action describing depositing money is also given. At this level of abstraction nothing is specified about customers or cards:

```
layer till is
  class Account is
    balance: integer;
  end;

  class Till is end;

  assert balanceOK is ∀ acc: Account :: acc.balance ≥ 0;

  action withdraw(acc:Account; till: Till; amount:integer) is
  when amount > 0 ∧ acc.balance ≥ amount do
    acc.balance := acc.balance - amount;
  end;

  action deposit(acc:Account; amount:integer) is
  when amount > 0 do
    acc.balance := acc.balance + amount;
  end;
end;
```

Layer `customer` (see below) refines specification `till` by adding aspects related to customers to the model. The layer specifies that withdrawals are only possible for customers from their own accounts. Ownership is specified by relation `CustAcc`. Three dots in a guard of an action refinement (lines 12 and 19 below) refer to the guard of the original action. In action body (lines 13 and 20) they refer to the original body:

```
    layer customer is
      import till;

      class Customer is
 5      wallet: integer;
      end;

      relation CustAcc(Customer, Account) is 0..1:1;

10    refined withdraw(cust:Customer; acc:Account; till:Till; amount:integer)
      of withdraw(acc, till, amount) is
      when ... CustAcc(cust, acc) do
        ...
        cust.wallet := cust.wallet + amount;
15    end;

      refined deposit(cust: Customer; acc: Account; amount: integer)
      of deposit(acc, amount) is
      when ... CustAcc(cust, acc) do
20      ...
        cust.wallet := cust.wallet - amount;
      end;
    end;
```

Layer `card` below adds aspects of cash cards to the specification. `Tills` are augmented with a state machine `state` to model whether a card is inserted or not. When a till has a card, attributes `card` and `dlEject` become valid. The former is a reference to the inserted card and the latter is a deadline for ejecting the card. A card must be ejected (or another withdrawal must be done) within `deltaToEject` time units after a withdrawal. The layer introduces two new actions: `insertCard` and `ejectCard` and refines one existing action `with-`

draw, which now assigns deadline $\tau + deltaToEject$ to attribute dlEject and adds it to the multiset Δ of deadlines:

```
layer card is
  import till;

  class Card is
  end;

  extend Till by
    state: (noCard, hasCard);
    extend hasCard by
      card: reference Card;
      dlEject: time;
    end;
  end;

  constant deltaToEject: time := 20.0;

  relation CardAcc(Card, Account) is 0..1:1;

  action insertCard(till: Till; card: Card) is
  when till.state'noCard ∧
       not (∃ till2: Till :: till2.state'hasCard.card = card) do
   till.state := hasCard(card);
  end;

  refined withdraw(acc: Account; till: Till; amount: integer; card: Card)
  of withdraw(acc, till, amount) is
  when ... till.state'hasCard.card = card ∧ CardAcc(card, acc) do
    ...
    till.state'hasCard.dlEject@ || -- remove possibly existing deadline
    till.state'hasCard.dlEject@(deltaToEject); -- add new deadline
  end;

  action ejectCard(till: Till; card: Card) is
  when till.state'hasCard do
    till.state := noCard() ||
    till.state'hasCard.dlEject@; -- remove deadline
  end;
end;
```

The composite layer complete below concludes the specification and gathers all the refinements together. After composition the specification consists of classes Till, Account, Customer, and Card. The actions are withdraw, deposit, insertCard, and ejectCard. They contain all the refinements made in imported layers:

```
layer complete is
  import card, customer;
end;
```

To illustrate the situation if the specification consisted of only one layer the refinement steps corresponding to the action withdraw have been gathered into one explicit action below:

```
action withdraw(cust: Customer; acc: Account; till: Till;
                amount: integer; card: Card) is
when amount > 0 ∧ acc.balance ≥ amount ∧ -- from layer till
     CustAcc(cust, acc) ∧ -- from layer customer
     till.state'hasCard.card = card ∧ CardAcc(card, acc) do -- from layer card
  acc.balance := acc.balance - amount || -- from layer till
  cust.wallet := cust.wallet + amount || -- from layer customer
  till.state'hasCard.dlEject@ || -- from layer card
  till.state'hasCard.dlEject@(deltaToEject); -- from layer card
end;
```

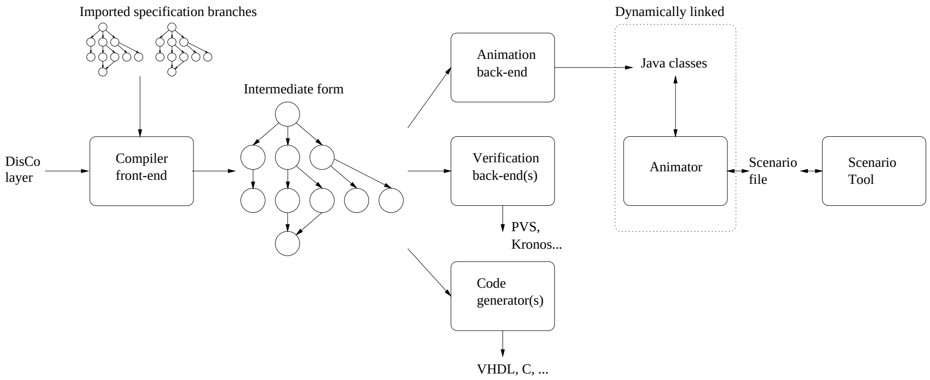

Figure 2: General architecture of the DisCo toolset.

4 Tools

4.1 Architecture

With the experience gained with the first generation of DisCo tools, we saw *portability*, *extensibility* and *usability* as the most important considerations when choosing implementation technologies and designing the general architecture of the new toolset. Portability was ensured by choosing ISO C++ and Java as the implementation languages. Extensibility was achieved by designing a general architecture, depicted in Figure 2, centered around a multi-purpose intermediate form. Moreover, a framework approach can be used to add or modify functionality of Animator. Usability has been a central factor in the design of the user interface.

The intermediate form produced by DisCo Compiler front-end is an explicit and flat representation of a layered specification. It is utilized by the compiler itself and several back-ends that produce input for different tools. Compiler front-end and back-ends, Animator and DisCo Scenario Tool pictured in Figure 2 are described in detail in the subsequent sections.

4.2 Compiler

DisCo Compiler plays a central role in the DisCo toolset. It works as a link between different tools and DisCo source code. Standard C++ was chosen as the implementation language for its good performance and portability. Object-oriented features and genericity of C++ are heavily utilized.

Functionality of the compiler is divided into two phases: *front-end* and several *back-ends*. The front-end produces an intermediate form of DisCo source. After successful translation into intermediate form, back-ends of the compiler may be used to produce input for different tools, like Animator or different verification tools.

The front-end of the compiler takes one DisCo layer and the intermediate forms of possibly imported specification branches as its inputs, and produces an intermediate form of the specification. The intermediate form corresponds one-to-one to the internal representation of the semantical tree (or DAG) of the specification being compiled. The intermediate forms of imported branches are merged into the tree representing the specification being compiled. In other words, the front-end produces composite specification described in 2.2. Moreover, it checks syntactic and semantic correctness. If an error occurs during the compilation, no intermediate form is produced.

Checking specifications by Compiler eliminates many awkward errors, and increases confidence in the correctness of them. For example the exact type system does not allow arbitrary operations between types which do not match. Moreover, Compiler checks that superposition is not violated in refinements. In composition Compiler joins the common parts of the specifications automatically, and checks that actions synchronized by the user are semantically correct.

Animation back-end of the compiler produces a *specification engine*, which is a Java package, for Animator. It offers an interface by which Animator can instantiate objects in the instantiation phase, and execute enabled actions in the execution phase. The engine notifies Animator about state changes and some other events. It also informs Animator about enabled actions. The assertions of the specification are guarded by the engine during execution.

Back-ends for producing input for several verification tools could be added to the toolset. We are also researching possibilities of producing VHDL or C code from an instantiation of a DisCo specification [PK99].

4.3 DisCo Animator

Because DisCo specifications are closed and have an operational interpretation they can be visualized and animated as discussed in Section 2. The general problem of executing a given specification is actually undecidable due to the expressive power of guards. In practice, however, most specifications can be executed automatically and user assistance can be used in complex specifications [Pit97].

Animator (Figure 3) enables validation, testing and debugging of specifications and offers an enhanced means of communication for designers, application experts and customers. Animation of DisCo specifications is very visual: objects, their state and their relations are depicted as graphical objects, executed actions and state changes are visualized by animation sequences and real time is depicted by a scrolling timeline displaying current time and any set deadlines.

Java was chosen as the implementation language of Animator for easy portability of the user interface and to enable producing a web-based version running as an applet for demonstration purposes. Moreover, the core of Animator is a framework that can be specialized to add or modify functionality by user-provided pieces of Java code. For example, custom classes can be created to handle the drawing of specification components. Basic Animator is actually just a default specialization.

Animation of a specification starts with instantiation. The user drags objects of classes she wants to instantiate onto the object view window and sets the initial

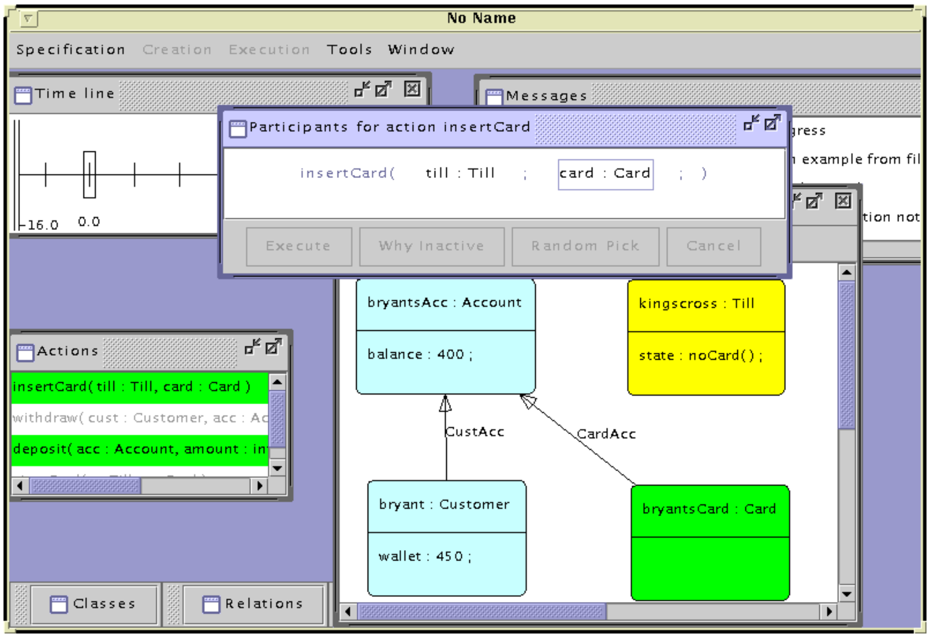

Figure 3: DisCo Animator.

values of variables using pop-ups that appear on the screen. Relations between objects are set by selecting a relation and then pointing at the objects one wants to add as relation pairs. Once the instantiation has been completed, animation may be started. First the tool checks that the initial conditions and assertions of the specification hold for the instantiation. Then, action guards are evaluated, and enabled actions are indicated by a green highlight color. In Figure 3, the user has selected action `insertCard` of the example cash-point specification to be executed and is now picking objects to participate in its roles.

Animation enables both validation and testing of specifications. The users can experiment with a specification on different levels of abstraction, and see which kind of executions are possible and which are not. Since animation is a very simple and intuitive yet accurate representation of behaviour, it can be used as an effective means of communication even when some involved party does not have the background to read formal specifications. This means that application field experts and end users can take part in validation in an early phase of development.

Animation can also be used as a means of testing the specification. Let us assume for a while that we had specified the `deposit` action of layer `till` (see Section 3) in the following way:

```
action deposit(acc:Account; amount:integer) is
when true do
  acc.balance := acc.balance + amount;
end;
```

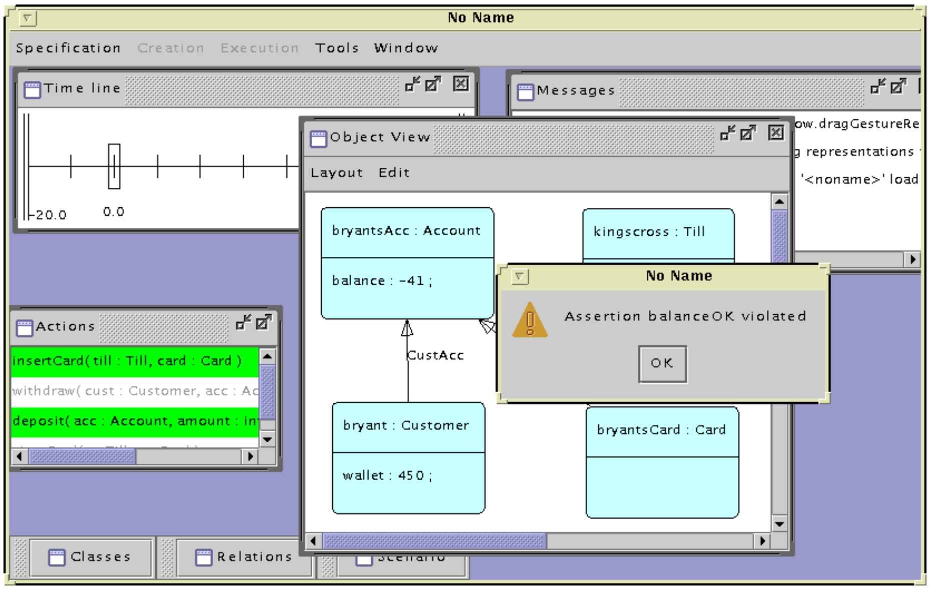

Figure 4: Assertion failure in ANIMATOR.

A mistake of this kind is quite easy to make, although on closer investigation it is obvious that a negative deposit is possible. Thus behaviours that do not satisfy the assertion `balanceOK` (balances are always non-negative) are allowed by the specification. Animation, especially random execution with random parameter values, often leads to the discovery of errors of this kind, and usually with considerably smaller effort than verification. Once the tool discovers that an assertion does not hold, it displays a pop-up window like the one shown in Figure 4. Relation form violations (e.g. a one-to-one relation is treated like one-to-many) are reported in a similar manner.

Another common class of errors that are often found with animation are also due to incorrect action guards. Since ANIMATOR indicates enabled actions in each state, actions that are altogether disabled or enabled when they should not can often be found to be erroneous even before they are selected for execution.

Execution traces can be rerun and saved as *scenario* files. After modifications specifications can be tested with ANIMATOR using saved scenarios. Furthermore they can be processed by DISCO SCENARIO TOOL.

4.4 DISCO SCENARIO TOOL

Message Sequence Charts (MSCs) are a widespread notation for describing inter-object communication. Their main strength is intuitive visual representation. Objects are depicted as vertical lines and messages sent to other objects as horizontal lines (arrows). MSCs are limited because they can only represent one

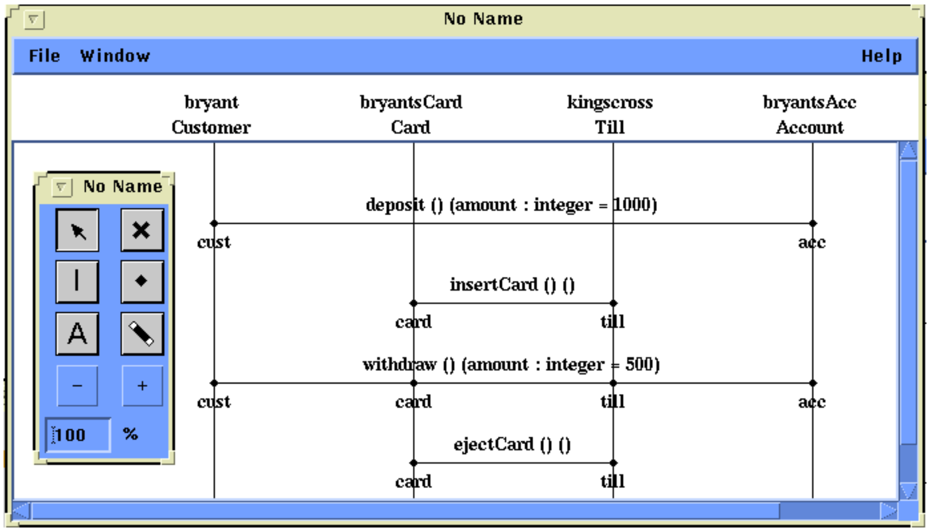

Figure 5: DisCo Scenario Tool.

scenario with a small number of objects at a time. However, at a high level of abstraction they can be used to capture some essential scenarios.

DisCo Scenario Tool (DST, Figure 5) is a tool for displaying execution scenarios as MSCs. In the Figure, the larger window displays a scenario and the smaller window contains buttons corresponding to different instruments available for modifying it. Executed actions are interpreted as messages between participating objects. The tool can be started from Animator to display the current scenario. It includes features to hide some of the actions and objects, add comments and print MSCs on multiple sheets. Furthermore, DST can be used to modify existing and create new MSCs which can be animated by Animator.

Because MSCs are a well-known notation, we believe that they make validation of specifications more user friendly and faster. They can be used to examine erroneous scenarios possibly ending in a assertion failure thus providing graphical representation of error traces. This is especially important in the case of lengthy scenarios produced by random execution. Moreover, DST can be used to capture some initial requirements as MSCs which can be later used to test the specification. Obviously, also new scenarios can be created for this purpose.

To illustrate the use of MSCs in conjunction with the example specification, consider the one drawn using DST in Figure 5. A diamond in each crossing of vertical and horizontal lines denotes that the object corresponding to the vertical line is a participant in the action corresponding to the horizontal line. In the MSC, the topmost horizontal line depicts an execution of action `deposit` with parameter value 1000. The three horizontal lines below depict executions of actions `insertCard`, `withdraw` and `ejectCard`, respectively. Using Animator it is possible to check that this scenario is indeed allowed by the example specification.

4.5 Verification

4.5.1 Verification versus Validation

Formal semantics enables rigorous verification of specifications (see section 2.3). In system design, validation and verification complement each other. The former answers the question whether we are designing the *right product* and the latter whether we are designing the *product right*. The DisCo toolset does not include any dedicated verification tool, instead a number of more general purpose verification tools can be used. Two main approaches to verification are *theorem proving* and *model checking*. Both approaches have been recently researched in the DisCo project.

4.5.2 Theorem Proving

The state space of a generic DisCo specification is inherently infinite. Therefore, the most natural verification method is theorem proving. In [Kel97b], a mapping from a subset of the DisCo language into the logic of the PVS [ORS92] theorem prover was described. PVS offers high level decision procedures which assist interactive theorem proving.

As an example, PVS was used to verify the assertion `balanceOK` of the layer `till` (see Section 3). We used a prototype tool which supports mechanical verification of invariant properties [Kel97a]. In the prototype, appropriate parts of TLA are formalized. First, the DisCo specification was translated into input of PVS. Then it was showed that executing action `withdraw` or `deposit` cannot break an invariant corresponding to the assertion. After proving that the invariant holds in the initial state, it was deduced inductively that the assertion holds in all behaviours of the specification. By disallowing assignments to old variables, the refinement mechanism of DisCo preserves all safety properties by construction. This means that the invariant holds also in all later refinements.

4.5.3 Model Checking

In addition to theorem proving, instantiations of DisCo specifications can be verified using model checking approach. A mapping from a subset of finite instantiations of DisCo specifications into *timed automata* [AD94] was described in [AKP00]. There are a number of model checkers that can be used to verify systems given as timed automata, including Kronos [Yov97] and Uppaal [LPY97].

Currently, instantiations have to translated manually, but mechanical support could be added to the current DisCo tools in the way explained in Section 4.2. For verifying the instantiations, we have used Kronos. In Uppaal the communication between automata is one-to-one, which makes the mapping of multi-object actions a bit more complicated.

An instantiation of the cash-point service system consisting of two instances of each class was translated into timed automata. In Figure 6, the two automata corresponding to a till are depicted. The automata on the left and right hand side correspond to the functional and real-time behaviour of the till, respectively.

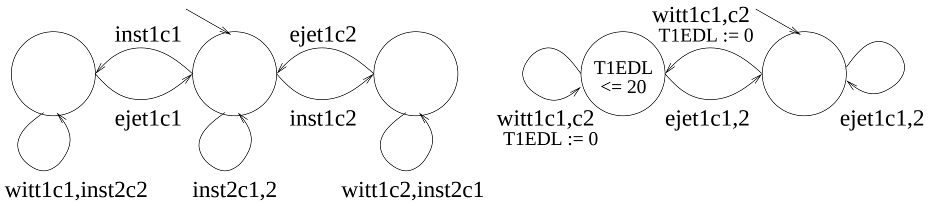

Figure 6: Timed automata corresponding to a till.

Non-Zenoness, or that time can proceed beyond any bound (see Section 2.4), can be verified by verifying the TCTL property [Yov97, HNSY94]

$$init \Rightarrow \forall\Box(\exists\Diamond_{=1} true) \ .$$

Intuitively the property means that it is always possible for time to proceed by one unit. If Non-Zenoness does not hold it might be that some safety properties only hold because time stops and nothing happens. By applying the forward analysis of Kronos, the instantiation is found to satisfy the property.

As mentioned above, model checking can only be applied to finite instantiations. However, this is not a severe restriction since almost all implementations of specifications are finite instantiations. Besides verification of real-time properties, the model checking approach enhances user-controlled mechanical theorem proving by finding counterexamples efficiently. Moreover, proposed invariants can be pre-checked for specific instantiations before attempting to prove them for the generic specification.

5 Related and Future Work

Related research can be searched in the area of animation and mapping between formalisms. In the literature, animation can refer to both graphical animation and plain simulation of specifications. However, the underlying principle, execution of the specification, is the same.

A well-known formal method B [Abr96] offers extensive tool support. For example, the B-toolkit [LH96], includes a non-graphical animation facility which allows the user to invoke B Abstract Machine operations interactively. The B-toolkit is a commercial product.

The "Tools for TLA based specifications" project of the University of Dortmund has done some work on graphical animation of TLA+ [Lam99] specifications. They report having used an interpreter in combination with an animation system originally intended for animating sequential algorithms [MK93]. TLA+ is basically syntactic sugar for writing more structured TLA specifications, which makes this work quite closely related to ours.

A very interesting TLA+-related future work item would be to investigate the possibility of integrating TLC [YML99], a model checker for TLA+ specifications, in the DisCo toolset.

6 Conclusions

There are a number of methods intended for formal specification, validation and verification of hardware/software systems. There are also tools, both academic and commercial, available to support some of the methods, mainly those that are trying get industrial attention. The diversity of applications entails that no single method or tool will ever overcome the others. DisCo focuses on reactive and distributed systems. In practice almost all safety critical systems are reactive, moreover, many of those have to meet some real-time requirements.

So far, the success stories of formal methods are mainly in the area of hardware verification. Formalisms and tools that are easily adopted in an industrial design process have advantage in the industrial setting. It is desirable that these methods will also pave the way for other methods trying to import new ways of thinking to the whole design process.

The main strengths of DisCo are the ability to capture collective behaviour at a high level of abstraction, stepwise refinement towards implementation, and behavioural structuring of specifications using logical layering. Furthermore, animation at the early stages of development has been commended. The new DisCo toolset will enable conducting industrial size case studies which are needed to evaluate real applicability of any method.

The new generation of the toolset includes Compiler, Animator and Scenario Tool. It has an extensible architecture centered around a multi-purpose intermediate form for DisCo specifications. Portability has been ensured by the use of standard C++ and Java as the implementation technologies. Furthermore, usability has been emphasized in the design of the user interface. The toolset consists of about 40,000 LOC C++ (Compiler) and 80,000 LOC Java (Animator and Scenario Tool) and is still under development. Additional tools have been planned to assist in verification and code generation.

Acknowledgements

Altogether around ten people have contributed to the development of the toolset during a period of three years. The authors would like to thank all the people involved in the development and especially Joni Helin for his work on Animator, and Tero Jokela, Tero Jussila, Tero Kivimäki and Mikko Vainikainen for developing DST. Funding from Tekes (National Technology Agency), Nokia Research Center and Space Systems Finland is gratefully acknowledged.

Pertti Kellomäki provided valuable help by proving the invariant using PVS, and Tommi Mikkonen by reading a manuscript of this paper carefully.

References

[Abr96] J.-R. Abrial. *The B-Book: Assigning Programs to Meanings.* Cambridge University Press, 1996.

[AD94] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, April 1994.

[AKP00] Timo Aaltonen, Mika Katara, and Risto Pitkänen. Verifying real-time joint action specifications using timed automata. In Yulin Feng, David Notkin, and Marie-Claude Gaudel, editors, *16th World Computer Congress 2000,* *Proceedings of Conference on Software: Theory and Practice*, pages 516–525, Beijing, China, August 2000. IFIP, Publishing House of Electronic Industry and International Federation for Information Processing.

[AL94] Martín Abadi and Leslie Lamport. An old-fashioned recipe for real time. *ACM Transactions on Programming Languages and Systems*, 16(5):1543–1571, September 1994.

[BKS88] R. J. R. Back and R. Kurki-Suonio. Distributed cooperation with action systems. *ACM Transactions on Programming Languages and Systems*, 10(4):513–554, October 1988.

[BKS89] R. J. R. Back and R. Kurki-Suonio. Decentralization of process nets with centralized control. *Distributed Computing*, 3:73–87, 1989.

[dis99] The DisCo project WWW page. At URL `http://disco.cs.tut.fi`, 1999.

[HNSY94] T. A. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic model checking for real-time systems. *Information and Computation*, 111(2):193–244, 1994.

[JKSSS90] H.-M. Järvinen, R. Kurki-Suonio, M. Sakkinen, and K. Systä. Object-oriented specification of reactive systems. In *Proceedings of the 12th International Conference on Software Engineering*, pages 63–71. IEEE Computer Society Press, 1990.

[Kel97a] Pertti Kellomäki. *Mechanical Verification of Invariant Properties of DisCo Specifications*. PhD thesis, Tampere University of Technology, 1997.

[Kel97b] Pertti Kellomäki. Verification of reactive systems using DisCo and PVS. In John Fitzgerald, Cliff B. Jones, and Peter Lucas, editors, *FME'97: Industrial Applications and Strengthened Foundations of Formal Methods*, number 1313 in Lecture Notes in Computer Science, pages 589–604. Springer–Verlag, 1997.

[KSK99] Reino Kurki-Suonio and Mika Katara. Logical layers in specifications with distributed objects and real time. *Computer Systems Science & Engineering*, 14(4):217–226, July 1999.

[KSM98] Reino Kurki-Suonio and Tommi Mikkonen. Liberating object-oriented modeling from programming-level abstractions. In J. Bosch and S. Mitchell, editors, *Object-Oriented Technology, ECOOP'97 Workshop Reader*, number 1357 in Lecture Notes in Computer Science, pages 195–199. Springer–Verlag, 1998.

[KSM99] Reino Kurki-Suonio and Tommi Mikkonen. Harnessing the power of interaction. In H. Jaakkola, H. Kangassalo, and E. Kawaguchi, editors, *Information Modelling and Knowledge Bases X*, pages 1–11. IOS Press, 1999.

[Lam94] Leslie Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, 1994.

[Lam99] Leslie Lamport. Specifying concurrent systems with TLA+. In Manfred Broy and Ralf Steinbrüggen, editors, *Calculational System Design*, pages 183–247. IOS Press, 1999.

[LH96] Kevin Lano and Howard Haughton. *Specification in B: an Introduction using the B Toolkit*. Imperial College Press, London, 1996.

[LPY97] Kim G. Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a nutshell. *International Journal on Software Tools for Technology Transfer*, 1(1+2):134–152, 1997.

[Mai00] Tom Maibaum. Mathematical foundations of software engineering: a roadmap. In *Future of Software Engineering*, pages 163–172, Limerick, Ireland, 2000. ACM.

[MK93] A. Mester and H. Krumm. Animation von TLA spezifikationen. Beitrag zum 3. Fachgespräch der GI/ITG-Fachgruppe 'Formale Beschreibungstechniken für verteilte Systeme', München, June 1993. German abstract available at http://ls4-www.informatik.uni-dortmund.de/RVS/P-TLA/papers.html.

[ORS92] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, number 607 in *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, USA, June 1992. Springer-Verlag.

[Pit97] Risto Pitkänen. DisCo-spesifikaatioiden simulointi Javalla (Simulation of DisCo specifications in Java). Master's thesis, Tampere University of Technology, June 1997. In Finnish.

[PK99] Risto Pitkänen and Harri Klapuri. Incremental cospecification using objects and joint actions. In H. R. Arabnia, editor, *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'99)*, volume VI, pages 2961–2967. CSREA Press, June 1999.

[Sys91] Kari Systä. A graphical tool for specification of reactive systems. In *Proceedings of the Euromicro'91 Workshop on Real-Time Systems*, pages 12–19, Paris, France, June 1991. IEEE Computer Society Press.

[YML99] Yuan Yu, Panagiotis Manolios, and Leslie Lamport. Model checking TLA+ specifications. In *Proceedings of Correct Hardware Design and Verification Methods (CHARME'99)*, number 1703 in Lecture Notes in Computer Science, pages 54–66. Springer–Verlag, 1999.

[Yov97] Sergio Yovine. Kronos: A verification tool for real-time systems. *International Journal on Software Tools for Technology Transfer*, 1(1+2):123–133, 1997.