# CONSTRAINT-ORIENTED FORMAL MODELLING OF OO-SYSTEMS*

Günter Graw
Peter Herrmann
Heiko Krumm

Department of Computer Science
University of Dortmund
D-44221 Dortmund, GERMANY
{ graw | herrmann | krumm } @ls4.cs.uni-dortmund.de

**Abstract:** In addition to static structures, the Unified Modelling Language UML supports the specification of dynamic properties by means of state charts and interaction diagrams. Each diagram, however, only reflects partial aspects of the system. A common behavior model is lacking while it is necessary to relate the diagrams with each other and to enable the verification of dynamic system properties. The formal process specification technique cTLA provides for modular descriptions of behavior constraints and its process composition operation corresponds to superposition. Therefore, a UML diagram can be represented by a cTLA description which is as well modular as it can be combined with the descriptions of other diagrams.

**Keywords:** formal object model, cTLA, UML, state chart, interaction diagram

## 1 INTRODUCTION

Many object systems which run in distributed environments have complex dynamic behaviors. In addition to the design of the object class structure, one has to develop suitable system configuration schemes, which describe the creation, deletion, and localisation of objects, the establishment of execution threads, as well as the interaction

* In Proceedings of the *2nd IFIP WG6.1 International Working Conference on Distributed Applications and Interoperable Systems (DAIS 99)*, pages 345–358, Helsinki, June/July 1999. Kluwer Academic Publisher.

behavior of objects during execution. There are major design tasks which call for support by precise and concise behavior descriptions as they are in the scope of typical formal process specification techniques like Lotos [10]. Unfortunately, many systems are very complex and the development of complete formal models would be too expensive. The systems often contain parts where concurrency and interactions are relatively easy to survey. Therefore partial models are of interest concentrating on the 'difficult' system parts. This is reflected by the Unified Modelling Language UML [14] proposing diagram types directly devoted to aspects of the dynamic behavior of parts of the object system [2]. So, state charts refer to the behavior of objects and interaction diagrams describe connections and supported interactions between objects.

Our work concentrates on the transformation of UML-based descriptions into formal specifications which support the understanding of and the formal reasoning with properties of the dynamic system behavior. In particular, we address the problem of partial specifications and their well-defined combination by the application of the specification technique cTLA [6, 11]. It is based on TLA [13] and especially supports the modular description of behavioral constraints of system parts by means of constraint processes (cf. [16]). Thus, single UML diagrams can separately be transformed to corresponding constraint processes. Due to the superposition character of cTLA's process composition, the properties expressed by single processes are present in each embedding system. Therefore, constraint compositions are consistent and describe suitable common models of diagram combinations.

The work is related to a series of existing approaches. Superposition of behavioral properties of processes has been introduced in [3]. Also, [1] uses superposition in a setting where structured state transition systems model processes which interact via joint actions. Several other authors reported on the formal modelling of aspects of the dynamic behavior of object systems. So, [17] has formalized diagrams of OOA languages. In [12] the services, which are exchanged between objects are specified state based. [9] reports on recent work which applies the action system approach in order to model interactions in object systems. The semantics of UML type structures has been formalized using Z in [4]. The approaches, however, do not yet support freely combinable modular diagram models.

In the sequel we firstly outline behavior aspects of object systems and relevant UML components. After shortly introducing cTLA, we enter into the description of the formal models. The global formal model provides a common understanding for the context of constraint processes. The constraint processes, their structure, and the way they model UML state chart and interaction diagrams are described thereafter.

## 2 DYNAMIC BEHAVIOR

Object-oriented methods and resulting object models have a wide field of application. This is also reflected by the Unified Modelling Language UML [14] and the different UML-based methods and procedures. Thus, on the one hand, UML may be applied in the whole spectrum of development process phases beginning with early requirements definitions up to the design of implementations. On the other hand, there is a wide spectrum of aspects and property types of object systems which is addressed by

UML. Our approach concentrates on the issues of concurrency and concurrent object interaction. In order to identify the scope of our work in more detail, we give a short outline of the underlying view to object systems. We view an object system as a set of objects and a set of threads of activity. An object system evolves during runtime from an initial object configuration. This execution is determinated by the (relatively) static object class definitions, by the description of the initial configuration, and by the interactions with the object system's environment. Each object encapsulates operations and attributes. It belongs to an object class defining the operations and attributes of the class. There may be one or more class hierarchies, where subclasses inherit properties of superclasses. We do, however, not concentrate on more static issues of classes, inheritance, and polymorphism. Instead, we concentrate on the object configurations occuring at runtime. Therefore, we want to refer to the initial configuration of an object system and to the different steps of execution which change the current system state. The relevant system state depends on the set of currently existing objects, and in more detail on their data and control states. The data state corresponds to the values of the object's attributes. The control state defines which operations, at which positions, and under which synchronisation conditions are currently under execution. The state of the object system as a whole identifies the set of currently existing objects and moreover contains the object states as components.

In the UML an execution step within an object corresponds to an **action**. Actions are distinguished in **send actions**, **call actions**, **local invocations**, **create actions**, **terminate actions**, **destroy actions**, **return actions**, **raise actions**, and **uninterpreted actions**. Some actions like local invocations effect only the local object. Other actions (e.g., call actions) influence foreign objects, too. Here, two objects communicate with each other. The communication is specified by means of **requests** which can be either **signal** requests or **operation** requests. A **signal** request triggers a receiver asynchronously with no reply (e.g., **exceptions** used in a fault situation). Objects communicate by means of **operation** requests if the calling object demands a service provided by the called object. The request is forwarded by a **message** instance which can carry a set of arguments. The called object may invoke a return action leading to a return message. Messages can be send sequentially in one thread and concurrently in different threads.

Services provided by objects to other objects are called **operations**. An operation has a unique signature restricting the possible actual parameters. Operations are triggered by the reception of a message due to a call action of another object. Operations may be called synchronously or asynchronously. In the case of a synchronous call the caller enters a waiting state and is blocked till the corresponding return message arrives. In case of an asynchronous call the caller proceeds without blocking. An object can also invoke own operations calling a local invocation action.

Object creation and deletion is originated by create, destroy and terminate actions. A create action creates an object based on the specified set of classifiers. A destroy action ceases a foreign object instance to exist while a terminate action ceases an instance itself to exist.

Figure 1 The class diagram.

In order to facilitate the application of our model, we assume the run-to-completion semantics (RTC).This fundamental semantics has the assumption that requests are processed in sequence, where each incoming request stimulates a run-to-completion (RTC) action sequence. The next external request is dispatched after the previous RTC action sequence has completed. This assumption simplifies the synchronization of an object since the incoming request is processed only after the object has reached a well-defined (stable) state configuration.

## 3 MODELLING BASED ON UML

Object-oriented systems are specified in UML by means of several diagrams each modelling a certain aspect of the system. Below, we will introduce class diagrams, use case diagrams, collaboration diagrams, and statechart diagrams which are particularly important for the dynamical aspects of object-oriented systems. The application of the diagrams is explained by a simple example consisting of a business transaction which withdraws money from a business account.

In the UML class diagrams model the static structure of classes and their relations. Each class is specified by an icon containing the class attributes and operations as well as privacy properties. Fig. 1 depicts the class diagram of our example system which consists of two classes modelling the two system parts business account and business transaction. Each class contains an attribute with the name *"PersNr"*. By these attributes unique identifiers as an account number are specified. The class *Business Account* contains an operation *withdraw* modelling the withdrawal of money from the account. In *Business Transaction* the operation *runTrans* specifies the transaction performing the withdrawal. The relations between classes in a class diagram may be associations, aggregations, compositions, and inheritance. In our example, the two classes are coupled by means of associations. The numbers at the end of the dashed line indicate the multiplicity of one. Thus, one business transaction must be associated with exactly one business account and vice versa. For our approach class diagrams are important for the definition of the global formal model (Sec. 5).

In order to facilitate the description of complex object-oriented systems, the UML enables the specification of single **use cases** each describing only a relevant function of the system. A use case is modelled by the sequence of messages exchanged between the system and outside interactors, so-called **actors**, together with actions performed by the system. An actor can participate in different use cases and therefore is modelled

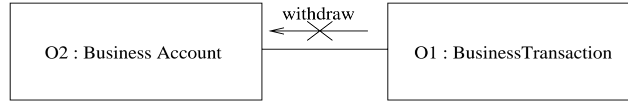


Figure 2 The collaboration diagram of the example.

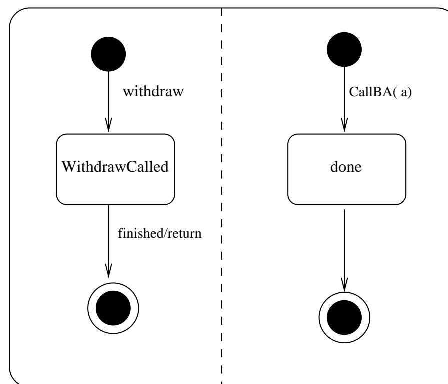


Figure 3 The Statecharts of the classes Business Account and Business Transaction.

as a set of roles. In each use case it plays one particular role. Use cases are specified by use case diagrams representing the relations between the system and the actors.

**Collaboration diagrams** describe a set of participants and relations that are meaningful to accomplish a certain purpose like a use case or an operation. Fig. 2 shows a collaboration diagram of our example. It specifies the use case performing a withdrawal operation of the business account. The participants in a collaboration, usually objects, are called **instances**. In the example, *O1* of the class *Business Transaction* and *O2* of the class *Business Account* are the instances. In order to keep a collaboration diagram simple, only the **Classifier Roles** of the instances, i.e., the roles essential for realizing the particular purpose, are specified. To specify the relation between instances, links and interactions are used. A **link** is an instance of the association specifying that instances are connected. The line between *O1* and *O2* denotes the link between the two objects. The messages sent between instances are modelled by **interactions**. In our example the interaction *"withdraw"* specifies the messages used to invoke the operation *withdraw* in *O2* and to return the result of this operation. By different interaction symbols one can specify if messages invoke operations synchronously or asynchronously. Furthermore, aspects of message synchronization can be described by means of numbers indicating the execution order of the messages.

The local behavior of objects or interactions are specified by statechart diagrams. Similarily to Harel's statecharts [5], a state machine models the behavior of an object or interaction by **states** and non-atomar **transitions**. Transitions are stimulated by events and depend on **guard conditions** modelling additional local conditions. Furthermore,

```
PROCESS Object (cf : ClassFrame ; id : OId ; class : ClassName)
  VARIABLES
    state : cf.State ;  ! object data, links, and control
    lifecycle : (unborn, alive, dead);  ! life cycle state
    qu : queue of Message ;     ! messages received
    awaitReturnOf : MessageId ; ! if blocked: call message id
     ...                        ! message id management, etc.
  INIT ≙ lifecycle = unborn ∧ ... ; ! initially, object does not exist
  ACTIONS
    callAction ( receiver : OId ; ! send Call-message
                 objState, objNextState : cf.State ;
                 message : Message ; mode : SyncMode ) ≙
      lifecycle=alive ∧ lifecycle'=lifecycle ∧
      cf.nextState(state,state,message,receiver,mode) ∧
      awaitReturnOf'=IF mode=blocking THEN message.id ELSE nullId ∧
      qu'=qu ∧ ... ;
    receiveAction ( objState, objNextState : cf.State ;
                    message : Message ) ≙ ! receive a message
      ... ; ! if message is a return message awaited, it is
            ! inserted at the front of qu otherwise appended.
    returnAction ( receiver : OId ; ! send Return-message
                   objState, objNextState : cf.State ;
                   message : Message ) ≙ ... ;
    createAction ( receiver : OId ; ! send Create-message
                   objState, objNextState : cf.State ;
                   message : Message) ≙ ... ;
    startAction ( ... ) ≙ ... ;     ! initiate object
    terminateAction ( ... ) ≙ ... ; ! destroy object
    localInvocation ( ... ) ≙ ... ; ! local operation call
    internalAction ( ... ) ≙ ... ;  ! internal execution step
   END
```

Figure 4 Process type Object.

the transitions are labelled by an action list identifier describing the actions executed if the transition fires. Fig. 3 depicts the **state charts** of the classes *Business Account* and *Business Transaction*. If the operation *runTrans* is called in *Business Transaction*, the transition between the initial state and the state *done* is fired invoking a call action for the operation *withdraw*. Since this operation is defined synchronously, the transition does not terminate until the reception of the return message. Afterwards, the system is in the state *done* and can proceed to the final state by an internal transition. In *Business Account* the reception of the call event triggers the *withdraw* operation and fires the transition from the initial state to *"WithdrawCalled"*. After an event indicating the end of the operation the transition to the final state is fired and a return action is invoked.

## 4 SPECIFICATION TECHNIQUE CTLA

The compositional TLA specification style cTLA [6, 11] supports the definition of parametrized process and system types. Systems are composed from processes which interact via joint actions like Lotos processes [10]. As in Lotos, processes can model implementation parts as well as logical system constraints (cf. [16]). Process specifications are formed by instantiation of cTLA process types. A process type may either describe a simple process or a (sub)system.

The process type *Object* outlined in Fig. 4 is an example of a process type specifying a state transition system directly. It describes the behavior of an object (cf. Sec. 5). The header of the cTLA process type declares the process type name and the process parameters *cf*, *id*, and *class*. The state of the process is modelled by the variables in the `VARIABLES` section. The process starts with an initial state corresponding to the condition `INIT`. The next state relation is described by means of actions which are defined in the part `ACTIONS`. An action (e.g., *callAction*) is a predicate over action parameters (e.g., *receiver*), state variables (e.g., *lifecycle*), and so-called primed state variables referring to the next state (e.g., *lifecycle'*). In the course of time, a process may perform action steps (i.e., it may change its state in accordance with the definition of an action) or it may perform so-called stuttering steps (i.e., the process does not change its state while the environment of the process performs a state transition).

The cTLA process type outlined above describes safety properties. Liveness constraints are described by additional fairness assumptions of actions (e.g. the expression `WF : callAction` denotes that *callAction* has to be performed weak fairly). Fairness forces the activity of a process. A fair action cannot be enabled for an infinite period of time without being executed. By weak fairness the execution of the action is required only if it would incessantly be enabled otherwise. By strong fairness the execution of an action is guaranteed even if the action is disabled sometimes. Unlike TLA/TLA+, cTLA provides for conditional fairness assumptions to keep the compositionality of systems. `WF : callAction` guarantees executions of *callAction* only, if *callAction* is enabled in its local process as well as the environment of this process tolerates *callAction*.

Systems and subsystems are described as compositions of processes. A composition consists of a set of concurrent processes. Each process encapsulates its variables and can change its state by the atomic execution of actions. The state of a system is represented by the vector of the state variables of its processes. State transitions of the system correspond to system actions. A system action is a logical conjunction of process actions and process stuttering steps where each process contributes to the system action by exactly one action or a stuttering step. Thus, concurrency can be modelled by interleaving and the coupling of processes can be modelled by joint actions.

Fig. 5 shows as an example the process type *GlobalSystem*. This system type is composed from the processes declared in the `PROCESSES`-section. Here, the system consists of *OId* many instances *obs[i]* of the process type *Object* which is described by the cTLA construct `ARRAY`.

The actions of the system *GlobalSystem* are listed in the `ACTIONS`-section. The processes composed to a system participate either by a process action or by a stuttering step to a system action. For instance, the action *operationCall*, modelling the call of an operation is a coupling of the process action *callAction* of the process instance *obs[caller]*, which describes the calling object, and of the process action *receiveAction* of the process instance *obs[callee]* specifying the called object. The other objects participate to *callAction* by stuttering steps. The transfer of data between processes is modelled by the action parameters, e.g., the parameters *message* of all process actions

```
PROCESS GlobalSystem ( cfs : ARRAY [class] OF ClassFrame ;
                       OId : data type ;
                       classOf : [OId → class] )
  PROCESSES  ! the infinite array of object processes
    ARRAY obs [OId] of Object(cfs[classOf(index)],index,classOf(index));
  ACTIONS ! system actions defining the coupling of the objects
    operationCall (caller, callee: OId ; ! caller calls operation of callee
                   callerState, callerNextState,
                   calleeState, calleeNextState: State ;
                   message : Message ; mode : SyncMode ) ≙
      obs[caller].callAction(callee,callerState,callerNextState,message,mode) ∧
      obs[callee].receiveAction(calleeState,calleeNextState,message) ∧
      ∀ i ∈ OId\{caller,callee} obs[i].Stutter ;
    operationReturn (caller, callee: OId ; ! callee operation returns to caller
                     callerState, callerNextState,
                     calleeState, calleeNextState: State ;
                     message : Message ) ≙
      obs[caller].receiveAction(callerState,callerNextState,message) ∧
      obs[callee].returnAction(caller,calleeState,calleeNextState,message) ∧
      ∀ i ∈ OId\{caller,callee} obs[i].Stutter ;
    objectDestroy (this : OId ; ! one object terminates
                   objState, objnextState : State ) ≙
      obs[this].terminate(state,nextState) ∧
      ∀ i ∈ OId\{this} obs[i].Stutter ;
    objectCreate (...) ≙ ... ; ! object sends create message
    objectStart (...) ≙ ... ;  ! object reacts on create message
    localInvocation (...) ≙ ... ; ! object performs local opcall
    internalAction (...) ≙ ... ;  ! object performs internal step
  END
```

Figure 5 Process type GlobalSystem.

coupled in the system action *callActions* have to carry identical values. cTLA facilitates the combination of different property types like safety and liveness. In the resource oriented specification style, this supports the combined description of all relevant aspects of a component in a single process type. In the constraint oriented specification style, different aspects of a component are described by separate constraint processes. However, liveness properties may be combined with models of the safety behavior of the component's environment in order to support the modularity of verifications (cf. [7]).

## 5 GLOBAL FORMAL MODEL

The global formal model models an object-oriented system in detail. It represents all object instances of the system, their class memberships, life-cycle states, data attributes, and links. Moreover, the state transitions of the object instance models correspond to execution steps of object operations. If one would define the global formal model of a special object-oriented system, one could analyse all relevant functional aspects on the basis of this model. The model, however, would be too complex to support practical verifications. Our approach therefore does not rely upon the availability of the detailed definition of the global formal model. We only need to know its structure in order to provide for a common understanding of the relevant behavior steps of the execution of object-oriented systems. In particular, the actions of the model are of importance.

```
ClassFrame ≙ ⟨ State ≙ [[ att1 : type1 ; att2 : type2 ; ...
                          link1, link2, ... : OBId ;
                          control : stack of ProgramCounter ]] ;
              nextState ( curState, nextState : State ;
                          actMessage : Message ; peer : OId ;
                          mode : (blocking, nonblocking) ) : relation ⟩
```

Figure 6 ClassFrame — scheme.

They will be used later on to relate constraint models to the global formal model. This defines the semantics of the interfaces of constraint models and indirectly relates the different constraint models of a system with each other. Accordingly the following description focuses on the model structure and its actions.

Each object is modelled as an instance of the cTLA process type *Object*, which represents the common features of objects as well as class specific ones. The class specific properties are imported into a process using a process parameter of type *ClassFrame*. We assume, that from each class definition of an object system a corresponding *ClassFrame*-structure can be compiled. As outlined in Fig. 6, a *ClassFrame* defines the data type *State* for the data and control state of an object as well as the relation *nextState* defining the detailed state transitions which correspond to execution steps of object operations. It consists of state pairs relating current states with their successors. Additionally, *nextState*-tuples can contain the value of a message which is connected with the corresponding state transition, the object identifier of the peer object of the message transfer, and in case of operation call transitions the synchronisation mode of the invocation.

The definition of the process type *Object* is outlined in Fig. 4. Besides of the *ClassFrame*-parameter *cf*, the parameter *id* represents the unique object identification and *class* the class membership. The following variable *state* comprises the object specific state components. The other variables reflect the common object model features. So, the variable *lifecycle* represents the current life cycle state of an object. We model the dynamically changing set of existing object instances of a system by a static but infinite set of processes. Thus, creation and deletion of objects is modelled by transitions of the lifecycle states of the corresponding processes. The following variables are devoted to the message interface of an object. With respect to the receipt of messages, pending messages are stored in the queue *qu*. In the case of synchronous operation calls, the variable *awaitReturnOf* will store the message identifier of that call message for which the return message is awaited. Other variables not shown here model details of message management and action scheduling.

The actions of the cTLA process type *Object* correspond directly to the actions of UML objects which are introduced in Sec. 2. Thus, a *callAction* performs the local state transition of a calling object in accordance to *cf.NextState* (this condition is relevant for all actions) and defines the value of that message which forwards the invocation to the callee (action parameter *message*). Dependent on the mode of the call, the caller may be blocked (variable *awaitReturnOf*). The *receiveAction* enqueues received messages into the message queue *qu*. The *returnAction* and the *createAction*

send return messages resp. create messages. The further actions define state transitions which are only of local interest.

The global model is defined by the process type *GlobalSystem* outlined in Fig. 5. In the array *obs*, it contains processes of type *Object* for all ever existing object instances. Each array component is parameterized by the *ClassFrame* of an object class and we assume that for each object class there exist inifinitely many array components. The actions of *GlobalSystem* conjoin real actions and stuttering steps of the *Object* process instances. There are two sorts of actions, message transfer actions and local actions. Message transfer actions (like *operationCall*) are a conjunction of a sending action of a sender process, of a receiving action of the receiver, and of stuttering steps of all others. Local actions (like *objectDestroy*) concern only one object. The corresponding process performs a real action while all others stutter.

## 6 FORMAL CONSTRAINT PROCESSES

The different diagrams of UML specifications are formally modelled by a series of constraint-oriented cTLA-processes. All constraint processes define actions which correspond in their names and basic parameters to the actions of the global formal model. This implicitly relates the constraint processes with each other. Formally, we can define an extended global model, which is a composition of the global formal model with all constraint processes and where in each case the equally named actions are conjoined to one system action. Due to the features of cTLA, any subsystem of the extended model can be used for formal verifications of system properties. In particular, subsystems may be used which do not contain the global formal model.

To discuss the formal modelling in more detail, we refer to the two example diagrams of Fig. 2 and Fig. 3. As described in Sec. 3, Fig. 2 shows a simple collaboration diagram. It references the relevant objects of the realization of the use case *"Perform a Withdrawal Operation"* and their links. It constrains the values and the ordering of the messages exchanged over the links. Accordingly, a corresponding cTLA constraint process will refer to the set of active use cases and impose conditions on values and ordering of messages between objects of the use case. Fig. 7 depicts the definition of the process type *CollaborationDiagramUnit* which models the collaboration diagram under following two assumptions formally. Firstly, we assume, that a process instance is introduced for each two objects which are relevant for the use case. The objects are addressed in the process type by means of the process parameters *O1* and *O2*. Secondly, a constraint process instance shall exist which manages the set of active use cases and introduces a corresponding parameter *activeUseCases* to the actions.

The variables of process type *CollaborationDiagramUnit* (Fig. 7) keep track of call messages (*actMessage*) and of blocking due to synchronous calls (*callerLocked*). The actions correspond to all system actions of the global formal model as mentioned above. Since the constraint process, however, shall constrain only those actions, which are related to the objects *O1*, *O2* and to the use case *myUseCase*, each action is a disjunction. Besides of disjunctive terms which apply real constraints (under condition $caller = O1 \wedge callee = O2 \wedge myUseCase \in activeUseCases$), each action has a term applying a stuttering step of the constraint process to those action occurrences which

```
PROCESS CollaborationDiagramUnit (O1, O2 : OId; myusecase : UseCase)
BODY
  VARIABLES
    actMessage : SUBSET(Message.Id); ! List of active messages
    callerLocked : {"yes","no"};     ! Is caller locked ?
  INIT actMessage = {} ∧ callerLocked = "no";
  ACTIONS
    operationCall (caller, callee : OId; message : Message;
                   mode : SyncMode; activeUseCases : SUBSET(UseCase) ) ≙
    ! If O1 is caller, O2 is callee, and myusecase is an active use case,
    ! only messages of type "Withdraw" may be send; message becomes
    ! active and caller is locked
      ( caller = O1 ∧ callee = O2 ∧
        myusecase ∈ activeUseCases ∧ callerLocked = "no" ∧
        ( ( message.operationname = "Withdraw" ∧ mode = "synchronized" ∧
            actMessage′ = actMessage ∪ {message.id} ∧
            callerLocked′ = "yes" ) ) ) ∨
    ! Otherwise process performs a stuttering step
      ( ( caller ≠ O1 ∨ callee ≠ O2 ∨ myusecase ≠ activeUseCases ) ∧
        actMessage′ = actMessage ∧ callerLocked′ =  callerLocked );

    operationReturn (caller, callee : OId; message : Message;
                     activeUseCases : SUBSET(UseCase) ) ≙
    ! If O2 is caller, O1 is callee, and myusecase is an active use case,
    ! only messages of type "Withdraw" may be returned; furthermore a
    ! message must be active; message becomes passive and caller is unlocked
      ( caller = O2 ∧ callee = O1 ∧
        myusecase ∈ activeUseCases ∧ message.id ∈ actMessage ∧
        ( ( message.operationname = "Withdraw" ∧
            actMessage′ = actMessage  {message.id} ∧
            callerLocked′ = "no" ) ) ) ∨
    ! Otherwise process performs a stuttering step
      ( ( caller ≠ O2 ∨ callee ≠ O1 ∨ myusecase ≠ activeUseCases ) ∧
        ( callerLocked = "no" ∨ caller ≠ O1 ∨ myusecase ≠ activeUseCases ) ∧
        actMessage′ = actMessage ∧ callerLocked′ =  callerLocked );

    ! Other actions are only enabled if they don′t occur in O1, if O1 is
    ! unlocked, or if myusecase is not an active Use Case
    objectCreate (creator : OId; activeUseCases : SUBSET(UseCase) ) ≙
      ( callerLocked = "no" ∨ creator ≠ O1 ∨ myusecase ≠ activeUseCases ) ∧
      actMessage′ = actMessage ∧ callerLocked′ =  callerLocked;
    objectDestroy (this : OId; activeUseCases : SUBSET(UseCase) ) ≙
      ( callerLocked = "no" ∨ this ≠ O1 ∨ myusecase ≠ activeUseCases ) ∧
      actMessage′ = actMessage ∧ callerLocked′ =  callerLocked;
    objectStart (this : OId; activeUseCases : SUBSET(UseCase) ) ≙
      ( callerLocked = "no" ∨ this ≠ O1 ∨ myusecase ≠ activeUseCases ) ∧
      actMessage′ = actMessage ∧ callerLocked′ =  callerLocked;
    localInvocation (this : OId; activeUseCases : SUBSET(UseCase) ) ≙
      ( callerLocked = "no" ∨ this ≠ O1 ∨ myusecase ≠ activeUseCases ) ∧
      actMessage′ = actMessage ∧ callerLocked′ =  callerLocked;
    internalAction (this : OId; activeUseCases : SUBSET(UseCase) ) ≙
      ( callerLocked = "no" ∨ this ≠ O1 ∨ myusecase ≠ activeUseCases ) ∧
      actMessage′ = actMessage ∧ callerLocked′ =  callerLocked;
END
```

Figure 7 Process type CollaborationDiagramUnit.

are irrelevant for the constraint. By this means, the constraint process expresses, that call messages between *O1* and *O2* have to call for the operation *Withdraw* in a synchronous mode. Moreover, no other message shall occur between the two objects, until the return of this call is exchanged.

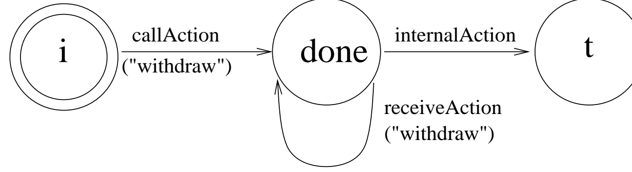


Figure 8 Equivalent transition system to Business Transaction state chart.

```
PROCESS BusinessTransaction (id : OId; myusecase : UseCase)
BODY
  VARIABLES
    state : {"i","done","t"}; ! actual process state
  INIT state = "i";
  ACTIONS
    callAction (caller : OId; message : Message;
                activeUseCases : SUBSET(UseCase) ) ≙
      ( state = "i" ∧ myusecase ∈ activeUseCases ∧ id = caller ∧
        message.operationname = "Withdraw" ∧ state′ = "done" ) ∨
      ( ( state ≠ "i" ∨ myusecase ∉ activeUseCases ∨
          id ≠ caller ∨ message.operationname ≠ "Withdraw" ) ∧
        state′ = state );

    receiveAction (callee : Oid; message : Message;
                   activeUseCases : SUBSET(UseCase) ) ≙
      ( state = "done" ∧ myusecase ∈ activeUseCases ∧ id = callee ∧
        message.operationname = "Withdraw" ∧ state′ = "done" ) ∨
      ( ( state ≠ "done" ∨ myusecase ∉ activeUseCases ∨
          id ≠ callee ∨ message.operationname ≠ "Withdraw" ) ∧
        state′ = state );

    internalAction (this : OId; activeUseCases : SUBSET(UseCase) ) ≙
      ( state = "done" ∧ myusecase ∈ activeUseCases ∧
        id = this ∧ state′ = "t" ) ∨
      ( ( state ≠ "done" ∨ myusecase ∉ activeUseCases ∨ id ≠ this ) ∧
        state′ = state ∧ blocked′ = blocked);
END
```

Figure 9 Process Type BusinessTransaction.

The second example refers to state chart diagrams. We perform the transformation of a state chart diagram to a corresponding cTLA-process type in two steps. Firstly, the state chart which may dispose of nested states and may define transitions labelled by action sequences, is transformed to a state transition system with a simple set of states and atomic transition labels. Secondly, we design a constraint process type which corresponds to the state transition system. With respect to the first step we refer to existing approaches (e.g., [8, 15]). So we can focus on an outline of the second step here. We use the very simple example of the state chart which is contained in the right side of Fig. 3. It can be translated to the small state transition system shown in Fig. 8. It expresses, that a corresponding object can at first perform a call action in order to call the operation *Withdraw*. Thereafter it can optionally receive a return message and terminate by an internal action.

Fig. 9 shows the corresponding cTLA process type *BusinessTransaction*. Again, we assume that for each relevant object *id* of the use case an instance of the process

type is introduced and that the active use cases are managed in accordance to a further constraint process. Like the state transition system in Fig. 8, the process type *BusinessTransaction* is also very simple. The three-valued data type of its one variable *state* directly represents the state nodes of Fig. 8. Since only the actions *callAction*, *receiveAction*, and *internalAction* are constrained, the process only contains definitions of these actions. Definitions of other actions (e.g. object *Create*) would be fully equivalent to stuttering steps and therefore are omitted. The constrained actions, as in process type *CollaborationDiagramUnit*, again are defined by disjunctions of terms where the last term describes stuttering of the process for non-relevant action occurrences. For instance, call actions are not relevant if the constraint process is not in state *i* or if the relevant use case *myUseCase* is not active or if the caller is not the constrained object *id*. In the complementary cases, a call action is constrained by our process. It has to call the operation *Withdraw*. Moreover, the value of the variable *state* changes to *"done"* in accordance with the state diagram.

## 7 CONCLUDING REMARKS

We proposed a method for the formal modelling of behavioral UML diagrams by means of constraint-oriented cTLA-processes. Thereby formal semantics for the diagrams are defined. Unlike other approaches, which define isolated models for the different diagrams of a project, in our approach, all constraint processes which concern the same object system have a common context. Though this common context is usually very complex, our method is practicable, since we only need to know the context structure and do not need to define it in detail. Due to the common context (and due to the superposition character of cTLA's process composition operation) it is now possible, to define formal models of sets of UML diagrams which in fact are very valuable for the early formal analysis of object system designs. We made the experience, that interesting system properties are mostly not consequences of single isolated diagrams but can be verified only if one considers a combination. Moreover, as a rule, the combinations (i.e., the subsystems of corresponding cTLA constraint processes) are of manageable complexity, so that it is practically possible to reason about interesting invariants of complex system designs. Present work aims to an application of these results which provides theorems connecting occurrences of behavioral design pattern instances with interesting system properties. Moreover, real-time extensions are in preparation in order to support the verification of hard quality of service requirements.

## References


[1] Back, R. J. R. and Kurki-Suonio, R. *Decentralisation of process nets with a centralized control*. Distributed Computing (3). pages 72–87, 1989.

[2] Breu, R., Grosu, R., Huber, F., Rumpe, B., and Schwerin, W. *Systems, Views and Models of UML*. In Schader, M. and Korthaus A. (eds.), The Unified Modeling Language, Technical Aspects and Applications. Physica Verlag, 1998.

[3] Chandy, K. M. and Misra, J. *Parallel Program Design - A Foundation*. Addison-Wesley, 1988.

[4] France, R., Bruel, J., Larrondo-Petrie, M., and Shroff, M. *Exploring The Semantics of UML Type Structures with Z.* In Derrick, J. (ed.) Formal Methods for Open Object-based Distributed Systems, pages 247–257, Chapman & Hall, 1997.

[5] Harel, D. *Statecharts: A Visual Formalism For Complex Systems*. Science of Computer Programming (8):231–274, 1987.

[6] Herrmann, P. and Krumm, H. *Compositional Specification and Verification of High-Speed Transfer Protocols*. In Vuong, S. T. and Chanson, S. T. (eds.), Protocol Specification, Testing, and Verification XIV, pages 339–346. Vancouver. B.C., IFIP, Chapman & Hall, 1994.

[7] Herrmann, P. and Krumm, H. *Re-Usable Verification Elements for High-Speed Transfer Protocol Configurations*. In Dembiński, P. and Średniawa, M. (eds.), Protocol Specification, Testing, and Verification XV, pages 171–186. Warsaw, IFIP, Chapman & Hall, 1995.

[8] Hooman, J., Ramesh, S., and de Roever, W.-P. *A compositional axiomatization of Statecharts*. Theoretical Computer Science (101):289–335, 1992.

[9] Kurki-Suonio, R. and Mikkonen, T. *Liberating object-oriented modeling from programming-level abstractions*. In ECOOP'97 Workshop Rd., Springer, 1997.

[10] ISO. *Lotos: Language for the temporal ordering specification of observational behavior*. International Standard ISO/IS 8807, 1989.

[11] Mester, A. and Krumm, H. *Composition and Refinement Mapping based Construction of Distributed Applications*. In Proceedings of the Workshop on Tools and Algorithms for the Construction and Analysis of Systems. Aarhus, Denmark, BRICS, 1995.

[12] Paech, B. and Rumpe, B. *State Based Service Descriptions*. In Derrick, J. (ed.) Formal Methods for Open Object-based Distributed Systems, pages 293–302, Chapman & Hall, 1997.

[13] Lamport, L. *The Temporal Logic of Actions*. In ACM Transactions on Programming Languages and Systems. 16(3):872–923, May 1994.

[14] The UML Group. *UML Semantics. Version 1.1*. Rational Software Corporation. Santa Clara, CA-95051, USA, July 1997.

[15] Uselton, A. C. and Smolka, S. A. *A Compositional Semantics for Statecharts using Labeled Transition System*. Lecture Notes in Computer Science 836, pages 2–17, Springer-Verlag, 1994.

[16] Vissers, A. C., Scollo, G., van Sinderen, M., and Brinksma, E. *Specification styles in distributed systems design and verification*. Theoretical Computer Science (89):179–206, 1991.

[17] Weber, M. *Systematic Design of Embedded Control Systems*. GMD-Bericht Nr.283, R.Oldenbourg Verlag, 1997.

## Biography

**Günter Graw** received the diploma degree in computer science in 1993. After working in industry, he became a graduate student at the University of Dortmund in 1997.
**Peter Herrmann** works as a researcher at the University of Dortmund, where he received the Ph.D. degree in computer science in 1997.
**Heiko Krumm** is professor for computer networks and distributed systems at the University of Dortmund since 1990.