In Proceedings of the *Workshop on Semantics of Objects as Processes (SOAP'99)*, NS-99-2, pages 7–22, BRICS, 1999

Composing object-oriented specifications and verifications with cTLA

Günter Graw, Peter Herrmann, Heiko Krumm

Dept. of Computer Science, Dortmund University, D-44221 Dortmund, Germany

Internet: {graw|herrmann|krumm}@ls4.cs.uni-dortmund.de

Abstract

In order to support formally correctness preserving refinement steps of object-oriented system designs, we refer at one hand to the practically well-accepted Unified Modelling Language (UML) and at the other hand to L. Lamport's Temporal Logic of Actions (TLA) which supports concise and precise notions of properties of dynamic behaviours and corresponding proof techniques. We apply cTLA which is an extension of TLA and supports the modular definition of process types. Moreover, in cTLA process composition has the character of superposition which facilitates the modular transformation of UML diagrams to corresponding formal cTLA process system definitions and their structured verification. We exemplify transformation and formal verification. Moreover we outline the application of this method for the establishment of domain-specific specification frameworks which can directly support the UML-based correct design of OO-systems.

1 Introduction

Meanwhile, the practical design of object-oriented application systems is mostly based on the Unified Modelling Language UML [21]. Systems are modeled and described by a series of UML diagrams where each diagram corresponds to a partial view of a system and concentrates on certain property types and aspects. So, class diagrams describe the static class structure. Use case diagrams are devoted to specific utilizations and the objects instances which are responsible for their realization. Collaboration diagrams focus on the partners and interactions of specific cooperation relations. Statechart diagrams describe the behaviour of object instances. While several approaches exist which assign formal semantics to the different diagram types (e.g., [16]), usually UML-based designs are non-formal. Since the diagrams support intuitive interpretations, the designers easily understand their pragmatical meanings without reference to formal models. Therefore, often formal designs are not desirable, especially, since the development and analysis of formal models would introduce considerable additional costs. Furthermore, for many interesting formal design checks separate formal models of single diagrams would not suffice. Instead, very complex models of diagram combinations would be necessary which model a set of diagrams in context with each other in order to cover interrelations.

The design of critical systems, however, can essentially profit from formal verifications. We expect benefits at least from formal checks of those functions, aspects, and properties

which are as well crucial as their provision depends on complex and not easy-to-understand mechanisms. In particular, aspects of the design of dynamic object system configuration at runtime, of concurrent execution threads, of combined behaviour of object instances, and of object interactions are inherently complex and difficult to master without formal support. In order to support formal modelling and analysis of partial aspects of UML-based object-oriented system designs, especially with respect to questions of concurrency, object behaviour, and interactions, we developed transformations from UML diagrams to formal cTLA specifications [9] and underlying state transition system models.

cTLA is based on L. Lamports Temporal Logic of Actions (TLA) [17] and refers to the concepts of state transition systems, refinement mappings [1], and the separate definition of both safety and liveness properties. Unlike TLA, the cTLA composition principle is oriented at CCS [19] and Lotos [14] and applies the principle of superposition like DisCo [6]. In comparison with [2], the cTLA processes do not interact via shared variables but perform joint actions. This stateless way of interaction has different benefits. Especially constraint-oriented processes can be represented (cf. [23]) which are well suited for the diagrams of the UML. Furthermore, cTLA supports decompositional proofs. A system is the logical conjunction of its processes and the style conventions assure the absence of contradictions in the system formula. Thus, process properties are directly inherited to the system. The compositionality of cTLA supports the transformation of UML-based descriptions since each UML diagram of a system description can be modelled by a single cTLA process which contributes to the system as a whole in a well-defined way. For the analysis of properties of interrelations relatively small subsystems can be used comprising only those processes which influence the properties of special interest.

This paper shortly outlines our approach as a whole and concentrates on the formal verification of refinement steps where a step is represented by two sets of UML diagrams. The first set describes the starting point of the refinement which we call the abstract model. The second set specifies the result of the refinement by means of the so-called refined model. Both models can be transformed to cTLA. Thus, there are two corresponding systems of cTLA processes, the first describing a more abstract state transition system, the second describing a refined state transition system. The formal verification shall prove that the refined system has in fact all those safety and liveness properties which are required by the specification of the abstract system. Since TLA's formal refinement relation directly corresponds to this practically relevant notion of correct refinement, verifications can be performed on the basis of TLA where a refinement step is correct, exactly if the implication '*Refined System implies Abstract System*' can proved to be a valid TLA-formula cf. [17].

Of course, the transformation of UML diagrams to cTLA processes and the TLA-based verification introduces additional efforts. Therefore methods are of high interest which support correctness relations directly applying to UML specifications of abstract and refined systems. These objectives are similar to those of the pUML group whose members investigate diagrammatical transformation rules where a rule directly applies to an abstract diagram and transforms it to the refined diagram of a correct refinement [20]. Thus, the approach of pUML transformation rules is very ambitious and shall combine the advantages of correctness-preserving source-code transformations [4] with those of graphical specification and modelling support. With respect to the preservation of behavioural

properties of concurrent and distributed systems, however, we made the experience that general correctness-preserving transformation rules are very difficult to handle in the course of practical design processes. Moreover, the rules are accompanied by so-called application conditions. The correctness of a transformation is only assured if the application condition holds. Since many application conditions are relatively complex, efforts for their proofs are necessary which are comparable to that of a posterio verifications of freely designed refinements.

Under these considerations, our present work investigates another direction of direct refinement support. It follows up the framework approach of software development (cf. [15]) and translates it into the field of specification development. Consequently, we study special domains of application (e.g., protocol design [11], distributed control of chemical plants). Corresponding collections of specification modules and patterns for abstract and refined systems are under development. Moreover, the relations between those abstract and refined modules and patterns are investigated which correspond to correct refinement steps. The results are documented by a collection of theorems. The theorems are implications between refined system patterns and abstract patterns. In principle, their function is comparable to that of general correctness preserving transformation rules. Nevertheless, the theorems connect domain-specific specification patterns and therefore can provide direct application-specific design support.

In the remainder, we outline the formal specification language cTLA. Thereafter we address basic notions of dynamic behaviours of object systems and their representations in UML models. We describe the essentials of the transformation from UML diagrams to cTLA processes. From that, the TLA-based verification of refinements is discussed. Transformation and verification are exemplified by means of a small application scenario. Finally, we sketch our present work which is constructing a domain-specific specification framework for distributed control of chemical plants.

2 Compositional specification style cTLA

cTLA [10, 18] is based on Leslie Lamports Temporal Logic of Actions (TLA) [17] and supports the definition of parametrized process and system types. A specification of a simple process or a (sub)system is formed by instantiating a cTLA process type resp. system type. As in the formal description language Lotos [14], systems are composed from processes which interact by means of joint actions. Due to this method of composition, processes can model not only implementation parts but also logical system constraints (cf. [23]).

As an example of a cTLA process type we outline *Object* in Fig. 1 describing the behaviour of an UML object (cf. Sec. 4). In the process type header the name *Object* and the process parameters *cf*, *id*, and *class* are declared. The state variables (e.g., *state*, *lifecycle*, *qu*) model the process state. The set of initial states is descibed by the predicate `INIT`. State transitions are specified by means of actions. An action (e.g., *callAction*) is a predicate about action parameters (e.g., *receiver*), state variables describing the state before executing the action (e.g., *lifecycle*), and so-called primed state variables modelling the state after executing the action (e.g., *lificycle'*). Besides of state transitions specified

```
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 ; objState, objNextState : cf.State ;
                 message : Message ; mode : SyncMode ) ≙ ! send Call-message
      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 ; objState, objNextState : cf.State ;
                   message : Message ) ≙ ... ; ! send Return-message
    createAction ( receiver : OId ; objState, objNextState : cf.State ;
                   message : Message) ≙ ... ; ! send Create-message
    ...;
   END
```

Figure 1: Process type *Object*.

by actions, a process may perform stuttering steps where it does not change its state while the process environment performs a state transition.

The cTLA process type *Object* describes safety properties. Liveness constraints (cf. [3]) are described by additional weak or strong fairness assumptions forcing the execution of an action if it would be enabled for an infinite period of time otherwise. Weak fair actions (denoted by `WF : callAction`) are only required to execute if the action would otherwise be incessantly enabled while execution of strong fair actions (denoted by `SF : callAction`) is guaranteed even if the action is sometimes disabled. Unlike the definition of [3] and TLA, cTLA provides for conditional fairness assumptions in order to keep the compositionality of systems. A fair action has to execute only if otherwise infinitely many states exist where the action is enabled as well as its execution is tolerated by the environment.

Systems and subsystems are described as compositions of concurrent processes which encapsulate their state variables and change their local states according to the process actions. The vector of the process state variables represent the state of the entire system. System state transitions are described by system actions which are logical conjuncts of process actions and process stuttering steps. Since each process contributes to each system action by exactly one action or a stuttering step, concurrency is modeled by interleaving and the coupling of processes by joint actions. The action parameters are used to describe data transfer between processes.

```
PROCESS GlobalSystem ( cfs : [class → 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 ;
                   callerState, callerNextState,
                   calleeState, calleeNextState : State ;
                   message : Message ; mode : SyncMode ) ≙
      ! caller calls operation of callee
      obs[caller].callAction(callee,callerState,callerNextState,
                             message,mode) ∧
      obs[callee].receiveAction(calleeState,calleeNextState,message) ∧
      ∀ i ∈ OId \ {caller,callee} obs[i].Stutter ;
    operationReturn (...) ≙ ...; ! callee operation returns to caller
    objectCreate (...) ≙ ...; ! object sends create message
    ...;
  END
```

Figure 2: Process type *GlobalSystem*.

As an example Fig. 2 shows the system type *GlobalSystem* modelling a system of UML objects. The processes composing the system are listed in the section `PROCESSES`. For instance, *GlobalSystem* consists of *OId* many instances *obs[i]* of the process type *Object* (cf. Fig. 1). The system actions are listed in the section `ACTIONS`. In the example, the action *operationCall* models that the object *obs[caller]* calls an operation of the object *obs[callee]*. Therefore *obs[caller]* participates to *operationCall* by the process action *callAction* and *obs[callee]* by the process action *receiveAction*. The other processes participate to *operationCall* by stuttering steps. Data between the caller and the callee are described by the system action parameter *message*. During the execution of *operationCall*, the process action parameters *message* in *obs[caller].callAction* and *obs[callee].receiveAction* have to carry identical values.

cTLA facilitates the combination of different property types like safety and liveness. Thus, in the resource oriented specification style, all relevant aspects of a component can be described by a single process type. In the constraint-oriented specification style one can specify different aspects of a component by separate constraint processes. In order to support the modularity of verifications, however, liveness properties may be combined with models of the safety behaviour of the component's environment (cf. [11]).

3 Dynamic behaviour

Since we concentrate on the issues of concurrency and concurrent object interaction we give a short outline of the according UML concepts. 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 performing steps of execution changing the system state. The relevant state of a system depends on the set of currently existing objects and their control

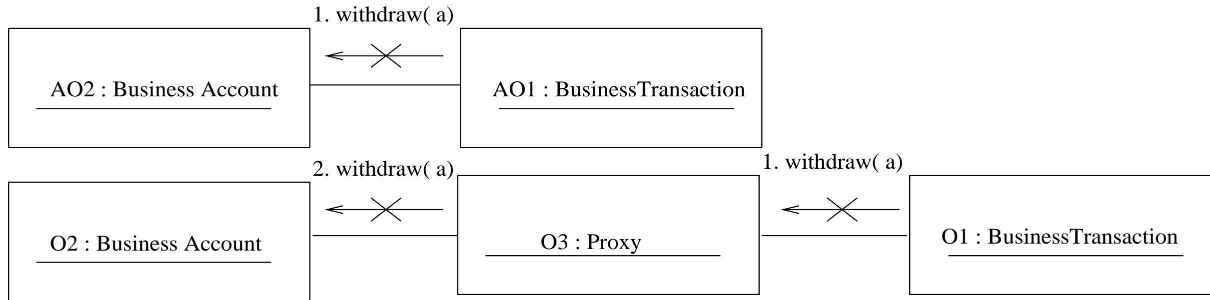

Figure 3: Collaboration diagram of the example system

and data (attribute values) states (cf. Fig. 1). The state of the object system as a whole identifies the set of currently existing objects and moreover contains the object states as components (see [9] for the according cTLA specification of the global system). The UML uses statechart, interaction (sequence and collaboration), and use case diagrams for the description of the dynamic behaviour. In the UML, an execution step with an object corresponds to an action which is modelled in a statechart. Actions may effect the local and foreign objects as well. There are several kinds of actions:

- A call action results in the invocation of an operation.
- Send actions result in the asynchronous sending of a signal.
- Create actions cause the creation of an instance of a class. They are not permitted to have a target object.
- By return actions a value or a set of values is returned to the caller.
- A terminate action results in the self-destruction of an object. It should not have parameters.
- Local invocation actions cause the local invocation of an operation without generating a call or signal event.
- Actions that are not previously defined are called uninterpreted actions.

Like the UML-metamodel [21] we assume run-to-completion semantics (RTC) for state machines which follows the idea that requests are processed in sequence one after the other. This assumption simplifies the synchronization of an object, since an incoming request is only processed, if the object has reached a stable state configuration. Communication between objects is specified by means of signal or operation requests. Objects communicate by means of operation (service provided for another object) requests if the calling object demands a service by the called object. A request is forwarded by a message instance which can carry a set of arguments. Operations can be called synchronously (sender is blocked, cross in message symbol) or asynchronously, which is modelled which is modelled in collaboration diagrams. The number which precedes the name of a message represents its order in an execution sequence.

4 Transformation

Since cTLA facilitates constraint-oriented specifications, the different diagrams of UML specifications can be modeled formally by a couple of individual cTLA-processes. Below, we will outline the transformation of UML collaboration diagrams and statecharts by means of a simple example specification. The idea of the example is that money is withdrawn from a business account in a business transaction. In the top of Fig. 3 a collaboration is shown which represents the withdrawal of money from the business account. A refined design of the example is presented in the bottom of Fig. 3. The refinement is manifested in the introduction of a new proxy object for the business account which is located in another address space. The statecharts of the according object classes are given in Fig. 4.

Collaboration diagrams are transformed to the cTLA process type *CollaborationDiagramUnit* shown in Fig. 5. We introduce a new process instance for each two objects which are relevant in the context of the according use case. The process parameters *O1* and *O2* in *CollaborationDiagramUnit* are used to address the corresponding process types. Moreover, a constraint process instance should manage the set of active use cases. This causes the introduction of a corresponding parameter *activeUseCases* to the actions. The state variable *actMessage* keeps track of call messages and *callerLocked* of blocking caused by synchronous calls (*callerLocked*). Since the process type *CollaborationDiagramUnit* shall constrain only those actions, which are related to the objects *O1*, *O2* and to the active use case *myUseCase*, each action is furnished with a term applying a stuttering step of the constraint process to those action occurrences which are irrelevant for the constraint (synchronous mode, no other message until termination.), except for disjunctive terms applying real constraints (under condition $caller = O1 \wedge callee = O2 \wedge myUseCase \in activeUseCases$).

The transformation of UML statechart diagrams to cTLA processes is performed in two steps. At first, a statechart which may contain nested states and transitions labelled by action sequences is transformed to an ordinary state transition system following the principles explained in [13, 22]. For instance, the statechart at the right side of Fig. 4 describing the BusinessTransaction *O1* is transformed to a simple state transition system

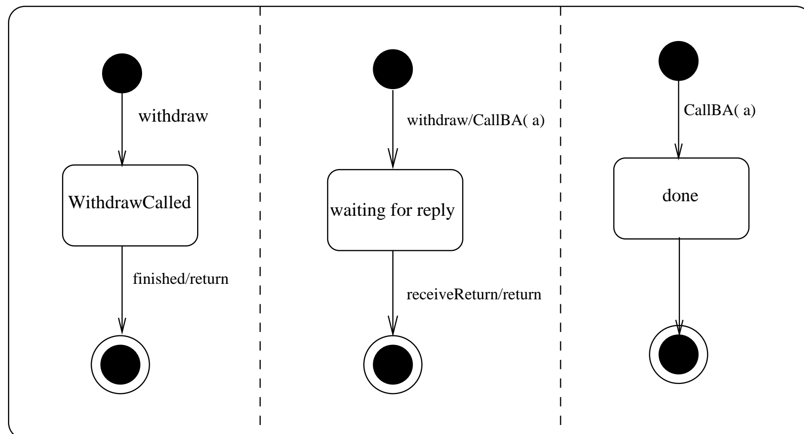

Figure 4: Statechart diagrams of the example system

```
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
```

Figure 5: Process type CollaborationDiagramUnit.

listed in Fig. 6.

In the second step the transition system is transformed to a cTLA process type. Since cTLA process types model state transitions in a direct way, this step is very simple. The process type *BusinessTransaction* (Fig. 7) contains the state variable *state* modelling the three states of the state transition system. The condition INIT specifies that "*i*" is the initial state and the actions *callAction*, *receiveAction*, and *internalAction* model the transitions. The process parameter *id* describes the object identifier while *myusecase* is used to manage the active use cases in accordance with a further constraint process.

Relations between UML diagrams are modeled in cTLA by means of process action

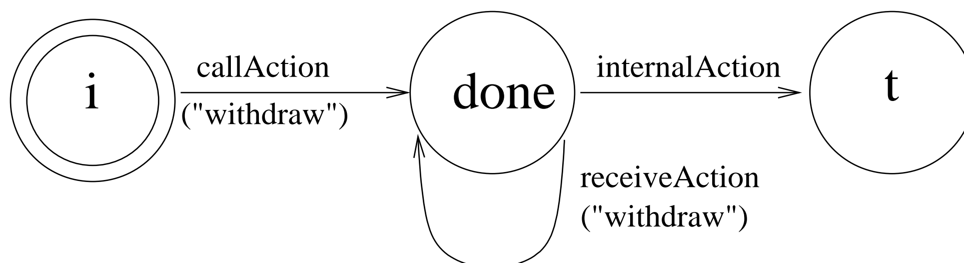

Figure 6: State transition system of the BusinessTransaction *O1*.

```
PROCESS BusinessTransaction (id : OId; myusecase : UseCase)
BODY
  VARIABLES
    state : {"i","wfr","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′ = "wfr" )  ∨
      ( ( state ≠ "i"  ∨  myusecase ∉ activeUseCases  ∨
          id ≠ caller  ∨  message.operationname ≠ "Withdraw" ) ∧
          state′ = state );
    receiveAction (callee : Oid; message : Message;
                   activeUseCases : SUBSET(UseCase) ) ≙
      ( state = "wfr" ∧ myusecase ∈ activeUseCases ∧ id = callee ∧
        message.operationname = "Withdraw" ∧ state′ = "wfr" )  ∨
      ( ( state ≠ "wfr"  ∨  myusecase ∉ activeUseCases  ∨
          id ≠ callee  ∨  message.operationname ≠ "Withdraw" ) ∧
          state′ = state );
    internalAction (this : OId; activeUseCases : SUBSET(UseCase) ) ≙ ...;
END
```

Figure 7: Process Type *BusinessTransaction*.

conjunctions. Assume that *O1 : BusinessTransaction* is the cTLA process specifying the example object $O1$ and *CollO1O3 : CollaborationDiagramUnit* the cTLA process describing the operation call *withdraw* in which $O1$ is the caller and $O3$ the callee. Since the operation call is triggered by $O1$ performing a *callAction*, the process actions *callAction* of cTLA process $O1$ and *operationCall* of *CollO1O3* are conjoined. Likewise, the process actions *returnAction* in $O1$ and *operationReturn* in *CollO1O3* are coupled. The action *internalAction* of $O1$ does not correspond to any collaboration diagram transitions and therefore is linked with a stuttering step of *CollO1O3*.

5 An example proof

Below we will outline the proof that the abstract system consisting of the business transaction object $AO1$ and the business account object $AO2$ is realized by a more detailed system consisting of $O1$, $O2$, and an additional proxy $O3$. The UML collaboration diagrams and statecharts are transformed into cTLA specifications according to Sec. 4. The proof utilizes the compositionality of cTLA. It can be decomposed into three simpler proof steps. At first, we have to prove that the subsystem $SO_{1/3}$ consisting of the processes representing the statecharts of $O1$ and $O3$ (Fig. 4) composed with the process *CollO1O3* modelling the collaboration diagram unit connecting $O1$ with $O3$ (Fig. 3) fulfills the process representing the abstract business transaction object $AO1$. Secondly, we prove that the process representing $O2$ implies that implementing $AO2$. Finally, we have to verify that the process describing the collaboration between $O3$ and $O2$ realizes that

representing the collaboration between $AO1$ and $AO2$.

Here, we will sketch only the first proof corresponding to the verification of the implication $SO_{1/3} \Rightarrow AO1$ which is performed as a regular TLA refinement proof (cf. [17]). In order to compare the two state spaces of $SO_{1/3}$ and $AO1$, we define a mapping between them, the so-called refinement mapping:

$$\begin{aligned} RM \widehat{=}\ & O3.state = "i" \rightarrow AO1.state = "i" \\ & O1.state = "t" \rightarrow AO1.state = "t" \\ & otherwise \rightarrow AO1.state = "withdrawCalled" \end{aligned}$$

Instead of $AO1$ we use the equivalent process $\overline{AO1}$ for the proof where the local variable $AO1.state$ is replaced by variables of $O1$ and $O3$ according to RM.

Firstly, we have to verify that all initial states of $SO_{1/3}$ are also initial states of $\overline{AO1}$. Since in the initial states of both processes the equation $O3.state = "i"$ holds, this proof is trivial. Secondly, we have to prove that each action of $SO_{1/3}$ implies either an action or a stuttering step in $\overline{AO1}$. This proof, however, cannot be performed directly. Before, we have to prove that the following formula I is an invariant of the subsystem $SO_{1/3}$:

$$\begin{aligned} I \widehat{=}\ & O1.state = "i" \Rightarrow O3.state = "i" \wedge O1.state = "t" \Rightarrow O3.state = "t" \wedge \\ & O3.state = "waitingForReply" \Rightarrow O1.state = "withdrawCalled" \wedge \\ & "withdraw" \in O3.qu \Rightarrow (O1.state = "withdrawCalled" \wedge O3.state = "i") \wedge \\ & "withdrawReturn" \in O1.qu \Rightarrow (O1.state = "withdrawCalled" \wedge O3.state = "t") \end{aligned}$$

The invariant proof is performed by checking that I holds initially and is preserved by all actions of $SO_{1/3}$.

Using the proven fact, that I holds before and after execution of any action in $SO_{1/3}$, we can now verify that the actions of $SO_{1/3}$ correspond to actions or stuttering steps of $\overline{AO1}$. As an example we outline that the action T changing the state *state* of $O1$ from *withdrawCalled* to t implies the action $\overline{T}$ of $\overline{AO1}$ changing the state $O1.state \neq "t" \wedge O3.state \neq "i"$ ($AO1.state = "withdrawCalled"$) to $O1.state = "t"$. T is a joint action conjoining the actions *receiveAction* of $O1$ and *operationReturn* of the collaboration between $O1$ and $O3$. It can only be executed if the message "*withdrawReturn*" is in the message queue $O1.qu$ of $O1$. This implies that due to the last conjunct of I the condition $O1.state = "withdrawCalled" \wedge O3.state = "t"$ holds before executing T. Thus, the enabling condition of T implies the enabling condition of $\overline{T}$. After the execution the condition $O1.state = "t"$ holds as well in $SO_{1/3}$ as in $\overline{AO1}$. Therefore, the effect of T implies the effect of $\overline{T}$, too, and T implies $\overline{T}$ as a whole. Likewise, all actions of $SO_{1/3}$ are proven.

6 Verification with cTLA patterns

In this section we focus on the UML descriptions of properties of a software product on different levels of abstraction and the correctness of these descriptions. Therefore we introduce two models on different levels of abstraction which stem from the software life cycle(requirements engineering, design) of a given product. These are:

- The abstract software model (ASM) serves as interface between application engineering and software development. It models the structuring of the software parts of the system into logical components. It is a result of analysis activities performed during the requirements engineering of a software product which typically bases on knowledge from previously performed domain engineering.
- The concrete software model (CSM) is a refinement of the ASM. It structures the software into implementation-oriented components. It explicitly refers to distribution and network communication, to fault- tolerant mechanisms and performance optimisation as well as to the allocation and management of resources.

Both models are described in terms of patterns which have currently a growing impact on software development. On the one hand there are the well- known design patterns (e.g. given in [8]). On the other hand, analysis patterns have to be defined for each individual domain on their own applying techniques for domain engineering. Analysis patterns have been applied by Fowler [7] who has found and applied them in several industrial projects. Analysis patterns are described by the terms and concepts of an application domain. Taking these concepts, it is obvious that analysis patterns are applied in the ASM while design patterns are used in the CSM. Furthermore, some practitioners claim that there exist relationships between collections of patterns which might be expressed in a so called system of patterns [5].

Now we concentrate on the verification of a CSM-level specification against the abstract requirements expressed by an ASM-level specification. We have to prove that the CSM specification implements the relevant properties of the ASM specification. For that purpose comprehensive and compatible formal models of the dynamic semantics of both, the ASM and the CSM model, are needed. Moreover, one needs the formal inference system TLA to perform these proofs. While cTLA is well-suited to the formal modelling of highly structured systems, there is a very high complexity when ASM and CSM specifications of practical systems are transformed to cTLA. As previously stated, the statecharts are translated to process behaviour descriptions. The interaction diagrams and the activity diagrams (used for the modelling of synchronisation aspects) are translated to configuration and process coupling descriptions. To model the dynamic creation of objects, infinite state space structures (i.e., for each object type an infinite array of object instances) are used which are accompanied with an explicit state representation of the current existence of an object. In general, the operations of objects can be executed concurrently and there is a wide spectrum of object interaction mechanisms and synchronisation methods. Since state transition systems model behaviours by series of atomic transitions, the wide spectrum of object interactions induces a very fine granularity of atomicity. Thus, models with a very complex state space and with a very fine transition structure are needed in general. We think that these models are too difficult-to-understood to form a convenient basis of manageable formal verifications in practice. On the other hand, we are aware that the complexity of the models does not result from cTLA but is a direct consequence of the modelling power of UML-descriptions of object-oriented systems. Therefore, the approach has to be enhanced by additional concepts.

In order to render possible manageable formal proofs of practical systems, we utilise the proposed application of conceptual patterns in the ASM and of software design pat-

terns in the CSM for the verification, too. The benefits of patterns are twofold. On the one hand, patterns restrict system structures, the interactions, the concurrency, and the synchronisation of objects. The formal modelling recognises the restrictions and provides for less complex models which are more easy-to-understand since they directly correspond to application-oriented interaction schemes. On the other hand, there are logical relationships between conceptual patterns and software design patterns since a design pattern serves for the purpose of implementation of a conceptual pattern. This implementation relation between patterns of a system of patterns are formally modelled by the refinement relation of TLA, i.e., there exist valid implications from design patterns to conceptual patterns. In connection with the modularity and the genericity features of cTLA, theorems are stated which correspond directly to the logical relationships between patterns. E.g. in the example proof listed in section 5 there is a theorem which states the refinement relationship from a proxy pattern and a refined controller to the more abstract analysis pattern controller. These theorems can easily be instantiated to represent the particular refinement relations of a specific practical project.

In the domain of communication protocols comprehensive libraries of patterns and theorems are already established [11] and the experience showed that even complex practical protocols can be verified by means of theorems only, i.e., in order to verify a protocol it was not necessary to perform basic TLA deductions since all necessary implications of the proofs were instantiated from theorems [12].

7 Conclusion

We reported on present work which aims to the establishment of domain-specific specification frameworks for the object-oriented and pattern-based design of concurrent and distributed software systems. In particular, the frameworks will supply theorems which describe patterns of correct refinements and facilitate formal verification enormously since theorems can replace nearly all complex original proofs of verifications. Our report concentrated on the formal background of theorems which is given by transformations of UML diagrams to modular cTLA specifications enabling the application of TLA-based proof methods. According to this procedure the theorems of the specification frameworks under development are proven. Besides of our former work supporting the cTLA-based formal specification and verification of communication protocols, there is additional work the specification framework approach is related to. So, meanwhile extensions of cTLA exist which support the handling of real-time and continuous properties. Under application of these extensions already several hazard analysis and safety proofs for chemical plants were accomplished.

References

[1] M. Abadi and L. Lamport. The Existence of Refinement Mappings. *Theoretical Computer Science*, 82(2):253–284, May 1991.

[2] M. Abadi and L. Lamport. Composing specifications. *ACM Transactions on Programming Languages and Systems*, 15(1):73–132, Jan. 1993.

[3] B. Alpern and F. B. Schneider. Defining liveness. *Information Processing Letters*, 21:181–185, 1985.

[4] The CIP Language Group: The Munich Project CIP Volume I: The Wide Spectrum Language CIP-L. Lecture Notes in Computer Science 183 : Springer 1985

[5] F. Buschmann, R. Meunier, H. Rohnert Pattern Oriented Software Architecture : A System of Patterns. Addison-Wesley, 1996.

[6] R. J. R. Back and R. Kurki-Suonio. Decentralization of process nets with a centralized control. *Distributed Computing*, (3):73-78, 1989.

[7] M. Fowler. Analysis Patterns : Reusable Object Models. Addison-Wesley, 1996.

[8] E. Gamma, R. Helm, R. Johnson, J. Vlissides. Design Patterns: Elements of Reusable Object Oriented Software. Addison-Wesley, 1994.

[9] G. Graw, P. Herrmann, and H. Krumm. Constraint-Oriented Formal Modelling of OO-Systems. To appear in: *Second IFIP WG 6.1 International Working Conference on Distributed Applications and Interoperable Systems (DAIS 99)*, Helsinki, June 1999. Kluwer Academic Publisher.

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

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

[12] P. Herrmann and H. Krumm. Modular Specification and Verification of XTP. *Telecommunication Systems* 9(2):207-221, 1998.

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

[14] ISO. *LOTOS: Language for the temporal ordering specification of observational behaviour*, International Standard ISO 8807 edition, 1989.

[15] R. Johnson and B. Foote. Designing reusable classes. *The Journal of Object-Oriented Programming*, 1(2):22–35, 1988.

[16] K.C.Lano and A.S.Evans. Rigorous Development in UML. In ETAPS'99, FASE workshop. LNCS, 1999.

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

[18] A. Mester and H. Krumm. 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, 1995. BRICS.

[19] R. Milner. *A Calculus for Communicating Systems.* Number 92 in Lecture Notes in Computer Science. Springer, Berlin, 1980.

[20] The pUML group. http://www.cs.york.ac.uk/puml/

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

[22] A. C. Uselton and S. A. Smolka. A Compositional Semantics for Statecharts using Labeled Transition Systems. In B. Johnsson and J. Parrow, editors, *CONCUR'94: Concurrency Theory*, number 836 in Lecture Notes in Computer Science, pages 2–17. Springer-Verlag, 1994.

[23] C. A. Vissers, G. Scollo, and M. van Sinderen. Architecture and specification style in formal descriptions of distributed systems. In S. Agarwal and K. Sabnani, editors, *Protocol Specification, Testing and Verification*, volume VIII, pages 189–204, Elesevier, 1988. IFIP.