Object Composition: A Case Study

Dunia Ramazani , Discreet Logic (Canada)
Gregor v. Bochmann, University of Ottawa (Canada)

Abstract:
In Software engineering, experimentation is a necessary adjunct to process improvement. Objective and meaningful case studies can help us understand particular object-oriented notations, methods, or languages. In [Rama96b], we have presented an object-oriented method, called eXtended Object Modelling Technique, XOMT for shorthand. In this paper, we modify the developmental approach used in XOMT to include the specification of the behavior of composite objects based on synchronous interactions. The new developmental process consists of describing the application structure in terms of objects and associations between these objects. This includes the description of composite objects as proposed in XOMT. Object associations are then further refined by describing the object interactions that occur in the context of these associations. Object and association behaviors are specified in CSL, a specification language based on rendezvous interaction. CSL specifications can be translated in TLA. This adds a reasoning capability to the development process. The translation into TLA is motivated by the existence of a wide variety of specification and verification tools for TLA.

Keywords: Case study, CSL, Formal reasoning, Object composition, OMT, TLA, XOMT.

1. Introduction

We have been involved in the specification of engineering and telecommunications applications. In these applications, there is a predominance of composite objects, i.e. objects consisting of other objects. We have observed that the way the structure of these objects influence their behavior can play a key role in the process of specifying such objects [Rama95b]. To take this aspect into consideration throughout the application development process, we have defined a conceptual framework for the specification of composite objects. This framework proposes a new approach to the description of composite objects based on the relation between the structure and the behavior of these objects. The framework has been integrated within OMT [Rumb91]. Its integration within OMT leads to a new method called eXtended OMT (XOMT). Details on these experiences can be found in [Rama96b].

Now, we believe that the proposed approach to specification of composite objects has reached the status where it can be (1) shared with other practitioners, and (2) used in the industry. In Software engineering, experimentation is a necessary adjunct to process improvement. Objective and meaningful case studies can help us understand particular object-oriented notations, methods, or languages. In [Rama96b], we have presented an object-oriented method, called eXtended Object Modelling Technique, XOMT for shorthand. The presentation focuses on the differences and similarities between the two methods. The paper was aimed at presenting the concepts and principles underlying XOMT and not to be some kind of recipe and bunch of guidelines helping the specifier when he uses XOMT for modeling systems with composite objects. We have designed a new specification language for capturing object composition, namely CSL which stands for Composition Specification Language. This work shows how these latter results can be combined with XOMT. A specification case study is used for conveying our ideas, demonstrating the usability of the results in the context of XOMT.

1.1. Summary of our experience

In this paper, the integration of CSL is achieved at the expense of modifying the developmental approach used in XOMT. We departed from the OMT based development process to a more light process based on describing the application and object structures in terms of objects and associations between these objects. This follows the description of composite objects as proposed in XOMT. Object associations are then further refined by describing the object interactions that occur in the context of these associations. Object behaviors and association behaviors are specified in CSL which is translatable into TLA [Lamp94]. It adds a reasoning capability to the development process, i.e. we may then use TLA tools for verifying the specifications. The application selected in this case study is the specification of a lift system. According to many authors, it has many features of complex systems while being short enough to fit into a paper.

1.2. Organization of this paper

The paper is structured as follows. In Section 2, we begin by reviewing the essential concepts and principles of XOMT. It is followed by a summary of features provided by CSL including its formal semantics based on its translation into TLA. We then describe the development process which uses CSL. In Section 3, the development process is put in practice through the specification of a lift system. This includes the description of the application requirements, the informal specification supported by XOMT and the specifications in CSL including the translation in TLA as well as the reasoning capability. In Section 4, we discuss about the pro and cons of our development approach. We close the paper with the lessons learned in this experiment and we describe the future work.

2. Overview of XOMT and CSL

2.1. Concepts and Principles underlying XOMT

Roughly speaking, XOMT is OMT with a few add-ons allowing to specify composite objects such that there is a linkage between the structure and the behavior of these objects. In XOMT, properties of composite objects are classified into inherent, aggregate, and emergent. A composite object is an object with an internal structure which consists of the components (type and number of instances within the composite object), and the interconnections including the dynamic interactions between the components of the composite object. The linkage between component properties and composite object properties is established by distinguishing inherent properties, i.e. properties of the composite object which semantics is provided by properties of its components, from aggregate properties, properties of the composite object obtained by combining the properties of its components using aggregation mechanisms, from emergent properties which are properties of the composite object which do not depend on component properties. The presence of inherent and aggregate properties distinguishes a composite object from a simple one.

This new classification requires notational changes to object models in order to capture visibility and hiding of components, promotion of component properties to the status of composite object properties, and aggregation mechanisms used to combine the component properties. Other areas of improvement of OMT include the specification of the communication between objects, especially the communication between the components of a given composition. A more abstract interaction mechanism based on behavior constrainment is proposed, namely Contract specifications where behavioral interactions are expressed in a more abstract way so that the description does not introduce implementation bias. It is achieved by constraining, through predicates, the behavior of the interacting objects. Contracts are related to object associations abstracting the interactions between the classes. They represent the concurrent composition of the participant statecharts (individual behaviors) such that the behavior of the participants conforms to the constraints explicitly stated in the contracts. Constraints are expressed using a combination of first order predicate logic and OMT constraints.

2.2. Steps and notation in XOMT

The description of composite objects proceeds in three steps. The configuration describes the structure of the composite in terms of components, associations and behavioral interactions among these. The next step, juxtaposition shows how the composite is linked to its components through the aggregation association, and inherent and aggregate properties. The last step, emergence concerns the specification of emergent properties. Aggregation association is represented like in OMT. Visibility/hiding of components is specified by having the component class represented respectively with doubled and simple framed rectangles. The attributes and operations of composite are classified using three separate rectangles. Inherent, aggregate, and emergent attributes and operations are represented using respectively inherent, aggregate, and emergent rectangles, that is three rectangles in place of one, like for classes in OMT. Inherent associations are of two kinds, those resulting from the visibility of components and those involving hidden components. The former are represented by associations crossing the boundary of the composite and ending at some visible component inside the composite. The latter are represented by associations ending at the boundary of the composite and continued within the composite by a dashed line to some hidden component. Aggregate and emergent associations are represented using associations ending at the boundary of the composite and respectively decorated with the annotations {A} and {E}. To describe the behavior, we use parallel composition of statecharts. This is represented by having the parallel statecharts separated by a dashed line while being enclosed in the composite statechart [Hare88]. Inherent behavior is represented by the statechart of visible components and a specific statechart (annotated with "promoted") for the behavior originating from hidden components. Aggregate and emergent behaviors are respectively specified using specific statecharts. The overall behavior of the composite object consists of parallel composition of inherent, aggregate, and emergent behavior.

2.3. Composition Specification Language (CSL)

Concepts and principles

CSL has its roots in the assumption that object interactions occur only over object associations. This assumption is grounded on the fact that object interactions portray collaborations between the involved objects. Intuitively, the terms and conditions of these collaborations must be stated and established somewhere. The object association is the preferred place where such terms and conditions may be layed down. In CSL, object interactions are called abstract events [Boch93b] and they correspond to joint-actions. They are characterized by:

(a) There is no asymmetric caller/callee relationship: It is not said which object makes the decision for the execution of an interaction.
(b) There may be more than two objects participating in a given interaction, i.e. multiparty interactions.
(c) Each participating object may impose certain conditions which must be satisfied when the interaction occurs. Each participating object may also define some local state changes that occur during the execution of the interaction.

It is important to note that the above concept of interaction does not include the notion of caller - callee. This means that it is not specified how an interaction is "triggered". Only the conditions for its occurrence are specified. This way of describing object interactions is abstract, and non deterministic, leaving such question as triggering of actions, their scheduling and the involved actors for a later, more detailed specification. Along with us, other authors [Cook94, DSou94, Jarv90] have also adopted a similar mechanism for describing object interactions. In particular, the concept of joint-action [Jarv90] is semantically equivalent to that of abstract event.

Once we are able to describe the interactions that occur in the context of object associations, we may use the same approach for describing the interactions between components in a composition. Composition in CSL is achieved by connecting the objects and then hiding certain abstract events which involve these objects. Using this approach, we are able to describe structures of compositions including the interactions that occur in the compositions.

In [Rama95a], we have shown that the structure of a given composition is linked to its behavior because the behavior of the composition is realized in terms of the behavior of its components and how these are interconnected. The approach proposed above can be extended to the description of the behavior of compositions in terms of the behavior of components while taking into account the structure of the compositions. This is achieved by hiding abstract events which occur in the composition, i.e. in the structure of the composite object. Hiding is used as an abstraction mechanism.

Notation

The notation proposed by CSL is portrayed below. In CSL, we assume that a simple object (non-composite) can be represented as a composite object which has no components, i.e. its structure is empty. Based on that assumption, we use a single template for specifying objects. Only object compositions have the "inherits" section which indicates the actions of the components which are also actions of the composition.

```
spec name
    constants name = value
    define typename : type;
    extends  $name_i$, .., $name_j$
    contains
        $name_k$ : spec $name_t$
        . .
        $name_p$: spec $name_s$
    abstract  events
        association  name
            event $ae_i$  = <abstract event definition>
                    . .
            event $ae_k$ = <abstract event definition>
        . .
        association  name
            event $ae_i$  = <abstract event definition>
                    . .
            event $ae_k$ = <abstract event definition>
    inherits
        $action_i$ from $name_i$.$action_k$
        . .
        $action_q$ from $name_q$.$action_l$
    local  variables
        <variable declaration>
    initial  conditions
        <assignment to local variables>
        <conditions on the components>
    invariants
        <predicates on the local variables>
        <predicates on the components>
    behavior
        action name (<parameter list>) = <action definition>
        . .
        action name (<parameter list>) = <action definition>
end spec name
```

Object template

The "constants" and "define" clauses are self-explanatory. They introduce constants and new types in the specification. The "extends" clause denotes specialization. Features of the specifications listed in the "extends" clause are augmented with new features introduced by the specification. Actions can be redefined, invariants more constrained, etc.

The "contains" clause indicates the structure of the composition. It lists the components, while the constraints on these components can be stated in the "invariants" clause. "abstract events" denote the abstract events occurring between the components. These abstract events are structured according to the associations between the objects inside the composition. "local variables" denote the states of the object or the composition. The clause "initial conditions" defines the initial state. It assigns the initial values to the local variables and may define the initial states of the components in the context of the composition. "invariants" are used to record the safety properties of the object, and its components.

The "behavior" clause defines the actions which are supported by the object. Actions may have parameters. The semantics of the behavior clause are: First the initial conditions are established, in any state whether an action occurs, or the local variables remain unchanged. The invariants must be always satisfied in the state before the execution of the action and in the state after the execution of the action.

This semantics provides a means for assessing if a specification is well-formed. At the level of an object, we assume interleaving concurrency between the actions supported by that object. For a composition, there is interleaving between its abstract events, and its actions.

Until now we have not said too much about actions. Actions are defined in terms of "enabling", "defined", and "changes" predicates. More precisely, we use a combination of mathematics and predicate logic to define the semantics of actions. Each action has the form :

```
action  action-name(parameters) =
                        enabled: <predicates>
                        defined: <predicates>
                        changes: <predicates>
```

The meaning of an action is:
(a) when "enabled" is satisfied and "defined" is also satisfied, then the action can be executed and "changes" will be true afterwards;
(b) when "enabled" is satisfied and "defined" is not satisfied, the action can be executed, but the result is undefined;
(c) when "enabled" is not satisfied, the action can not be executed.

In action definitions, the "defined" predicate indicates the hypotheses about the environment for the action.

An abstract event is the synchronization of two or more actions. It is specified by a list of actions which are synchronized and a constraint over the parameters of the synchronized actions. It has the form:

```
event  abstract event-name= local: <local variables>
                            actions: <list of actions>
                            constraint: <constraint between action parameters and local variables>
```

The abstract event is enabled in states where all its composing actions are enabled. It is disabled if at least one of its composing actions is disabled. The defined clause of actions involved in an abstract event are used to assess the validity of the abstract event definition. Because, the defined clause denotes the hypotheses about the environment. The hypotheses of one action have to be consistent with the definition of the other actio s involved in the abstract event. This is refered to as internal consistency of the definition of an interaction.

Formal semantics

To reason about CSL specifications, we need an underlying execution model as well as an underlying logic for proving properties. We have favored the translation of CSL specifications into TLA [Lamp94], an existing formal specification language. The choice of TLA (Temporal Logic of Actions) is motivated by its computational model which is based on interaction, its built-in notion of behavior including the temporal ordering of actions, and nonetheless the availability of a wide range of specification and verification tools [Mest94].

The main difference between CSL and TLA is on the semantics of actions. CSL actions have explicit parameters. *In addition, each CSL action indicates what are its assumptions about the environment since the objects are composed through their actions.* This semantics introduce possible undefined behavior for actions. The action also has an explicit enabling condition.

When translated into TLA, CSL action parameters are introduced as additional variables of the TLA specification. In order to represent the possible undefined situations in TLA, two approaches can be taken for translating the action semantics. They correspond to the interpretations given to undefined situations. One approach consists to view an undefined situation as a state where the entire specification becomes deadlocked. The other approach consists to assume that in an undefined situation, the specification is allowed to do anything it wants.

With have extended TLA with two keywords, namely **chaos** and **anything** which can be used in action definitions to represent undefined situations. Let consider $LV(a_i)$ which represents the set of local variables appearing in the CSL action a_i. V is the local variables of the CSL specification. $CH(a_i)$ is a subset of $LV(a_i)$. It denotes the local variables which are modified by the action a_i. $PAR(a_i)$ is the set of the parameters of the action a_i. It can be decomposed into two sets $PAR_{in}(a_i)$ for input parameters and $PAR_{out}(a_i)$ for output parameters. Recall that a CSL action is defined by three predicates, namely **enabled**, **defined**, **changes** which will be respectively referred to in the sequel by a_i.**enabled**, a_i.**defined**, a_i.**changes**. The translation of the CSL action a_i into TLA is as follows.

Chaos semantics:

$$\forall\ LV(a_i) : \forall\ PAR(a_i): \quad a_i.\mathbf{enabled} \wedge ((a_i.\mathbf{defined} \Rightarrow (a_i.\mathbf{changes} \wedge \mathbf{unchanged}((V - CH(a_i)) \cup PAR_{in}(a_i)))) \vee (\neg a_i.\mathbf{defined} \Rightarrow \mathbf{chaos}))$$

where **chaos** denotes $\forall\ CH_1(a_i), \exists\ PAR'_{out}(a_i): (CH'(a_i) \neq CH_1(a_i)) \wedge (PAR_{out}(a_i) \neq PAR'_{out}(a_i))$.

Anything semantics:

$$\forall\ LV(a_i) : \forall\ PAR(a_i): \quad a_i.\mathbf{enabled} \wedge ((a_i.\mathbf{defined} \Rightarrow (a_i.\mathbf{changes} \wedge \mathbf{unchanged}((LV(a_i) - CH(a_i)) \cup PAR_{in}(a_i)))) \vee (\neg a_i.\mathbf{defined} \Rightarrow \mathbf{anything}))$$

where **anything** denotes $\exists\ CH_1(a_i), \exists\ PAR_{out}(a_i): (CH'(a_i) = CH_1(a_i)) \wedge (\neg PAR_{out}(a_i) = PAR'_{out}(a_i))$.

In this translation, $CH'(a_i)$ and $PAR'_{out}(a_i)$ represent respectively the values after the execution of the action a_i of the local variables and the action parameters which are modified by that action. Notice that explicit TLA **unchanged** predicates are added in order to indicate which variables remain unchanged by the action. Let the notation **TLA-action**(a_i) denotes the resulting TLA action when a CSL action a_i is translated into TLA. The two possible translations described above lead to two possible meanings of the TLA action operator *Enabled* . In TLA, the operator *Enabled*, e.g. *Enabled* a, is used to denote the fact that the action a is enabled, i.e. it may be executed. With the **chaos** interpretation, *Enabled* **TLA-action**(a_i) corresponds to the conjunction of a_i.**enabled** and a_i.**defined**, while with the **anything** interpretation, *Enabled* **TLA-action**(a_i) is equivalent to a_i.**enabled**.

With respect to the behavior, TLA and CSL have the same semantics, i.e. the initial conditions hold first and then either an action is executed or the specification variables remain unchanged. Therefore, the translation is one to one for the object behavior.

Once the CSL actions are translated into TLA, abstract events can be also be translated into TLA. We take advantage of conjunction of actions in TLA to define the semantics of abstract events. Abstract events are represented by the conjunction of the involved actions and the constraint between the parameters of these actions. Local variables of an abstract event are introduced using TLA temporal existential quantification $\exists$ which denotes hiding of variables in TLA. Hiding of abstract events in a composition is translated by hiding the parameters of the actions involved in these abstract events. Specialization in TLA is based on the existence of a refinement mapping between the specifications. It assumes (execution) steps simulation between the specifications. Steps simulation is general enough to cover CSL specialization. In other words, let **TLA-spec**(M) denotes the CSL specification M translated into TLA, if M and N which are CSL specifications are such that N is a specialization of M, then **TLA-spec**(N) $\Rightarrow$ **TLA-spec**(M). But the converse is not necessarily true since it is not the case that any TLA specification has a corresponding CSL specification.

The intuition behind CSL composition rule is: given a CSL composition of M_1 and M_2, if

1) we have internal consistency in that composition, i.e. each composonent behaves well in an environment including the other components with respect to the abstract events relating the latter components to the former;
2) and the composition is a specialization of another high-level specification M
then the composition of M_1 and M_2 is an implementation of M.

This means once internal consistency in a composition is shown, specialization suffices to have a sound composition rule. Notice that internal consistency can be demonstrated by showing that none of the abstract events lead to **chaos** or **anything** depending on the interpretation which is adopted. In fact, internal consistency of a composition, once the CSL specifications are translated into TLA, can be demonstrated as an invariant of the resulting TLA specification. This is achieved using the TLA rule INV1 [Lamp94]. Specialization is proven at the level of the TLA specifications by establishing a refinement mapping between the specifications. This has for consequence that, once translated, CSL compositional reasoning can b[redacted] achieved at the level of the resulting TLA specification using TLA rules.

To reason about speci[redacted]ns w[redacted]h [redacted]clu[redacted] explicit assumptions about the environment, TLA introduces the opera[redacted]r [redacted][A[redacted]d9[redacted]A speci[redacted]ation E ±> M is a specification of an open system where E denotes the[redacted]umpti[redacted]s ab[redacted] the be[redacted]vior of the environment. It is given by the canonical TLA formula Init[redacted][Action[redacted]I denotes [redacted]e behavior of the component, it is given by the formula Init$_M$ [redacted][Actions$_M$] [redacted]$_M$ where F$_M$ [redacted]epresents the fairness requirements. In CSL, the assumptions about the environment are at the level of actions, while in TLA, E ±> M specifications are introduced to take into account the assumptions about the environment at the level of the whole specification. This means undefined situations are dealed at the level of the whole specification, i.e. undefined situations have coarser granularity in TLA E ±> M specifications. Since these latter specifications can be translated into TLA, one may suggest that the resulting specifications should be comparable with the specifications obtained by translating CSL specifications into TLA. However, E ±> M specifications require explicit synchronization between the environment and the module actions. This synchronization is achived by shared variables. This contrasts with the fact that CSL action synchronization is translated into action conjunction, resulting in a more abstract specification. E ±> M specifications because of the explicit synchronization through shared variables have an implementation bias. CSL specifications are more intuitive than E ±> M specifications since the specifier is not requried to know the exact semantics of the operator ±>. Finally, CSL specifications can be viewed as an approach to write object-oriented TLA specifications.

2.4. Developmental approach and Tools

The developmental approach which is used in this case study proceeds as follows. First the structure model is built. It describes the application objects and the associations between them including aggregations. The notation which is used is the one proposed in [Rama96b]. In the structure model, the different associations may have application-specific behavior and behavioral constraints as described in [Rama97]. These allow to identify provided and required actions of the objects participating in the associations and the interactions between these objects in terms of abstract events (synchronous interactions). The description of these interactions is achieved in the interaction model.

Once the structure and interaction models are built, we then use CSL to describe in more detail the objects, composite objects, actions provided and required by these objects, and the interactions in terms of abstract events. As noted earlier, XOMT is an extension of OMT, i.e. it follows closely the OMT rules. In OMT, the behavior of objects is described by means of statecharts. By adopting CSL as a vehicle for specifying object behavior, interactions, and compositions, we should say how the computational model of CSL compares to statecharts. CSL describes the behavior of an object by its initial conditions, the invariants, and its actions. The initial conditions represent the starting state in a statechart. Actions can be interpreted as a combination of OMT events and the operations performed when the event occurs.

In OMT, events are atomic and they can be specialized. In CSL, actions are atomic and they can be specialized. However, in OMT events may denote an elapsed time. This is not actually handled in CSL, although it can be achieved by timing the action executions, or using the technique proposed by Lamport which imposes a minimum and a maximum time for th [redacted] action execu [redacted] n [Lamp94]. OMT event trace diagram corresponds to the trac [redacted] allowed b [redacted] traces [redacted] f an object can be derived from its CSL specification.

In CSL, states are implicitly captured by [redacted] e values [redacted] llowed for the loca [redacted] var [redacted] bles. The organization of states into disjoint substates or composit [redacted] tates ca [redacted] be ach [redacted] ates on the values of the local variables. For instance, let us co [redacted] ider the state [redacted] $P(x_i)$ $Q(x_i, y$ [redacted] If $P(x_i)$ is satisfied or $Q(x_i, y_i)$ is satisfied and $\forall x_i, y_i : P(x_i)$ [redacted] $(x_i, y_i) \Leftrightarrow$ fa [redacted], this may represent two disjoint substates, i.e. $P(x_i)$ and $Q(x_i, y_i)$ represent two distinct substates of s. It can be demonstrated that a substate inherits the properties of its superstate. However, composite states can only be specified in the context of compositions. For instance, the initial conditions of a composition may imply the initial conditions of its components. The initial state of the composition is a composite state bundling the initial states of its components.

In CSL, concurrency is based on interleaving, i.e. at the level of a single object there is no concurrency. When considering a composition, the internal events occurring in the composite are interleaved with its actions. In an internal event, more than two components may simultaneously interact. This means, there is concurrency between the interacting components.

OMT assumes that an atomic object is a finite state machine with a queue for incoming events. Composite objects may have more queues of events corresponding to each of its components. This contrasts with CSL where even if an object is a sequential system (or process), it has no queue. It accepts or blocks the execution of an action. In addition, it may display an undefined behavior or undergo non-deterministic changes by executing alone one of its internal actions.

OMT assumes that objects communicate by sending events. In addition, they can interact implicitly if one object has a guard condition that depends on the state of another object, such as being in a given sate. Our experience with XOMT has shown that describing synchronous interactions with such a model is cumbersome. In CSL, interaction is based on the concept of abstract event [Boch93b]. Abstract events are only meaningful in the context of an object association. *This has the consequence that the structure of an application shapes its dynamic behavior*.

Apart from events, in a statechart, there are transitions. These roughly correspond to the changes predicates defined in the semantics of actions in CSL. The so-called λ transitions in OMT correspond to the non-deterministic changes allowed by an object in CSL.

In CSL, an object action is defined in three parts: the enabling condition, the defined condition and the "changes" predicate. The enabling condition defines the states in which the action is enabled. The defined conditions determine the states in which when the action is executed it will results in a well-defined behavior. The changes predicate determines the changes made to the local state of the object as well as the result of output values produced by the action. *As a rule of thumb, the interaction specification is done when we may easily determine the enabling and defined states of the objects involved in each abstract event*. If we can not determine the enabling and defined of states of objects involved in the abstract events, we may not determine which abstract events do occur. Therefore the specification can be seen as incomplete.

After providing the CSL specifications, they are translated into canonical TLA formulas. With the TLA specifications which are obtained, the specifier may now use different TLA tools to verify the specifications. Results of this process can be reflected directly into CSL specifications since the translation process is a one-to-one mapping. Therefore from the canonical TLA formula using a simple translation we obtain the corresponding CSL specification.

3. Case Study

As an application of this case study, we have selected the lift problem which is one of the problems proposed for demonstrating the adequacy of a specification method for complex systems. We omit certain details which do not contribute to the essential points of this paper.

3.1. Problem statement

A lift system is to be installed in a building with m floors. The lift is aimed at moving people from one floor to another. The lift is used under the following constraints:

1. Each lift has a set of buttons, one for each floor. These illuminate when pressed and cause the lift to visit the corresponding floor. The illumination is canceled when the corresponding floor is visited by the lift.
2. Each floor has two buttons (except for ground and top floors), one to request an up-lift and one to request down-lift. These buttons illuminate when pressed. The illumination is canceled when a lift visits the floor and is either moving in the desired direction, or has no outstanding requests. In the latter case, if both floor request buttons are pressed, only one should be canceled. The algorithm to decide which to service first should minimize the waiting time for both requests.
3. When a lift has no requests to service, it should remain at its final destination with its doors closed and await further requests.
4. All requests for lifts from floors must be serviced eventually, with floors given equal priority.
5. All requests for floors within lifts must be serviced eventually, with floors being serviced sequentially in the direction of travel.
6. Each lift has an emergency button which, when pressed causes a warning signal to be sent to the site manager. The lift is then deemed "out of service". Each lift has a mechanism to cancel its "out of service" status.
7. The doors shall be closed when the lift moves.
8. The lift is stopped when it reaches the "out of service" state. Furthermore requests made from the lift carriage are then cleared, and no new requests from the lift carriage are accepted.

Many requirements of a lift are not mentioned, since the designer is expected to know what a lift is.

3.2. Developing the XOMT specification

Structure model

The structure model is about modeling the associations between objects as well as the structure of these objects in terms of other objects. We use the following rule of thumb for establishing associations between objects. We are justified in establishing an association between two objects, A and B, if and only if we want to express structural or behavioral constraints between the two objects. On the other hand, the structure of objects depend on the specifier and the level of detail that is required for the model.

We first provide an high-level structure model of a building since the user, the lift and the floors are in the context of a building. In structure models, objects are represented by rectangles and aggregation is shown by embedding one object into another. The structure model for the building can be interpreted as follows. We have a building which is a composite object consisting of Floor and Lift objects. When necessary the cardinality of the components are shown in a structure model. The object associations are shown in a structure model. Floor objects are associated between them. Each Floor is associated to the Lift through several associations.

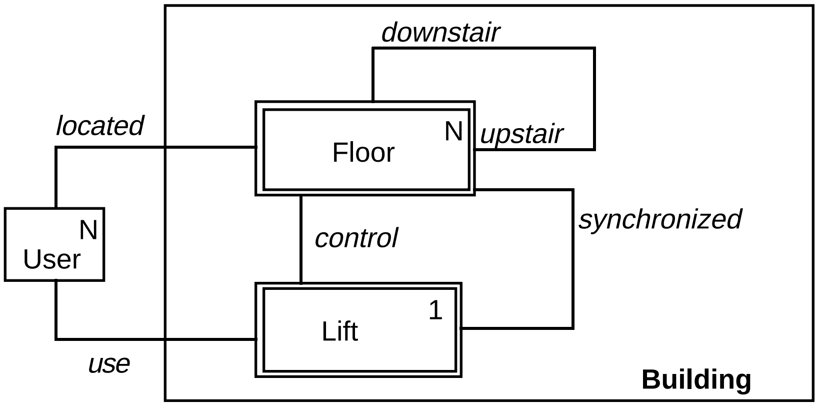

Figure 2: High level structure model of the lift application

The main object associations abstract the constrainment between the floor and the lift. When a user pushes the button at a floor, this propagates through the association "control" linking that floor to the lift. In addition, when the lift arrives at a given floor, if it stops and opens its doors, the floor doors must also be opened. This is captured by the association "synchronized". A user may be located at a given floor or traveling through the building using the lift. These information are respectively captured by the associations "located" and "use".

After providing this high-level view, we provide more details by describing the Lift object. As a rule of thumb, we use one structure model per composition when the composition is enough complex, or we regroup several compositions in a single structure model. In the structure model below, the Lift and its components are detailed. Please do not confuse Floor(i) with Floor objects, the former represent buttons inside the Lift while the latter is the real Floor.

In the structure model of the application, illustrated in Figure 2, the lift being a visible component of the building, it follows that the visible components of the lift are also visible at the level of the building. A lift has a ControlPanel and doors. The ControlPanel has various buttons including two buttons which control the opening and closing of the lift doors.

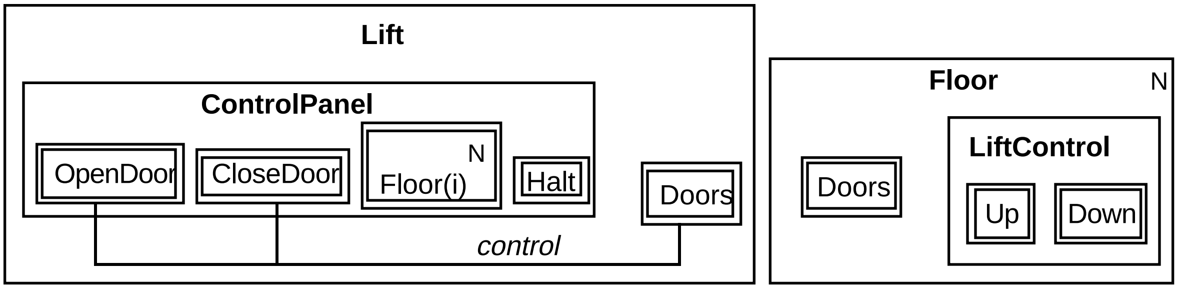

Figure 3: Detailed structure model of the lift and the floor

This is followed by a detailed structure model of the Floor. Each Floor object consists of Doors and a LiftControl. The LiftControl is an abstraction for the two buttons which control the Lift at each Floor. Notice that in order to avoid too much details on this structure model, we have left out the case of the first and the last Floors which have only one button per LiftControl. However, this can be captured by making the components of the LiftControl optional.

The structure models describe one lift per building. In a multi-lifts building we may have considered an additional object responsible for the coordination of the lifts. Such an object may be called a LiftManager.

Interaction specification

An interaction specification portrays the interactions between the objects forming the application. These interactions are described in terms of abstract events involving the objects of the applications. These abstract events occur only in the context of object associations. The associations which embed abstract events include but are not limited to the following : (1) lift doors and floor doors, (2) up button and the lift, (3) down button and the lift, (4) floor(i) button and the lift, (5) openDoor button and the lift, (6) closeDoor button and the lift, (7) halt button and the lift, (8) user and the lift.

The association 1 corresponds to "synchronized"; 2 and 3 correspond to "control"; 4, 5,6, and 7 correspond to the associations within the Lift composition; Finally, the association 8 corresponds to "use". The abstract events are described based on the requirements and assumptions below:

Lift and the doors
- When the lift arrives at a given floor where the button is pressed and the lift is going toward the direction indicated by the button, this makes the lift to stop at this floor and to open its doors.
- When the lift arrives at a given floor which was a target direction, this makes the lift to stop and to open its doors.
- When the lift is stopped and the button open door is pressed, this makes the lift to open its doors.
- When the lift doors are opened (respectively closed), the corresponding floor doors are also opened (respectively closed).

Up and down buttons
- When the lift arrives at a given floor which was a target direction, and the button up or down is pressed. These buttons should be deactivated. If both are pressed, non-deterministic choice is applied to deactivate only one button.
- When the lift arrives at a given floor which is in the direction of the target floor, and the button leading to the target floor is pressed, this makes the button to be deactivated.

Control panel
- When floor(i) button is pressed, this causes the indicated floor to be a target destination.
- When the button open door is pressed if the lift is stopped, this makes the lift to open its doors.
- When the button close door is pressed, if the lift is stopped, this makes the lift to close its doors.
- When the button halt is pressed, this leads the lift to the nearest floor and stop the lift at that floor canceling all the target destinations.

To make the description of the behavior of the lift tractable, we have decomposed its movement into two specific discrete steps, namely move(i,j) and stop(k). move(i,j) models the displacement from the floor i to the floor j. It is achieved in one complete execution step and it is atomic. stop(k) models the action forcing the lift to stay at the floor k. For instance, if the lift moves from the floor i and k and then stops at the floor k. If there is a floor j which is after the floor i and before the floor k, the lift movement is represented by three steps move(i,j) , move(j,k) , and stop(k).

In the following, we describe a simplified Lift system which can be later refined to include all the requirements. This is done for sake of brevity since the entire CSL specification of the Lift system takes several pages. The simplified system includes a button, the lift, the user and the doors specifications.

A button is described by the following CSL specification :

```
spec Button(d:Direction,[redacted]
 define ButtonState = {i[redacted]e,o[redacted]
 define Direction = {no[redacted], u[redacted]down}
  local  variables
        state : ButtonStat[redacted]servi[redacted]; floor : [redacted]oor[redacted]mber; d[redacted]ction : Direction
  initial  condition[redacted]
        state = idle    [redacted]rviced = false[redacted]tion = d    [redacted]oor = f
 behavior
        action push(f) =    [redacted]
```

```
                        enabled: [illegible]
                        defined: true
                        changes: (state' = on) ∧ (f = floor)
     action call(f,d) = [illegible] d:Direction
                        enabled: state = on
                        defined: true
                        changes: ((d = direction) ∧ (f = floor)
     action serviced(f,d) = [illegible] d:Direction
                        enabled: state = on
                        defined: (serviced=false)
                        changes:(d = direction) ∧ (f = floor) ∧ (serviced'= true)
     action release =   enabled: (state = on) ∧ (serviced = true)
                        defined: true
                        changes: (state' = idle) ∧ (serviced'= false)
  end spec Button
```

The specification is parameterized to allow the generic specification of buttons according to the direction and the floor which is serviced by the button. Based on the semantics of CSL actions introduced in Section 2, there are many specification variants for an action. For instance consider the push(f) action. It can be specified as:

```
 action push(f) = [illegible]
                        enabled: true
                        defined: true
                        changes: (state' = on) ∧ (f = floor)
```

The action is always enabled (i.e. never blocks) and it is always well-defined (i.e. never results in an undefined behavior). In other words, you may call the action anytime.

```
 action push(f) = [illegible]
                        enabled: state = idle
                        defined: true
                        changes: (state' = on) ∧ (f = floor)
```

The action is enabled in states where the button is idle (i.e. it blocks in states where the button is on) and it is always well-defined. In other words, you may call the action only when the button is idle.

```
 action push(f) = [illegible]
                        enabled: true
                        defined: state = idle
                        changes: (state' = on) ∧ (f = floor)
```

The action is always enabled. It results in undefined behavior in states where the button is on. In other words, you may call the action anytime but if the button is not idle, the behavior is undefined.

```
 action push(f) = [illegible]
                        enabled: state = idle
                        defined: state = idle
                        changes: (state' = on) ∧ (f = floor)
```

The action blocks in states where the button is on and it never results in undefined behavior.

It is up to the specifier to select the intended behavior. The actions of the button can be described informally as follows. The push action consists to press the button. call(f,d) is an action by which the button communicates with the lift. serviced(f,d) is an action by which the lift notifies the button that it has stopped at the corresponding floor.

In the requirements, we have stated that buttons are illuminated. This is specified by having a light in each button. The light object is described below.

```
spec Light
 local variables
       bulb : Boolean
 initial conditions
       bulb = false
 behavior
       action illuminate =
                                 enabled: bulb = false
                                 defined: true
                                 changes: bulb' = true
       action deilluminate =
                                 enabled: bulb = true
                                 defined: true
                                 changes: bulb' = false
end spec Light
```

The specification of a button which has a light is as follows :

```
spec ButtonWithLight
extends Button
 contains
       visible light : Light
 abstract events
       association is-part-of
             event enlighten =     actions:      illuminate
                                                 light.illuminate
             event darken =        actions:      deilluminate
                                                 light.deilluminate
 initial conditions
       light.bulb = false
 behavior
       action illuminate =
                                 enabled: state = on
                                 defined: true
                                 changes: true
       action deilluminate =
                                 enabled: state = idle
                                 defined: true
                                 changes: true
end spec ButtonWithLight
```

The abstract event enlighten is the synchronization of the actions illuminate of ButtonWithLight and the action illuminate of the component light of ButtonWithLight. Both actions do not have parameters. In the remaining of this paper, for sake of brevety, we no longer describe the inner-workings of actions in terms of **enabled**, **defined** and **changes** predicates.

In a Floor there are no specific interactions between the components. We assume the operators : *first_floor()* returning the first floor of the building, *last_floor()* returning the last floor of the building, and *next_floor(direction, f, n)* returning the floor following a given floor based on the direction. Direction can be none, up, and down.

```
spec Floor(id:FloorNumber)
 contains
       up : ButtonWithLight(up, id)
       down : ButtonWithLight(down, id)
       doors : Doors
 end spec Floor
```

```
 spec Doors
 define DoorState = {closed, open}
  local variables
       state : DoorState
  initial conditions
       state = closed
  behavior
       action open
       action close
 end spec Doors
```

```
 spec Lift
  contains
       doors : Doors
  local variables
       closeDoor, openDoor :[redacted]oo[redacted]n; s[redacted]e :[redacted]e; locati[redacted]: FloorNumber; direction : Direction
       up_req : Requests; do[redacted]re[redacted]: Re[redacted]ues[redacted]
  initial conditions
       closeDoor = false [redacted]penDo[redacted]= fa[redacted]e [redacted]te[redacted]= stopped[redacted]
       location = first_fl[redacted]r()
       direction = up [redacted]p_req = ∅ [redacted]own_req = ∅[redacted]
  behavior
       action open_door
       action close_door
       action select_destination(f)
       action out_of_service
       action called(f,d)
       action service(f,d)
       action move
       action atfloor
 end spec Lift
```

```
spec SimplifiedLift
 contains
       lift : Lift
       user : User
       floors : n : i..j Floor(n)
 abstract events
       association control
               event  callup =       local:        f : FloorNumber, d : Direction
                                     actions:      floors[n].up.call(f, d)
                                                   lift.called(f, d)
               event calldown =      local:        f : FloorNumber, d : Direction
                                                   floors[n].down.call(f, d)
```

```
                                        lift.called(f, d)
      ... the other abstract events are defined similarly
  end spec SimplifiedLift
```

3.3. Developing the formal specification

Now, we translate the CSL specifications into Lamport's TLA so that TLA tools can be used. For the lift system, the state where the button interacts with the lift acts as a synchronization state allowing the proof of the property: *when we press the button up at a given floor, then the lift will eventually service that floor*. Notice that we have selected a progress property because it also illustrates how to prove invariance properties. In the following, we only portrays the translation of Button, Light and ButtonWithLight specifications.

TLA Specifications

In the TLA specifications below, each CSL specification is mapped into a module. A *extends* clause corresponds to the TLA extends clause. A *contains* clause is mapped into instantiation of the modules corresponding to the components. Local variables are introduced as parameters of the module. At this point, the translation become one to one since the initial conditions clause is translated into an Init predicate, and the actions aretranslated based on the approach sketched in Section 2. They are included in the temporal section of the module. The temporal logic is only used to describe the behavior of the module in terms of Init predicate and actions. For the CSL specification Light, V = {bulb}, LV(illuminate) = {bulb}, LV(deilluminate) = {bulb}, PAR(illuminate) = ∅, and PAR(deilluminate) = ∅. This leads to the following translation if we use the **chaos** semantics:

[redacted]

```
                                   Mod[redacted]le Light
parameters [redacted]
      bulb : Variable [redacted]
predicates [redacted]
      Init     bulb = fal[redacted]
actions
      illuminate [redacted]e ⇒ bulb' [redacted]e)     (false [redacted] c[redacted]os))
      deilluminate [redacted]ll[redacted]f[redacted]) [redacted](true ⇒ bulb' [redacted]rue)     (false = [redacted] chaos))
temporal [redacted]
      Actions    il[redacted]n[redacted]
      Behavior    Init [redacted][Actions]<bul[redacted]
```

TLA Specification of the object Light

The translation of the CSL specification Button is more complicate. We have taken the care of adding extra predicates state ∊ ButtonState and floor ∊ FloorNumber in order to reflect the typing of the local variables in the TLA specification.

```
                                   Module Button
parameters
      direction : Variable [redacted]
      floor : Variable [redacted]
      state : [redacted]
      serviced : Variable [redacted]
      calling [redacted]
predicates [redacted]
      state ∊[redacted] ∊  Floor[redacted]er
      Init     state = "i[redacted]"  [redacted]ed = false [redacted]lling = false [redacted]
actions [redacted]
      push(f)        [redacted]ate = "idle")[redacted]
```

```
[redacted]true[redacted] = fl[redacted]r) [redacted] = [redacted]e)
[redacted]ser[redacted]floor, d[redacted]ion))
[redacted])
    call(f,d)      [redacted] true)
                   ((true ⇒ [redacted]cti[redacted] = floor) [redacted] = false)
                   [redacted]anged([redacted]ced, floor,[redacted]on))
                   [redacted]))
    serviced(f,d)  [redacted]ate = [redacted]
                   [redacted](service[redacted]f[redacted]) = [redacted]direc[redacted] = floor)
                   [redacted]e[redacted]ce[redacted] = tru[redacted]
                   [redacted]changed(sta[redacted] floor, direction))
                   [redacted](¬(serv[redacted]ec[redacted] f[redacted]se)⇒ c[redacted]os))
    release        [redacted]viced = t[redacted]
                   [redacted]e' = "idle[redacted](serviced'= false[redacted])
                   (false [redacted]aos))
temporal
    Actions        [redacted] d2, d3[redacted]Di[redacted]tion
                   [redacted](f2,[redacted])   service[redacted]3,d3)   release [redacted]
    Behavior       [redacted]Init [redacted]Actions]<state,serviced, calling>
```

TLA Specification of the object Button

```
                   [redacted]Module ButtonWithLight
extends Butt[redacted]
Local light    instance [redacted]
predicates [redacted]
    Init    light.[redacted]
actions
    illuminate     [redacted] true) [redacted](false [redacted]ch[redacted]s))
    deilluminat[redacted]') [redacted]> true) [redacted](false [redacted] chaos))
    enlighten  [redacted]llumi[redacted]te [redacted]
    darken    dei[redacted]mina[redacted]
temporal
    Actions    e[redacted]
    Behavior    Init [redacted][Actions]
```

TLA Specification of the object Button with Light

Based on the TLA specifications, various properties of the lift system may be verified. For instance, deadlock freedom is a safety property. It means that the program can never reach a state where all processes are blocked. In our case study, it is modeled by invariant(E) where E denotes the disjunction of the enabling conditions of all the abstract events. The property that all the requests will eventually be serviced is the conjunction of:
1) deadlock freedom;
2) if the button up (or down) of a given floor is pressed then eventually the lift will stop at this floor.
This latter property can be expressed as the conjunction of:
- pressing the button causes the lift to be aware of the request;
- if there is a request for a floor then the lift will go to that floor.

The proof of the property *if the button up (or down) of a given floor is pressed then eventually the lift will stop at this floor* implies considering the three following cases :

a) the lift is already at the floor;
b) the request is made in the direction towards which the lift is going;
c) the request is made in the opposite direction towards which the lift is actually going.

In the following, we show the TLA rules applied for proving these properties.

Proving various properties

For the lift system, [redacted]adlock freedom can be interpreted as the conjunction of two safety properties:
a) In the initial st[redacted] a[redacted]ast one abstract event (represented by a TLA action) is enabled.
b) Each abstract [redacted]ads the objects into states where at least one abstract event is enabled.

The fairness r[redacted]en[redacted] will make the enabled action to occur. In order to prove this property, we use TLA IN[redacted]ule sho[redacted]elow and described in [Lamp94].

$$\frac{I \quad [\text{redacted}] \Rightarrow I'}{I \quad [\text{redacted}][N]_f \Rightarrow \square I}$$

This rule is used to prove th[redacted] a program s[redacted]isfies an invariance property □*I*. The hypothesis asserts that a $[N]_f$ step cannot falsi[redacted]*I*. The concl[redacted]on asserts th[redacted]if *I* is true initially and every step is a $[N]_f$ step, then *I* is always true. *I* [redacted]presents[redacted] disjunction[redacted] the actions provided by the program. It suffices to choose the invariant[redacted]roperty [redacted]s at least one[redacted]bstract event is enabled to prove the deadlock freedom property using the ab[redacted]e [redacted]le. F[redacted] the Bu[redacted]onWithLight, this can be expressed as the following TLA formula

Enabled illuminate *Enabled* [redacted]illuminate[redacted]

which can be reduced to (state = "on" state = [redacted]lle") by predicate logic since an abstract event is enabled when all its constituant actions are enabled.

Liveness properties such as when the button of a floor is pressed the lift will eventually stop at this floor are proved using [redacted]A WF1 or SF1 rules described in [Lamp94]. To ease the proof of this property, we decom[redacted] property int[redacted]two other properties. The WF1 rule is more complicated than INV1. It is u[redacted]perties [redacted]form P*leads-to*Q from a weak fairness condition $WF_f(A)$. Here *A* denotes a[redacted] An *A*[redacted]ep is understood as the execution of the action *A*. It can be applied when a[redacted]ste[redacted]th [redacted]ue makes Q true, the reader is referred to [Lamp94] for more details. WF1 r[redacted] as[redacted]ws:

P [redacted]$]_f$ ⇒ [redacted]
[redacted]
P ⇒ *Enabled* [redacted]A[redacted]
[redacted]
□[[redacted]) ⇒ P[redacted]*ad*[redacted]*to*Q

a) When w[redacted]on of a [redacted]then the flo[redacted]er [redacted]entually be listed in the request list of the lift. This propert[redacted]ov[redacted] with T[redacted]A WF[redacted] ta[redacted]g

P as [redacted]].u[redacted]state = "o[redacted]" [redacted] f[redacted]rs[n].up[redacted]oo[redacted]
[redacted](f[redacted]rs[[redacted].down.state[redacted] "o[redacted] [redacted]= floors[n].d[redacted]or)

Q as [redacted]up_req f ∈ lift.[redacted]wn_req

A as the abstract events:

call = [redacted]ber, d_1, d_2 : Direction : floors[n].up.c[redacted](f_1, d_1)
lift.called(f[redacted]d_2)
($d_1 = d_2$ [redacted]$= f_2$)

respectively,

call = [redacted]ber, d_1, d_2 : Direction : floors[n].down.call(f_1, d_1)
lift.called(f_2, d_2)

$(d_1 = d_2 \wedge f_1 = f_2)$

WF$_f(A)$ is deduced from the fairness of floors[n].up.call, floors[n].down.call, and lift.call actions.

b) If the floor number is listed in one of the request lists of the lift then the lift will stop at the corresponding floor. This is internal to the lift and it can be deduced from the sequence of states of the lift module, i.e. its behavior. It can be also broken down into smaller properties.

Nonetheless, proofs are tedious and complicate. A great deal of these proofs can be mechanical and take advantage of the structure of the formulas to decompose the proofs. There are many tools which can assist in proving the properties of TLA specifications. At the University of Dortmund, a certain number of tools for developing, preparing, building, testing and verifying TLA specifications have been prototyped [Mest94]. Among these tools, we selected eTLA+ which is an interpreter allowing the interpretation of specifications combined with graphical visualization of their execution.

Once the CSL specifications are translated into TLA, TLA specifications can then be translated into eTLA+ which is used as input for eTLA+ interpreter. The eTLA+ interpreter allows symbolic debugging including stepwise or continuous execution and tracing. Non-determinism is handled by allowing the user to select the action to execute or if he may prefer a scheduling strategy.

Also CSL specifications can be translated in TLALight another variant of TLA. TLALight specifications are implemented using a C++ translator which derives a distributed implementation prototype as a set of communicating processes in a workstation network.

4. Discussion and conclusions

There is a practical requirement to convert the specifications written in CSL into C++ for prototyping. The converted code is also a useful starting point for implementation. Code generation can also commence from the structure model. C++ declarations and some of the procedural code for a program can be directly generated from the structure and the interaction models. The problem of automating the translation is not covered in the paper.

There is implicit relation between the formal specifications and the informal ones. The characteristics of component associations are such that their presence simplifies proofs of behavioral properties, as compared to general associations. In addition, we can derive useful properties of the composition based on that of its components. For instance, specification invariants of the composition implies that of its components. This is due to the encapsulation of the local variables in the components. Deadlock freedom of components once proved individually, there is no need to prove these again when the components are incorporated in a composition and we would like to prove deadlock freedom of the composition. The reasoning is achieved only at the level of the composition. Based on the interconnections between components, we may prove progress properties for the composition where P and Q are predicates over distinct components.

Nonetheless, we need tools which support the approach. Such tools will increase usability and easier the practice with the approach. In addition, they will help to accumulate experience with the approach and pave the way for its maturation and large usage. The existence of such tools will allow the integration of the approach in the context of industrial application development. In this paper, we sketched how different tools can be combined for supporting the XOMT developmental approach. In order to support this development process, the adaptation of a case tool, namely MetaEdit tool [Meta95a] is underway. MetaEdit tool supports various object notations and methods, in addition it is parameterizable in the sense that you may define your own notation and method by defining a meta model of the notation. MetaEdit tool views an object-oriented method as a set of notations and specifications constructed using these notations. We use this latter feature for defining a meta model for XOMT.

In this paper, we have modified the developmental approach used in XOMT in order to integrate a new approach to the description of composite object behaviors. XOMT has now two distinct views which portray the structure and the dynamic behavior of applications. The developmental approach is compositional since we can connect different pieces of the design by establishing associations between objects of these pieces. It also supports refinement. In addition, the approach can be used for the description of component based-systems since any object can be refined into a composite object and composite objects can be components of other objects.

Throughout this case study, we show how to use the conceptual framework in the context of a light version of an object-oriented method. The structure of composite objects serves (in extenso of the application) for describing the dynamic behavior of these objects. The approach is based on the linkage between the structure and the behavior of composite objects. Such an approach has shown to be adequate when dealing with complex systems since the complexity of systems is concentrated on the interactions between components of such a system. CSL specifications allow to reason about the properties of the application in terms of traces. They can be translated in TLA, from which tools can be used for verification purposes as well as checking the composition of different pieces of the design.

Future development of this work includes the design and the implementation of a tool supporting CSL with an integrated environment for verifying the specifications. This may require the design and implementation of an automatic translator of CSL specifications to TLA specifications as well as the automatic reasoning with trace specifications.

References

[Abad95] Abadi, M., Lamport, L., Conjoining Specifications, ACM Transactions on Programming Languages and Systems, November 1995.

[Boch93b] Bochmann, G.v., Abstract dynamic modelling of complex systems, Publication départementale No 863, Dépt. IRO, Université de Montréal, Janvier 1993.

[Cook94] Cook, S. and Daniels, J. Designing Object Systems: Software isn't the real world. JOOP May 1994, 22-8.

[DSou94] D'Souza, D., Graff, P., Object Communication, JOOP, September 1994, p.14-23.

[Hare88] Harel, D., Statecharts: a Visual Formalism for Complex Systems, Science of Computer Programming, Vol. 8, No. 3, pp.231-274, 1988.

[Jarv90] Järvinen et al., Object-Oriented Specification of Reactive Systems, Proc. of 12 th ICSE, March 1990, IEEE Computer Society Press, p. 63-71.

[Lamp94] Lamport, L., The Temporal logic of actions, ACM TOPLAS 16(3):872-923, May 1994.

[Mest94] Mester, A., Herrmann, P., Tools for TLA-based specifications, RvS-TLA-94-35, February 1994, University of Dortmund.

[Meta95a] MetaEdit Personal 1.2, Customisable Case Tool to meet your requirements, MetaCase Consulting, Finland, 1995.

[Rama95a] Ramazani, D., Bochmann, G.v., A Conceptual Framework For Object Composition and Dynamic Behavior Description, Publication départementale #949, DIRO, Université de Montréal, Canada, 1995.

[Rama95b] Ramazani, D., Contribution of Object-Oriented Methodologies to the Specification of Complex Systems, Proceedings of Fifth Complex Systems Engineering Synthesis and Assessment Technology Workshop (CSESAW'95).

[Rama96b] Ramazani, D., Bochmann, G.v., Extending Object Modelling Technique for the Specification of Composite Objects, In Proceedings of TOOLS-USA96, July 1996.

[Rama97] Ramazani, D., Bochmann, G.v., Approaches to the Specification of Object Associations, In Proceedings of FMOODS97, July 1997.

[Rumb91] Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., Lorensen, W., Object-Oriented Modeling and Design, Prentice-Hall, Englewood Cliffs, NJ, 1991.