Compositional Verification for Object-Z

Kirsten Winter and Graeme Smith

Software Verification Research Centre
University of Queensland 4072, Australia
kirsten@svrc.uq.edu.au smith@svrc.uq.edu.au

Abstract. This paper presents a framework for compositional verification of Object-Z specifications. Its key feature is a proof rule based on decomposition of hierarchical Object-Z models. For each component in the hierarchy local properties are proven in a single proof step. However, we do not consider components in isolation. Instead, components are envisaged in the context of the referencing super-component and proof steps involve assumptions on properties of the sub-components. The framework is defined for Linear Temporal Logic (LTL).

1 Introduction

Object-Z [Smi00,Smi92] is an extension to Z [Spi92] which facilitates modelling in an object-oriented style through the addition of classes. Thus, an Object-Z specification models a system in a natural way by means of its components. It seems quite obvious to suggest a *compositional* approach for the analysis of such specifications that exploits this compositional structure. This raises the questions: Is it possible to split the proof task for the whole system into smaller sub-tasks in which we consider only a single sub-component at a time? Are these sub-tasks suited to being solved by model checking?

Smith [Smi95b] suggests an approach for modular reasoning by means of an axiomatic semantics which provides a deductive system based on the logic $\mathcal{W}$ [WB92]. This semantics allows single state and operation schemas to be analysed enabling class invariants to be proved by structural induction. When a class is used as an object within another class, its invariants can be used to help prove invariants of the incorporating class. Arbitrary properties on the behaviour of classes, however, cannot be proven.

Similarly, Griffiths [Gri97] introduces an approach for modular reasoning for Object-Z facilitating proof-steps for single classes. As in [Smi95b], this work is based on a reference semantics for Object-Z. Griffiths adopts a particular view on the reference semantics that allows for *strict modularity*. Strict modularity renders classes semantically independent of the rest of the specification. The semantic properties of an object are thus independent of its environment and can be proven in isolation (in contrast to system properties which must be proven for a particular specification as a whole). To achieve this independence, operations involving calls to operations in other components are considered to consist of an internal transition and an external interaction. Similarly, an independence of the

object's state is achieved by viewing attributes of other components as referenced variables which do not influence the local state semantically. The effect is that components are treated as *open* systems whose environment is unknown, and hence unconstrained.

Both approaches were developed for use with an interactive theorem prover (e.g., [SKS02]). Theorem provers have no limitation in terms of the model's state space and its environment. However, as soon as model checking is considered for the verification task, the complexity of the state space of targeted components becomes a vital criterion for applicability. Model checkers, as automated tools, handle finite systems that are *closed*. That is, the component has to be considered together with its environment. If the environment is unrestricted (as in the approach of Griffiths [Gri97]), this leads to an explosion of the state space and makes model checking infeasible.

In this paper we present an approach for modular verification of Object-Z specifications aimed at using model checking. It does not consider single components of a system in isolation but *maximal restrictions* of components. A maximal restriction of a component represents an object in the *specific* context in which it is used. The environment is thus restricted to the conditions of the actual specification. This notion allows us to treat the smallest possible entity of a complex system at each step. Since the context imposes restrictions on the behaviour of a component, impossible behaviour is cut out.

The components in our approach are objects, not classes. Classes could also be considered as components since they can be incorporated into other classes via inheritance. However, the flexibility of inheritance in Object-Z, and especially the ability to cancel and redefine operations [Smi00], means that behavioural properties are not in general shared between a class and the classes it inherits. Hence, the potential for modular reasoning is limited.

Maximally restricted components can only be defined for hierarchical object systems without circularities. Therefore, our approach focuses on Object-Z specifications with fixed object hierarchies and with value semantics [Smi92] rather than reference semantics. As shown recently by Smith [Smi02], Object-Z specifications with value semantics can be refined to those with reference semantics. Hence, our approach does not limit the potential for transformation of specifications to object-oriented code. It does, however, focus reasoning on the functionality of the specified system rather than the lower-level details of the object-oriented design.

Restrictions on components are not only given through the context of the super-component but also through the properties of the sub-components. For instance, not much can be proven about the behaviour of a component without any knowledge of the effect of operations of its sub-components that are involved in the behaviour. To solve this problem we adopt the *assume-guarantee style* reasoning that is suggested for the verification of parallel processes and hardware designs (e.g., [Pnu85,GL94]).

Within the assume-guarantee paradigm, assumptions about the environment are employed when verifying properties of a process. Properties are stated as a

triple of the form $\langle\varphi\rangle M\langle\psi\rangle$, where φ and ψ are temporal logic formulas and M is a process. This triple is satisfied if M satisfies ψ whenever the environment of M satisfies φ. A typical proof rule of this paradigm supports compositional reasoning, e.g.:

$$\frac{\begin{array}{ccc}\langle true\rangle & M & \langle\varphi\rangle \\ \langle\varphi\rangle & M' & \langle\psi\rangle\end{array}}{\langle true\rangle \; M \parallel M' \; \langle\psi\rangle}$$

The overall system consists of the sub-process M and M' running in parallel. Properties on each sub-process are proven in single steps where property φ, proven for process M, is used as an *assumption* to prove property ψ on process M'. From these two proof steps it can be concluded that property ψ also holds for the system as a whole.

We adopt the assume-guarantee paradigm for a compositional proof rule for Object-Z. The parallel composition of two processes $M \parallel M'$ is replaced by the concept of *incorporating* maximal restrictions of Object-Z components. We base the formal definition of incorporating components and maximal restrictions of components on *OZ structures*. An OZ structure defines the semantics of an Object-Z component in terms of a temporal structure (or Kripke structure). This provides the foundation for the compositional proof strategy for Object-Z and allows us to prove soundness of the corresponding proof rule for Linear Temporal Logic (LTL) [Eme90].

Section 2 introduces our compositional strategy in terms of maximal restrictions of system components. The underlying concept of OZ structures and their corresponding operations are formally defined in Section 3 and Section 4. This formalisation is used in Section 5 to formalise our proof rule in order to prove its soundness. We conclude in Section 6 with a discussion of future directions.

2 Decomposition of an Object-Z class hierarchy

Our work is based on a value semantics for Object-Z [Smi95a]. As a consequence, an Object-Z model does not specify any object references. Instead, a class may instantiate other objects, which are then part of the class. Therefore, we are able to give a *hierarchy of components* that is free of circularities. Each component is instantiated by one *super-component*, i.e., this super-component is unique, and it can only refer to *sub-components* that are strictly lower in the hierarchy (see Figure 1 where unsuitable relations between classes are crossed out). The hierarchy is given in terms of *levels*. The example in Figure 1 comprises

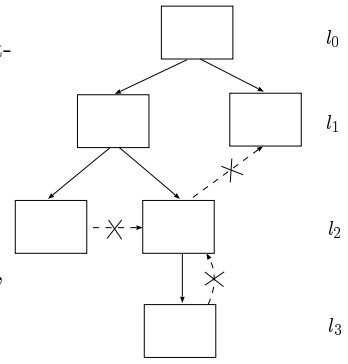
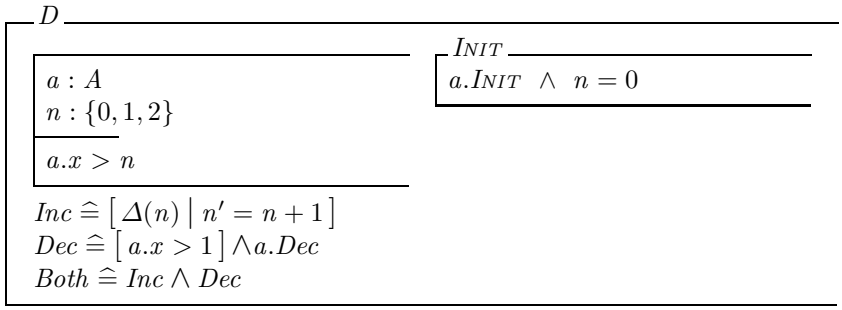
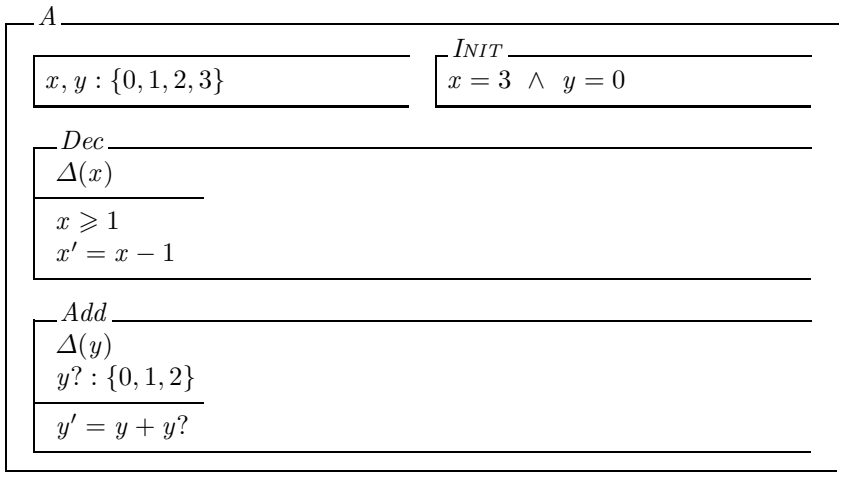


Fig. 1: Hierarchy of components for value semantics

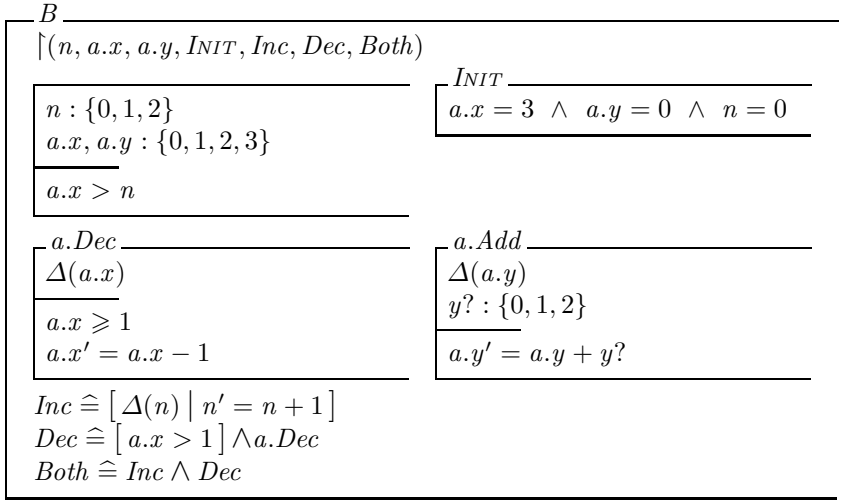
levels l_0, l_1, l_2, and l_3. We exclude models in which one operation evokes more than one operation on the same sub-component (since this violates Object-Z's history semantics [Smi95a]).
Semantically, every super-component together with its sub-components can be considered as an object of an ordinary Object-Z class in which the class definition of each sub-component is simply *incorporated* into the class definition of the super-component.

Example We present a simple example of a super-component incorporating its sub-component. Assume we define two classes D and A as follows:





Class D, the class of the super-component, contains an object a of class A, the sub-component. The full system of A incorporated into D can be modelled as an Object-Z class B below. The operations incorporated from class A are not included in B's visibility list. The operation $a.Dec$ is called from D's operation Dec. The operation $a.Add$ is not used by D and so can never occur.



Note that B is self-contained with respect to all definitions of state variables and operations that are used within the class. The object declaration $a : A$ has been replaced by declarations of two new variables representing its state variables x and y. To avoid name clashes, the names of these variables include the prefix '$a.$' (i.e., $a.x$, $a.y$).

2.1 Maximally restricted components

While proving properties, however, we would like a stepwise approach instead of targeting a component incorporating all its sub-components. We want to be able to consider only the smallest sub-system at each step. Therefore, we are aiming at the *maximal restriction* for each component within a hierarchy of Object-Z components.

The maximal restriction of a component is defined in terms of the operator *driven by*: A component a of class A is *driven by* a component d of class D. This operator captures the notion of a component operating within the particular context of its super-component. It allows the component to undergo only the subset of its class's behaviour that is actually possible in the particular hierarchy that is given.

We define the driven-by operator more formally based on temporal structures in Section 4.1. In terms of Object-Z classes, we can derive the class definition for a sub-component driven by its super-component from the class definitions of sub-component and super-component (classes A and D in our example) in four steps:

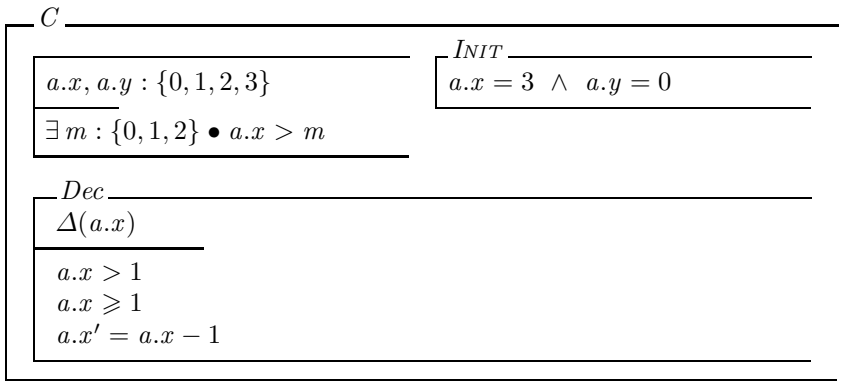
1. Replace the initial conditions of A with those initial conditions given in D that concern A (i.e., that contain state variables of A). Note that sub-components must be explicitly initialised in Object-Z using the notation $a.\textsc{Init}$ if this is intended. This is necessary since it is also possible that a

sub-component is not in its initial state when its super-component is in its initial state.
2. Add all state invariants of D to A which concern state variables of A. If such an invariant involves a state variable x of D then all occurrences of x must be replaced by a local variable which can take on any value of x's type.
3. Remove all operations from A that are never called in D.
4. Add all preconditions that occur on A's operations within D to the operations in A.

The following example shows how to apply this simple procedure to a given Object-Z model.

Example revisited. Given the class definitions of D and A as above, then the driven sub-component a is an object of a class C which can be modelled as shown below. Note that all attributes in class C are referred to using the prefix '$a.$', i.e., $a.x$ and $a.y$.

Class C contains only the operation *Dec* whose precondition is further restricted by the precondition $(a.x > 1)$, the precondition on the operation call in class D. Furthermore C adopts the state invariant on variable $a.x$ from D, ensuring that $a.x > 0$. To get this invariant we have to replace n, which is a state variable within class D, by its possible values and therefore have the expression $\exists\, m : \{0, 1, 2\} \bullet a.x > m$. The initial state remains unchanged since it coincides with the initial condition in class D.



The *maximal restriction* of a component is given as the component driven by its maximally restricted super-component. We adopt the notation $\left[\_\right]_\_$ for the driven-by operator. Assume $c(i)$ is a sub-component on level i of the given hierarchy and $c(i-1)$ is the super-component of $c(i)$ on level $i-1$. Then the maximal restriction of $c(i)$ is denoted as $\widetilde{c(i)} = \left[\, c(i) \,\right]_{\widetilde{c(i-1)}}$. On the top-most level of a hierarchy $\widetilde{c(0)} = c(0)$.

2.2 Compositional proof strategy

With the definition of a maximal restriction of a component we can now introduce a proof strategy that relies on a decomposition of a hierarchy of components.

Assume we have levels $l_0, l_1, \ldots, l_n$ in the hierarchy of the given system specification. We start with the lowest level in this hierarchy, namely l_n.

1. For all maximally restricted components on level l_n, $\widetilde{c(n)}$, we prove some properties $\{\varphi_n\}$ that are *observable* in $\widetilde{c(n)}$. Properties are observable in a component if all free variables contained in the property are local state variables in the component.
2. We use the properties $\{\varphi_n\}$ which are proved on the components $\widetilde{c(n)}$ as *assumptions* for proving properties on the maximal restriction of the super-component $\widetilde{c(n-1)}$.
3. We repeat the last step until we reach the component on the highest level, $\widetilde{c(0)}$.

For this stepwise proof procedure the user has to find for each level the necessary observable properties that can be proven locally on a maximally restricted component and will be helpful to prove properties on the next higher level. The benefit of this approach is that at each step only the local behaviour of an entity has to be considered. We observe that the maximally restricted components on each level are smaller than components that incorporate all sub-components of all lower levels.

In the remainder of this paper, we formalise this procedure in order to prove it sound. We introduce a simple proof rule for temporal logic properties which is formally defined in terms of temporal structures. The next sections introduce these temporal structures for Object-Z, called *OZ structures*, and the corresponding operations that are used in our context.

3 A Z Specification of OZ Structures

In this section, we introduce the notion of an *OZ structure* to represent the value semantics of an object in Object-Z. An OZ structure models the behaviour of one object and its interface to other objects. It comprises a unique identifier together with a single state transition system of the form $\langle S, I, R \rangle$, where S is a set of *states*, I is a set of *initial states*, and R is a *transition relation*. Since an OZ structure represents a single object of a class and not the class itself, the identifier is needed in order to refer to the object from OZ structures of other objects in the specification.

Each OZ structure covers the information that is observable at its own level. Thus, the OZ structure of each component includes information about the interface to its sub-components, i.e., input variables, operation calls and the existence of output variables, but not definitions from its sub-components.

Inputs and output variables are embedded into the state space following the approach of Smith and Winter [SW03]. Special variables are included in the

state to denote the component and sub-component events which occurred in the transition to the current state. A component may also refer to state variables of sub-components for the sake of restricting them, e.g., within state invariants. This allows state variables from the sub-component to be related to the local variables. Hence, such referenced variables are also included in the states of an OZ structure.

3.1 OZ Structure

A *state* of an OZ structure maps a finite set of (variable) names to their current values.

$$[Name, Value]$$

$$State == Name \pfun Value$$

For notational convenience, we assume names comprise identifiers such as n, a, etc., denoting local state variables; $a.x$, $a.y$, etc., denoting sub-component state variables; and the special names ev and $a.ev$, etc., denoting the names of the operation last called locally and on sub-components respectively.

Values comprise allowable Z values as well as operation names. The latter are assigned only to names ev, $a.ev$, etc., and include the values $none$, which models that no operation was called, and $init$, which models that initialisation has just happened.

An OZ structure is defined as follows.

OZStruct

$$Ident : Name$$
$$S : \mathbb{P}\, State$$
$$I : \mathbb{P}\, State$$
$$R : \mathbb{P}(State \times State)$$

$$\forall\, s_1, s_2 : S \bullet \operatorname{dom} s_1 = \operatorname{dom} s_2$$
$$I \subseteq S$$
$$R \subseteq (S \times S)$$
$$\forall\, s : S \bullet \exists\, s' : S \bullet (s, s') \in R$$

Apart from the identifier, OZ structures are defined similarly to *temporal structures* (Kripke structures) [Eme90]: Each state refers to the same variable names, i.e., the set of state variables cannot be increased or decreased in an OZ structure. The set of initial states is a subset of all states in the structure, i.e., $I \subseteq S$. The transition relation R is total, which is a characteristic of temporal structures. That is, each state in S has an outgoing edge. When deriving an OZ structure from an Object-Z class, this completeness can be achieved by adding to each state s without an outgoing edge (i.e., each state that is not a valid pre-state to any of the available operations) a transition back into itself such

that no operation is called. However, since the event variable ev is part of the state space, we have to introduce a copy of the state in which we modify the event variable to *none* (i.e., all state variables remain unchanged except ev).

Usually, a labelling function L is defined for temporal structures which maps each state of the structure to a set of satisfied atomic propositions AP, i.e., $L : S \rightarrow AP$. In OZ structures, this information is encoded into the states themselves: The mapping from variable names to their current evaluation in a state provides the set of atomic propositions that are satisfied in the state.

The example revisited To illustrate our notion of structures we describe the structure of an object d of the class D of the example introduced in Section 2 in terms of its state graph (see Figure 2).

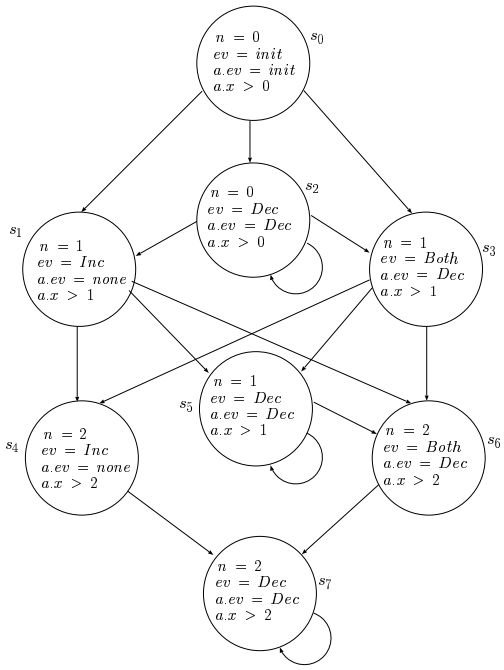


Fig. 2. Structure of object d of class D.

To keep the representation finite for the figure, we refer to the value of $a.x$ "symbolically" by means of the given state invariant stating that $a.x$ is greater than n. The states, in fact, represent sets of states that form a sub-graph whose behaviour is not distinguishable on the level of d.

Note that on the level of d the effect of operation Dec, and $a.Dec$ respectively, is not observable. Therefore, states s_2, s_5, and s_7 can loop forever. These looping

transitions also help to provide a *total* transition relation between the states. Therefore, we do not have to introduce additional states in which no event occurs (i.e., $ev = none$). To prove properties in D, we obviously have to employ assumptions on the effect of $a.Dec$ on variable $a.x$.

3.2 Auxiliary Functions on OZ Structures

To allow for a relation between states of sub-components and super-components, we define an auxiliary *dot* operator as a meta-relation on states. This operator changes the names of a state to include a prefix reflecting the sub-component to which the state belongs. That is, given that *id* is the identifier of a sub-component and $x_1, \ldots, x_n$ are names in the domain of the state of that sub-component then:

$$id \ \underline{dot} \ \{x_1 \mapsto v_1, \ldots, x_n \mapsto v_n\} \ = \ \{id.x_1 \mapsto v_1, \ldots, id.x_n \mapsto v_n\}$$

We also define a notion of agreement between states. A state s_1 *agrees with* another state s_2, $s_1 \approx s_2$, if it has the same value for any name the two states have in common.

$$\_ \approx \_ : State \leftrightarrow State$$

$$\forall s_1, s_2 : State \bullet$$
$$s_1 \approx s_2 \Leftrightarrow (\forall n : \operatorname{dom} s_1 \cap \operatorname{dom} s_2 \bullet s_1(n) = s_2(n))$$

Additionally, we define a function *names* for retrieving the *domain* of a structure. The domain of a structure is the set of state variable names occurring in the domain of its states:

$$names : OZStruct \rightarrow \mathbb{F}\, Name$$

$$\forall m : OZStruct \bullet$$
$$\forall s : m.S \bullet names(m) = dom(s)$$

4 Operations on OZ Structures

We now define operations on OZ structures that correspond to the operations on Object-Z classes which are informally introduced in Section 2, namely *A driven by D* and *D incorporating A*.

4.1 A driven by D

An OZ structure a can be seen in the environment of another OZ structure d, $[a]_d$. That is, we look at a within the context of d. This imposes those restrictions on states and initial states of a that are defined in d. Especially, the possible operations are reduced to those which are actually called by d.

This restriction is specified using the relation $\approx$ between states of the driven component and states of the driving component.

We define the OZ structure of a driven sub-component as follows:

$$
\begin{array}{|l}
\left[\_\right]\_ : (OZStruct \times OZStruct) \pfun OZStruct \\
\hline
\forall\, a : OZStruct;\ d : OZStruct \bullet \\
(a, d) \in \mathrm{dom} \left[\_\right]\_ \Leftrightarrow a.Ident \in names(d) \land \\
(a, d) \in \mathrm{dom} \left[\_\right]\_ \Rightarrow \\
\quad (\textbf{let}\ \ c == \left[a \right]_d;\ id == a.Ident \bullet \\
\quad c.Ident = id \land \\
\quad c.S = \{ s : State \mid s \in a.S \land (\exists\, ds : d.S \bullet (id\ \underline{dot}\ s) \approx ds) \\
\qquad\qquad \bullet\ id\ \underline{dot}\ s \} \land \\
\quad c.I = \{ i : State \mid i \in \mathrm{ran}\, a.R^* (\!|\ a.I\ |\!) \land (\exists\, di : d.I \bullet (id\ \underline{dot}\ i) \approx di) \\
\qquad\qquad \bullet\ id\ \underline{dot}\ i \} \land \\
\quad c.R = \{ s : State, s' : State \mid (s, s') \in a.R \land \\
\qquad\qquad (\exists\, ds : State;\ ds' : State \mid (ds, ds') \in d.R \bullet \\
\qquad\qquad\qquad (id\ \underline{dot}\ s) \approx ds \land (id\ \underline{dot}\ s') \approx ds') \\
\qquad\qquad \bullet\ ((id\ \underline{dot}\ s), (id\ \underline{dot}\ s')) \})
\end{array}
$$

All names in the domain of the states in a are substituted in $\left[a \right]_d$ by names with the appropriate prefix. For example, the variable name x is replaced by $a.x$ in our example in Section 2. This applies to all state variables, including the variable ev.

The set of states of a driven sub-component includes only those states of the sub-component that agree with a state in the super-component. That is, identical variable names carry the same value in these states. We use the dot operator to gain identical names, i.e., $(id\ \underline{dot}\ s) \approx ds$.

Similarly, the set of initial states collects all reachable states of the sub-component that agree with an initial state in the super-component. If the initial condition of the sub-component does not coincide with the initial condition of the super-component then the latter condition is adopted. That is, the initialisation of the driven structure is overwritten by the driving environment. However, initially the driven sub-component must be in a state reachable within the structure, i.e., in the range of the reflexive-transitive closure of relation R on initial states ($\mathrm{ran}\, a.R^* (\!|\ a.I\ |\!)$). This is required by the history semantics of Object-Z [Smi95a].

The transition relation of a driven structure is defined as a set of pairs of states of the sub-component that have a matching pair of states in the super-component. That is, for each transition (s, s') there is a corresponding transition in the super-component such that pre- and post-state agree with s and s' (modulo name prefixes).

4.2 Stuttering components

If we consider components in the environment of super-components, we have to allow for non-active behaviour in which the super-component is active but

not referring to the local operations of the driven sub-component. In terms of structures, this forces us to introduce *stuttering* behaviour of sub-components.

Stuttering is represented in a structure by stuttering states. A stuttering state leaves all state variables unchanged except the event *ev* which becomes *none*. The structure may stay arbitrarily long in a stuttering state before it becomes active again. Infinite stuttering is not excluded.

We formalise these additions to states and transitions in the following way. (Note that *id.ev* denotes a name in this definition and not an expression.)

$$
\begin{array}{|l}
stutt : OZStruct \nrightarrow OZStruct \\
\hline
\forall\, c : OZStruct \bullet \\
c \in \mathrm{dom}\, stutt \Leftrightarrow (\exists\, a, d : OZStruct \bullet c = \left[\, a \,\right]_d) \wedge \\
c \in \mathrm{dom}\, stutt \Rightarrow \\
\quad \textbf{let}\ \ e == stutt(c);\ id == c.Ident \bullet \\
\quad e.Ident = id \wedge \\
\quad e.S = c.S \cup \{ s : State \mid \mathrm{dom}\, s = names(c) \wedge s(id.ev) = none \wedge \\
\qquad\qquad (\forall\, p : (names(a) \setminus \{id.ev\}) \bullet (\exists\, cs : c.S \bullet s(p) = cs(p)))\} \wedge \\
\quad e.I = c.I \wedge \\
\quad e.R = c.R \ \ \cup \{ s : e.S, s' : e.S \mid s'(id.ev) = none \wedge \\
\qquad\qquad\qquad (\forall\, p : (names(a) \setminus \{id.ev\}) \bullet s(p) = s'(p))\} \\
\qquad\qquad \cup \{ t : e.S, t' : e.S \mid t(id.ev) = none \wedge t = t' \} \\
\qquad\qquad \cup \{ q : e.S, q' : e.S \mid q(id.ev) = none \wedge \\
\qquad\qquad\quad (\exists\, cs : c.S \mid (cs, q') \in c.R \bullet \\
\qquad\qquad\qquad\quad (\forall\, p : (names(c) \setminus \{id.ev\}) \bullet cs(p) = q(p)))\}
\end{array}
$$

An example of a stuttering component is given in Figure 3 where a and d are objects of classes A and D, respectively, of the example given in Section 2. The structure $\left[\, a \,\right]_d$ consists of the states s_0, s_1, and s_2. To get the structure $stutt(\left[\, a \,\right]_d)$ we have to extend the set of states by s_{0B}, s_{1B}, and s_{2B}, in which the structure is passive. These states, although not important on the level of a, are necessary for generating a correct incorporating structure (see Section 4.3).

Again, this graph only shows events locally observable to a. Operation *Add* is never active in this structure since within the environment of d it is never called. Since the state variable $a.y$ does not occur in a delta-list of any of the operations of $\left[\, a \,\right]_d$, it remains unchanged in every state.

4.3 D incorporating A

A system comprising an OZ structure d *incorporating* an OZ structure a is denoted by $d \ll \{a\}$. Since d may incorporate several objects, the right-hand argument is modelled as a (finite) set of the corresponding OZ structures. To be self-contained, $d \ll aset$ incorporates all definitions of state variables and operations that are referred to in the super-component d but leaves out non-referenced definitions and operations of the sub-components. The definition of $\_ \ll \_$ coincides with our suggested Object-Z model of class B in the example in Section 2 and is formalised as follows.

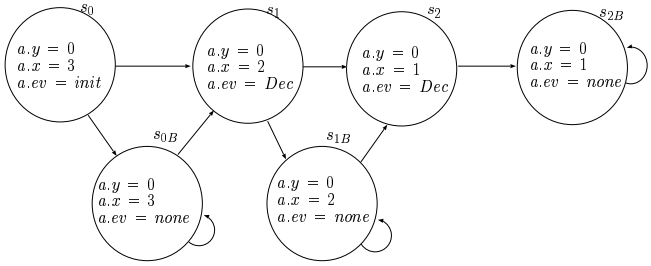


Fig. 3. Structure of the object $stutt([a]_d)$.

$$
\begin{array}{|l}
\_ \ll \_ : (OZStruct \times \mathbb{F}\, OZStruct) \to OZStruct \\
\hline
\forall\, d : OZStruct;\ aset : \mathbb{F}\, OZStruct \bullet \\
(d, aset) \in \mathrm{dom}\, \_ \ll \_ \Leftrightarrow (\forall\, a : aset \bullet a.Ident \in names(d)) \land \\
(d, aset) \in \mathrm{dom}\, \_ \ll \_ \Rightarrow \\
\quad (\textbf{let}\ \ b = d \ll aset;\ aset' = \{a : aset \bullet stutt(\lceil\, a \,\rceil_d)\} \bullet \\
\quad b.Ident = d.Ident \\
\quad b.S = \{s : State \mid \mathrm{dom}(s) = (names(d)\ \setminus\ \{a : aset \bullet a.Ident\}) \\
\qquad\qquad\qquad\qquad \cup\ \bigcup\{a : aset' \bullet names(a)\} \\
\qquad \land\ (\exists\, ds : d.S \bullet ds \approx s) \\
\qquad \land\ (\forall\, a : aset' \bullet \exists\, as : a.S \bullet as \approx s)\} \\
\quad b.I = \{is : b.S \mid (\exists\, di : d.I \bullet di \approx is) \\
\qquad \land\ (\forall\, a : aset' \bullet \exists\, ai : a.I \bullet ai \approx is)\} \\
\quad b.R = \{s : b.S, s' : b.S \mid \\
\qquad\quad (names(d) \lhd s, names(d) \lhd s') \in d.R \\
\qquad \land\ \forall\, a : aset' \bullet \\
\qquad\quad (names(a) \lhd s, names(a) \lhd s') \in a.R\})
\end{array}
$$

The definition relies on restricting all sub-components a in $aset$ to stuttering components driven by the super-component, i.e., to the form $stutt(\lceil\, a \,\rceil_d)$. As a consequence, the restrictions from the super-component are already included. All state variable names in the sub-components are given with an appropriate prefix (see definition of $\lceil \_ \rceil \_$). The initial states do not necessarily agree with the initial states of each of the sub-components a but need only agree with one of their reachable states (see the definition of initial states in a driven structure in Section 4.1). Also, the sub-components include passive behaviour (when none of their operations are called).

This assumption keeps the definition of the operator $\ll$ very simple: Each state of the incorporating structure $d \ll aset$ contains those names that are names of the super-component d except the identifiers of the sub-components (i.e., $\{a : aset \bullet a.Ident\}$) and the names of the sub-components which are annotated with the identifier of the sub-component through prefixing (e.g., $a.x$).

Moreover, for each state in the state space of $d \ll aset$ there exists a matching state in the super-component as well as a matching state in the sub-components. This is defined by means of the relation $\approx$. Accordingly, each initial state of the incorporating structure has a matching initial state in the super-component as well as in each of the sub-components.

The definition of the transition relation ensures that each pair of pre- and post-states has a matching pair of states in the super-component and in the sub-components. The inclusion of stuttering states in the sub-components in *aset* enables this definition of the transition relation to be satisfied.

5 Compositional Proofs

Based on the definition of OZ structures in Section 3 and operations thereof in Section 4, we are now able to formally define our proof strategy employing decomposition.

Following the strategy informally given in Section 2.2, a proof of a temporal property of a large hierarchical system is divided into smaller proof-steps. In each of these steps, we prove a *locally observable* property for a maximally restricted component on a single level. For proving local properties, we employ properties proven for the sub-components on the next lower level as *assumptions*. For the system depicted in Figure 4, three proof step are suggested: proof-step 1 involves component E and assumptions proven on sub-component F, proof-step 2 involves component B and assumptions proven on sub-components D and E, proof-step 3 involves component A and assumptions proven on B and C.

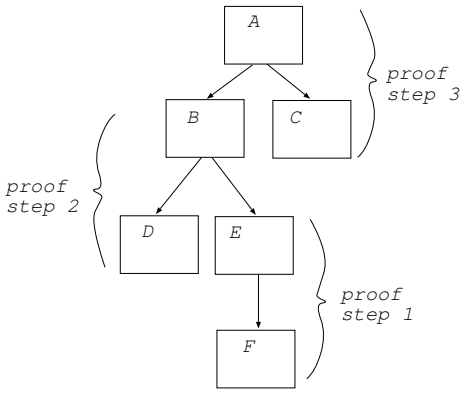


Fig. 4: Proof steps for a hierarchical system

To argue that this stepwise procedure is sound, we introduce the following proof rule on OZ structures.

Definition 5.1: Proof rule for hierarchical OZ structures

Let φ_e, φ, ψ be temporal logic properties and A and B be two OZ structures, where B is a sub-component of A. Then the following proof rule can be assumed:

$$\frac{\begin{array}{ccc} \langle true \rangle & stutt(\lceil B \rceil_A) & \langle \varphi \rangle \\ \langle \varphi_e \wedge \varphi \rangle & A & \langle \psi \rangle \end{array}}{\langle \varphi_e \rangle \qquad A \ll \{B\} \qquad \langle \psi \rangle}$$

If $\langle true \rangle$ $stutt(\lceil B \rceil_A)\langle \varphi \rangle$ and $\langle \varphi_e \wedge \varphi \rangle A \langle \psi \rangle$ can be proven, we can deduce that $\langle \varphi_e \rangle$ $A \ll \{B\}$ $\langle \psi \rangle$ is satisfied as well. (Property *true* represents that no assumption is made. φ_e represents any arbitrary assumption on the environment of the overall system.)

Using the proof rule above, our proof steps are simplified to local proofs on the smaller components $stutt(\lceil B \rceil_A)$ and A instead of the incorporating structure $A \ll \{B\}$ as a whole. Structure $stutt(\lceil B \rceil_A)$ reduces B to that part that is used within the context of A. Structure A does not incorporate attributes and state variables of B (other than those that are already referred to in A itself), instead the proof step relies on assumptions on the behaviour of B, namely φ.

The list of proof steps in our proof rule can easily be extended if we consider larger hierarchies of components (e.g., as shown in Figure 4):

$$\frac{\begin{array}{ccc} \langle true \rangle & stutt(\lceil F \rceil_E) & \langle \varphi_1 \rangle \\ \langle \varphi_1 \rangle & stutt(\lceil E \rceil_B) & \langle \varphi_2 \rangle \\ \langle true \rangle & stutt(\lceil D \rceil_B) & \langle \varphi_3 \rangle \\ \langle \varphi_2 \wedge \varphi_3 \rangle & stutt(\lceil B \rceil_A) & \langle \varphi_4 \rangle \\ \langle true \rangle & stutt(\lceil C \rceil_A) & \langle \varphi_5 \rangle \\ \langle \varphi_e \wedge (\varphi_4 \wedge \varphi_5) \rangle & A & \langle \psi \rangle \end{array}}{\langle \varphi_e \rangle \quad A \ll \{B \ll \{D, \{E \ll \{F\}\}\}, C\} \quad \langle \psi \rangle}$$

Note that each single proof step targets a much smaller component than the overall system $A \ll \{B \ll \{D, \{E \ll \{F\}\}\}, C\}$ which incorporates six components.

We prove the soundness of our proof rule for Linear Temporal Logic (LTL). The following section introduces LTL and its semantics.

5.1 The temporal logic LTL

LTL is a temporal logic for which model checking algorithms exist. It is defined on *paths*, i.e., sequences of states of a temporal structure, in the following way [GL94,Eme90]:

Definition 5.2: Linear Temporal Logic (LTL)

LTL formulas are those which can be generated by the following rules
- each atomic proposition $n = v$ is a formula, where n is variable name and v a value in the domain (i.e., type) of n
- if φ and ψ are formulas, then $\neg\varphi$ and $\varphi \wedge \psi$ are formulas
- if φ and ψ are formulas, then $\varphi \mathbf{U} \psi$ and $\mathbf{X}\varphi$ are formulas

These rules allow us to derive formulas of the form $\varphi_1 \vee \varphi_2$, $\varphi_1 \rightarrow \varphi_2$ (implication), the Boolean constants *true* and $false$, as well as $\mathbf{F}\varphi = true\,\mathbf{U}\,\varphi$ ("eventually φ") and $\mathbf{G}\varphi = \neg\mathbf{F}\neg\varphi$ ("always φ").

The semantics of LTL is given in terms of temporal structures (or OZ structures as defined in Section 3). A *path* of a temporal structure $M = (S, I, R)$ is an infinite sequence of states $\pi = s_0 s_1 s_2 \ldots$ such that $(s_i, s_{i+1}) \in R$ for all indices $0 \leq i$. The notation π_i is used for the suffix of path π starting at index i, i.e., $\pi_i = s_i s_{i+1} s_{i+2} \ldots$

Definition 5.3: Semantics of LTL

Assume M is a temporal structure, $\pi = s_0 s_1 s_2 \ldots$ a path of M, and $\varphi, \varphi_1, \varphi_2$ are LTL formulas.

$M, \pi \models (n = v)$	if and only if	$(n \mapsto v) \in s_0$
$M, \pi \models \neg\varphi$	if and only if	$M, \pi \not\models \varphi$
$M, \pi \models \varphi_1 \wedge \varphi_2$	if and only if	$M, \pi \models \varphi_1$ and $M, \pi \models \varphi_2$
$M, \pi \models \mathbf{X}\,\varphi$	if and only if	$M, \pi_1 \models \varphi$
$M, \pi \models \varphi_1 \mathbf{U}\,\varphi_2$	if and only if	$\exists\, j (M, \pi_j \models \varphi_2)$ and $\forall\, k < j (M, \pi_k \models \varphi_1)$

A formula φ is called *valid* in structure M, if $M, \pi \models \varphi$ for any path π of M that starts in an initial state of M. That is, to satisfy an LTL property *every* possible behaviour of our system or sub-component has to satisfy the property. We lift the operator $\models$ to a relation on structures and formulas in order to denote *validation* of a formula in a structure which is then denoted by $M \models \varphi$.

5.2 Soundness of compositional proofs

Since the semantics of LTL is given in terms of the relation $\models$ we reformulate the proof rule given in Definition 5.1. The statement $\langle\varphi_1\rangle M \langle\varphi_2\rangle$ can be formulated in the following way: φ_2 is valid in M under the assumption that φ_1 holds if any path π from an initial state in M satisfies $\varphi_1 \rightarrow \varphi_2$, i.e., $M, \pi \models (\varphi_1 \rightarrow \varphi_2)$ for all π and therefore $M \models (\varphi_1 \rightarrow \varphi_2)$.

To ensure soundness of the proof rule, we have to prove the following theorem for all LTL formulas.

Theorem 1

$$\forall\, \varphi, \varphi_e, \psi \bullet \mathit{stutt}(\left[\, B \,\right]_A) \models \varphi \quad \wedge \quad A \models (\varphi_e \wedge \varphi) \rightarrow \psi \quad \Rightarrow \quad (A \ll \{B\}) \models \varphi_e \rightarrow \psi.$$

With the two following lemmas the proof of Theorem 1 becomes straightforward.

Lemma 1 $\forall\, \varphi \bullet \mathit{stutt}(\left[\, B \,\right]_A) \models \varphi \quad \Rightarrow \quad (A \ll \{B\}) \models \varphi$

If a property is valid in structure $\mathit{stutt}(\left[\, B \,\right]_A)$ then it is also valid in structure $A \ll \{B\}$.

Lemma 2 $\forall\, \varphi \bullet A \models \varphi \quad \Rightarrow \quad (A \ll \{B\}) \models \varphi$

If a property is valid in structure A then it is also valid in structure $A \ll \{B\}$.

Intuitively, these lemmas are true since the structure $A \ll \{B\}$, the full system, is more restricted than structures A or $stutt(\lceil B \rceil_A)$.

Proof of Theorem 1: Let φ, φ_e, and ψ be any LTL formulas. Assume $stutt(\lceil B \rceil_A) \models \varphi$ and $A \models (\varphi_e \wedge \varphi) \rightarrow \psi$. With Lemma 1 and Lemma 2 it follows that $A \ll \{B\} \models \varphi$ and $A \ll \{B\} \models (\varphi_e \wedge \varphi) \rightarrow \psi$. According to the semantics of LTL, this implies $A \ll \{B\} \models \varphi \wedge ((\varphi_e \wedge \varphi) \rightarrow \psi)$ from which it follows that $A \ll \{B\} \models \varphi_e \rightarrow \psi$. □

For the proof of Lemma 1 and Lemma 2, we introduce two additional Lemmas, Lemma 3 and Lemma 4 later on.

In the following, we refer to S_A, S_B and S_{AB} as the sets of states of the corresponding structures A, $stutt(\lceil B \rceil_A)$, and $A \ll \{B\}$ according to the definitions in Sections 3.1, 4.1, 4.2, and 4.3. A similar notation is used for sets of initial states I_A, I_B and I_{AB}, and transition relations R_A, R_B and R_{AB}. Recall also that function *names* provides the set of state variables of a structure (as defined in Section 3.2, note that $names(stutt(\lceil B \rceil_A)) = names(\lceil B \rceil_A)$).

Lemma 3 *For all paths $\pi^{AB} = t_0 t_1 \ldots$ in structure $A \ll \{B\}$ there exists a path $\pi^B = s_0 s_1 \ldots$ in structure $stutt(\lceil B \rceil_A)$ such that $\forall\, i \geq 0 \bullet t_i \approx s_i$.*

For every path in the incorporating structure $A \ll \{B\}$ there exists a *corresponding* path in the *stuttering* driven sub-component $stutt(\lceil B \rceil_A)$. That is, every state in the path of the incorporating structure has a corresponding state (i.e., a state that agrees with it) in the path of the driven component. This lemma holds only for sub-components which include stuttering states as defined in Section 4.2. They allow the sub-component to remain unchanged while the super-component calls operations outside the sub-component.

Proof: $\forall\, t_i$ in path $\pi^{AB} = t_0 t_1 \ldots$ in $A \ll \{B\}$ there exists a state $s_i \in S_B$ such that $(names(\lceil B \rceil_A) \lhd t_i) \in S_B$ and $\forall\, i \geq 0 \bullet (s_i, s_{i+1}) \in R_B$ (per definition of R_{AB} in Section 4.3). It follows that there exists a path $\pi^B = s_0 s_1 \ldots$ in structure $stutt(\lceil B \rceil_A)$. □

Using Lemma 3 we are now able to prove Lemma 1. The proof is given inductively over the structure of LTL formulas.

Proof of Lemma 1:

- Assume $\varphi = (n = v)$ and $stutt(\lceil B \rceil_A) \models \varphi$.
  Proof by contradiction:
  Assume $A \ll \{B\} \not\models \varphi$

$\Rightarrow$ $\exists\, \pi^{AB} = t_0^{ab} t_1^{ab} \ldots$ of $A \ll \{B\}$ such that
$(A \ll \{B\}), \pi^{AB} \not\models (n = v)$
$\Rightarrow$ $\exists\, t_0^{ab} \in I_{AB} \bullet (n, v) \notin t_0^{ab}$
$\Rightarrow$ $\exists\, s_0^b \in I_B \bullet (n, v) \notin s_0^b$ (by definition of I_{AB})
$\Rightarrow$ $\exists\, \pi^B = s_0^b \ldots$ of $stutt(\lceil B \rceil_A)$ such that
$stutt(\lceil B \rceil_A), \pi^B \not\models (n = v)$
$\Rightarrow$ $stutt(\lceil B \rceil_A) \not\models \varphi$ (by definition of $\models$)

– Assume $\varphi = \mathbf{X}\, \varphi_1$ and $stutt(\lceil B \rceil_A) \models \varphi$.
Proof by contradiction:
Assume $A \ll \{B\} \not\models \varphi$
$\Rightarrow$ $\exists\, \pi^{AB} = t_0^{ab} t_1^{ab} \ldots$ of $A \ll \{B\}$
such that $(A \ll \{B\}), \pi_1^{AB} \not\models \varphi_1$
$\Rightarrow$ $\exists\, \pi^B = s_0^b s_1^b \ldots$ of $stutt(\lceil B \rceil_A)$ such that $\forall\, j \geq 0 (s_j^b \approx t_j^{ab})$ and
$stutt(\lceil B \rceil_A), \pi_1^B \not\models \varphi_1$ (by Lemma 3)
$\Rightarrow$ $\exists\, \pi^B$ of $stutt(\lceil B \rceil_A)$ such that $stutt(\lceil B \rceil_A), \pi^B \not\models \varphi$
$\Rightarrow$ $stutt(\lceil B \rceil_A) \not\models \varphi$ (by definition of $\models$)

– Assume $\varphi = \varphi_1 \mathbf{U}\, \varphi_2$ and $stutt(\lceil B \rceil_A) \models \varphi$.
Proof by contradiction:
Assume $A \ll \{B\} \not\models \varphi$
$\Rightarrow$ $\exists\, \pi^{AB} = t_0^{ab} t_1^{ab} \ldots$ of $A \ll \{B\}$ such that
$\nexists\, j (A \ll \{B\}, \pi_j^{AB} \models \varphi_2)$ or
$\big(\exists\, j (A \ll \{B\}, \pi_j^{AB} \models \varphi_2)$ and $\exists\, k < j (A \ll \{B\}, \pi_k^{AB} \not\models \varphi_1)\big)$
$\Rightarrow$ $\exists\, \pi^B = s_0^b s_1 \ldots$ of $stutt(\lceil B \rceil_A)$ such that $\forall\, j \geq 0 (s_j^b \approx t_j^{ab})$ and
$\nexists\, j (stutt(\lceil B \rceil_A), \pi_j^B \models \varphi_2)$ or
$\big(\exists\, j (stutt(\lceil B \rceil_A), \pi_j^B \models \varphi_2)$ and $\exists\, k < j (stutt(\lceil B \rceil_A), \pi_k^B \not\models \varphi_1)\big)$
(by Lemma 3)
$\Rightarrow$ $\exists\, \pi^B$ of $stutt(\lceil B \rceil_A)$ such that $\Rightarrow$ $stutt(\lceil B \rceil_A), \pi^B \not\models \varphi_1 \mathbf{U}\, \varphi_2$
$\Rightarrow$ $stutt(\lceil B \rceil_A) \not\models \varphi$ (by definition of $\models$)

– Assume $\varphi = \neg \varphi_1$ and $stutt(\lceil B \rceil_A) \models \varphi$.
Proof by contradiction:
Assume $A \ll \{B\} \not\models \varphi$
$\Rightarrow$ $\exists\, \pi^{AB} = t_0^{ab} t_1^{ab} \ldots$ of $A \ll \{B\}$ such that
$(A \ll \{B\}), \pi^{AB} \models \varphi_1$
$\Rightarrow$ $\exists\, \pi^B = s_0^b a_1^b \ldots$ of $stutt(\lceil B \rceil_A)$ such that $\forall\, j \geq 0 (s_j^b \approx t_j^{ab})$ and
$stutt(\lceil B \rceil_A), \pi^B \models \varphi_1$ (by Lemma 3)
$\Rightarrow$ $\exists\, \pi^B$ of $stutt(\lceil B \rceil_A)$ such that $stutt(\lceil B \rceil_A), \pi^B \not\models \varphi$
$\Rightarrow$ $stutt(\lceil B \rceil_A) \not\models \varphi$ (by definition of $\models$)

– Assume $\varphi = \varphi_1 \wedge \varphi_2$ and $stutt(\lceil B \rceil_A) \models \varphi$.
$\Rightarrow$ $stutt(\lceil B \rceil_A) \models \varphi_1$ and $stutt(\lceil B \rceil_A) \models \varphi_2$

$\Leftrightarrow \quad (A \ll \{B\}) \models \varphi_1$ and $(A \ll \{B\}) \models \varphi_2$

(following the results from above)

$\Leftrightarrow \quad (A \ll \{B\}) \models \varphi$

□

The proof for Lemma 2 follows the same induction. In fact, all proof steps are similar if we use the following Lemma 4 instead of Lemma 3.

Lemma 4 *For all path $\pi^{AB} = t_0 t_1 \ldots$ in structure $A \ll \{B\}$ there exists a path $\pi^A = s_0 s_1 \ldots$ in structure A such that $\forall\, i \geq 0 \bullet t_i \approx s_i$.*

All paths in the incorporating structure $A \ll \{B\}$ have a corresponding path in structure A which does not incorporate all restrictions of sub-component B.

Proof of Lemma 4:
For all paths $\pi^{AB} = t_0 t_1 \ldots$ in $A \ll \{B\}$ it holds that $\forall\, i \geq 0 \bullet \exists\, s_i, s_{i+1} \in S_A$ such that $(names(A) \lhd t_i) = s_i$ and $(names(A) \lhd t_{i+1}) = s_{i+1}$ and $(s_i, s_{i+1}) \in R_A$ (with definition of path and R_{AB} in Section 4.3). It follows that $\pi^A = s_0 s_1 \ldots$ is a path in A. □

6 Conclusion and Future Work

This paper introduced a compositional proof strategy for Object-Z that is inspired by results for the verification of parallel processes and hardware design (e.g, [Pnu85,GL94]). Based on a value semantics for Object-Z, this approach allows us to prove temporal properties given in Linear Temporal Logic (LTL). It aims at the use of model checking for single proof steps on sub-components. OZ structures, a concept for temporal structures of Object-Z components, is introduced as a semantic foundation of the proof rule.

We adopt a value semantics for Object-Z in order to avoid circularities in the hierarchy of the system specification. However, referring to work by Smith [Smi02], we argue that a system specification on an abstract level given in a value semantics can be refined to a more concrete specification in a reference semantics. Compositional verification, as suggested in this paper, is to be applied on the abstract level focusing on properties of a system's functionality, rather than details of its object-oriented design.

The sub-components to be considered in a single proof step in the compositional strategy are still possibly infinite structures. Thus, to render our approach feasible for model checking, a suitable *abstraction technique* is needed. An abstraction relation over temporal structures maps an infinite structure to a finite (more abstract) one which preserves the properties to be shown. The work by Smith and Winter [SW03] introduces such an abstraction technique for Z. Future work will investigate how this abstraction technique can be adapted for Object-Z and how it can be combined with our compositional proof strategy.

Further investigation is also necessary to develop a proof strategy for systems with a non-fixed hierarchy in which the number of components on each level may change.

Acknowledgements

The authors wish to thank the anonymous referees for their detailed comments on this paper. This work was supported by a University of Queensland External Support Enabling Grant.

References

[Eme90] E. A. Emerson. Temporal and modal logic. In J. van Leeuwen, editor, *Handbook of Theoretical Coomputer Science*, volume B. Elsevier Science Publishers, 1990.

[GL94] O. Grumberg and D.E. Long. Model checking and modular verification. *ACM Transactions on Programming Languages and Systems*, 16(3):843–871, 1994.

[Gri97] A. Griffiths. Modular reasoning in Object-Z. In W. Wong and K. Leung, editors, *Proc. of the Joint 1997 Asia Pacific Software Engineering Conference and International Computer Science Conference*, IEEE, pages 140–149. Computer Society Press, 1997.

[Pnu85] A. Pnueli. In transition from global to modular temporal reasoning about programs. In K. R. Apt, editor, *Logics and Models of Concurrent Systems*, volume 13 of *NATO ASI Series*, pages 123–144. Springer-Verlag, 1985.

[SKS02] G. Smith, F. Kammüller, and T. Santen. Encoding Object-Z in Isabelle/HOL. In D. Bert, J.P. Bowen, M.C. Henson, and K. Robinson, editors, *Proc. of Int. Conf. of Z and B Users (ZB 2002)*, volume 2272 of *LNCS*, pages 82–99. Springer-Verlag, 2002.

[Smi92] G. Smith. *An Object-Oriented Approach to Formal Specification*. PhD thesis, Department of Computer Science, University of Queensland, 1992.

[Smi95a] G. Smith. A fully abstract semantics of classes for Object-Z. *Formal Aspects of Computing*, 7(3):289–313, 1995.

[Smi95b] G. Smith. Reasoning about Object-Z specifications. In *Proc. of the Asia-Pacific Software Engineering Conference (APSEC95)*, IEEE, pages 489–497. Computer Society Press, 1995.

[Smi00] G. Smith. *The Object-Z Specification Language*. Kluwer Academic Publishers, 2000.

[Smi02] G. Smith. Introducing reference semantics via refinement. In C. George and H. Miao, editors, *Proc. on Int. Conference on Formal Engineering Methods (ICFEM 2002)*, volume 2495 of *LNCS*, pages 588–599. Springer-Verlag, 2002.

[Spi92] J.M. Spivey. *The Z Notation - A Reference Manual*. Prentice Hall, 1992.

[SW03] G. Smith and K. Winter. Proving temporal properties of Z specificatons using abstraction. In *3rd International Conference of Z and B USers (ZB 2003)*, LNCS. Springer-Verlag, 2003. This volume.

[WB92] J.C.P. Woodcock and S.M. Brien. $\mathcal{W}$: A logic for Z. In *Z User Workshop (ZUM'92)*, Workshops in Computing, pages 77–98. Springer-Verlag, 1992.