Inheritance of Temporal Logic Properties

Heike Wehrheim

Universität Oldenburg, Fachbereich Informatik,
26111 Oldenburg, Germany
wehrheim@informatik.uni-oldenburg.de

Abstract. Inheritance is one of the key features for the success of object-oriented languages. Inheritance (or specialisation) supports incremental design and re-use of already written specifications or programs. In a formal approach to system design the interest does not only lie in re-use of class definitions but also in re-use of *correctness proofs*. If a provably correct class is specialised we like to know those correctness properties which are preserved in the subclass. This can avoid re-verification of already proven properties and may thus substantially reduce the verification effort.
In this paper we study the question of inheritance of correctness properties in the context of state-based formalisms, using a temporal logic (CTL) to formalise requirements on classes. Given a superclass and its specialised subclass we develop a technique for computing the set of formulas which are preserved in the subclass. For specialisation we allow addition of attributes, modification of existing as well as extension with new methods.

1 Introduction

Object-oriented languages nowadays play a major role in software development. This has even further increased with the advent of component-based programming. Object-orientation supports encapsulation (of data and operations within a class) and re-use of already written specifications or code. The latter aspect is (mainly) achieved by the concept of *inheritance* or *specialisation* allowing to derive new classes from existing ones.

In a formal approach to software development classes (or systems) are specified using a formal specification language. A formal specification having a precisely defined semantics opens the possibility of *proving correctness* of the design with respect to certain properties. Ideally, the design should be complemented by a number of formally stated requirements which the system is supposed to fulfill. In this context the concept of specialisation poses a new question: besides re-using specifications can we also re-use correctness proofs? A positive answer to this question would save us from a large verification effort: we would not have to redo all the correctness proofs. Within the area of program verification, especially of Java programs, this question has already been tackled by a number of researchers [9, 12, 8]. In these approaches correctness properties are mainly

E. Najm, U. Nestmann, and P. Stevens (Eds.): FMOODS 2003, LNCS 2884, pp. 79–93, 2003.

formulated in Hoare logic, and the aim is to find proof rules which help to deduce subclass properties from superclass properties. In order to get correctness of these rules it is required that the subclass is a *behavioural subtype* [10] of the superclass. This assumption is also the basis of [15] which studies preservation of properties in an event-based setting with correctness requirements formulated as CSP processes. In this paper we look at inheritance of properties to specialised classes form another point of view. Classes are not written in a programming language or as a CSP process but are defined in a general (state-based) mathematical framework. Correctness requirements on classes are formalised in a temporal logic (CTL). The major difference to previous work in this area is, however, that we do not restrict our considerations to specific subclasses (namely subtypes), but start with an arbitrary subclass and *compute* the set of properties which are preserved.

The class descriptions we employ describe *active* objects, providing as well as refusing services (method invocations) at certain points in time. To achieve this, every method is characterised by a *guard* defining its applicability and an *effect* defining the effect of its execution on attributes. A (passive) class which never refuses any call can also easily be described by setting all guards to *true*. For convenience of reading we will write class specifications in a formalism close to Object-Z [13], however, will work with a more general, formalism-independent definition of classes. There are two kinds of properties we are interested in (manifested in the choice of atomic propositions): first, properties over values of attributes, e.g. class invariants, and second, properties about the availability of services, e.g. that a certain method will always be executable[1].

Specialisation is not defined as an *operator* in a language but as a general relationship between class definitions[2]. Specialisation allows the addition of attributes, the modification of existing methods and the extension with new methods. It does not allow deletion of attributes or methods which would also be rather unusual for an inheritance operator.

The basic question we investigate in this paper can thus be formulated as follows: given two class definitions A and C, where C is a specialisation of A, which CTL-formulas holding for A hold for C as well? In general, a specialised class may inherit none of A's properties since it is possible to modify all methods and thus completely destroy every property of the superclass. Hence it is necessary to find out which modifications of methods (or extensions with new methods) influence the holding of CTL formulas. For this, we use a technique close to the *cone of influence reduction* technique used in hardware verification [2]. For every modified or new method we compute its *influence set*, i.e. the set of variables it directly (or indirectly) influences. Atomic propositions depending on variables in this influence set will thus potentially hold at different states in C than in A. Therefore formulas over atomic propositions of the influence set might not be inherited from A to C. All formulas independent of the in-

[1] Since methods are guarded, they can in general be refused in some states.

[2] Depending on the concrete formalism employed, inheritance might induce a specialisation relationship among classes in the sense used here.

fluence set are, however, preserved. This result is proven by showing that the two classes are *stuttering equivalent* with respect to the set of unchanged atomic propositions. Stuttering equivalence has been introduced by [1] for comparing Kripke structures according to the set of CTL (CTL*) formulas they fulfill. There are two different variants of stuttering equivalence (divergence-blind and divergence-sensitive) which correspond to two different interpretations of CTL [4]. We choose only one of them and will argue why this one is appropriate for our study of property inheritance and why the corresponding CTL interpretation might be more adequate for classes.

The paper is structured as follows. In the next section we discuss the basics necessary for formulating the result. We introduce the logic CTL, formalise class definitions and supply them with a semantics by assigning a Kripke structure to every class. We furthermore define divergence-blind stuttering equivalence and state the result that stuttering equivalent Kripke structures fulfill the same set of CTL-X (without Next) formulas. Section 3 starts with defining and explaining the influence set of methods. It furthermore states and proves the main result about inheritance of temporal logic properties over variables not in the influence set. Some examples further illustrate the result. Section 4 concludes.

2 Background

In this section we introduce the definitions necessary for our study of property inheritance. We describe class definitions and give an example of a class which we later use when looking at properties and specialisations. Furthermore we define the syntax and semantics of the logic CTL and state a result relating stuttering equivalence and CTL formulas.

2.1 Classes

Classes consist of attributes (or variables) and methods to operate on attributes. Methods may have input parameters and may return values, referred to as output parameters. We assume variables and input/output parameters to have values from a global set D. A *valuation* of a set of variables V is a mapping from V to D, we let $R_V = \{\rho : V \to D\}$ stand for the set of all valuations of V, the set of valuations of input parameters *Inp* and output parameters *Out* can be similarly defined.

A class is thus characterised by

- A set of *attributes* (or variables) V,
- an *initial valuation* of V to be used upon construction of objects: $I : V \to D$, and
- a set of methods (names) M with input and output parameters from a set of inputs *Inp* and a set of outputs *Out*. For simplicity we assume *Inp* and *Out* to be global. Each $m \in M$ has a guard $guard_m : R_V \times R_{Inp} \to \mathbb{B}$ ($\mathbb{B}$ are booleans) and an effect $effect_m : R_V \times R_{Inp} \to R_V \times R_{Out}$. The guard specifies the states in which the method is executable and the effect determines the outcome of the method execution.

The semantics of a class is defined in terms of *Kripke structures*, the standard model on which temporal logics are interpreted. A Kripke structure is similar to a transition system.

Definition 1. *Let AP be a nonempty set of atomic propositions. A* Kripke structure *$K = (S, S_0, \rightarrow, L)$ over AP consists of a finite set of states S, a set of initial states $S_0 \subseteq S$, a transition relation $\rightarrow \subseteq S \times S$ and a labeling function $L : S \rightarrow 2^{AP}$.*

Note that we do not require totality of the transition relation, i.e. there may be states $s \in S$ such that there is no s' with $s \rightarrow s'$.

The basic properties we like to observe about a class (or an object) are its state and the availability of methods. Thus the atomic propositions AP_A that we consider for a class $A = (V, I, M, (guard_m)_{m \in M}, (effect_m)_{m \in M})$ are

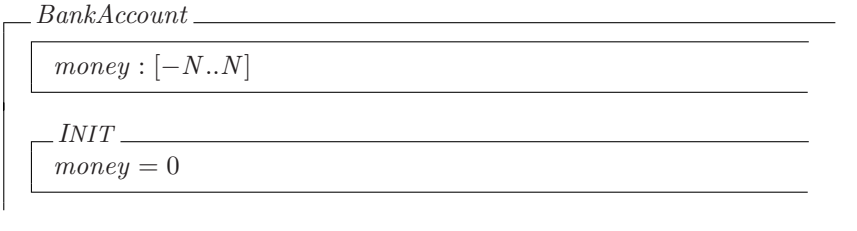
- $v = d$, $v \in V, d \in D$ and
- $enabled(m)$, $m \in M$.

We assume method execution to be atomic, i.e. we do not distinguish beginnings and ends of methods and do not allow concurrent access to an object.

Definition 2.
The semantics *of (an object of) a class $A = (V, I, M, (guard_m)_{m \in M}, (effect_m)_{m \in M})$ is the Kripke structure $K = (S, S_0, \rightarrow, L)$ over AP_A with*

- $S = R_V$,
- $S_0 = I$,
- $\rightarrow = \{(s, s') \mid \exists\, m \in M, \exists\, \rho_{in} \in R_{Inp}, \rho_{out} \in R_{Out} : guard_m(s, \rho_{in}) \wedge effect_m(s, \rho_{in}) = (s', \rho_{out})\}$,
- $L(s) = \{v = d \mid s(v) = d\} \cup \{enabled(m) \mid \exists\, \rho_{in} \in R_{Inp} : guard_m(s, \rho_{in})\}$.

Since the atomic propositions do not refer to inputs and outputs of methods, they are not reflected in the semantics. The following example gives a class definition of a simple bank account with methods for disposal and withdrawal. To enhance readability we have chosen a presentation which is close to Object-Z. The first box (schema) describes the attributes (with their domains, N being a constant), the *Init* schema defines the initial valuation and schemas labeled *guard* and *effect* define guards and effects of the methods. Primed variables refer to attribute values in the next state, the notation *var*? describes an input parameter and $\Delta(\ldots)$ denotes the set of variables which are changed upon execution of particular methods.



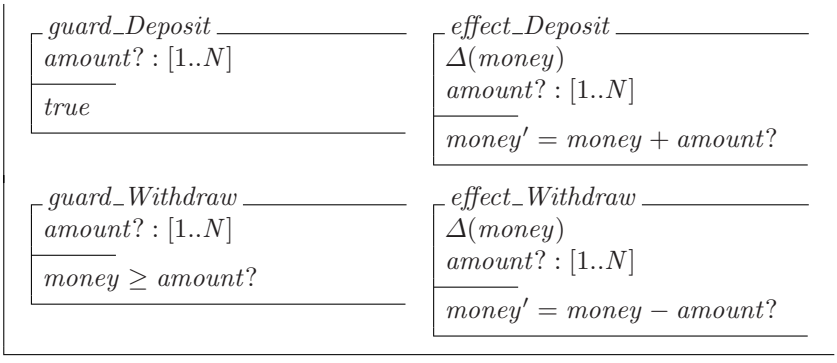


Figure 1 shows the Kripke structure (without L) of class *BankAccount*. The numbers indicate the values of attribute *money*. All states with $money < 0$ are unreachable. The upper arrows correspond to executions of *Deposit*, the lower to those of *Withdraw*.

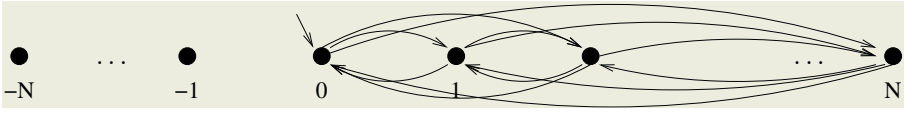


Fig. 1. Kripke structure of class *BankAccount*.

2.2 CTL

The temporal logic that we use for specifying properties of classes is CTL [5]. The temporal operators of CTL consist of a path quantifier (E for existential and A for universal quantification) plus operators for expressing eventuality (F), globally (G), next state (X), and until (U).

Definition 3. *The set of CTL formulas over AP is defined as the smallest set of formulas satisfying the following conditions:*

- $p \in AP$ *is a formula,*
- *if* φ_1, φ_2 *are formulas, so are* $\neg\varphi_1$ *and* $\varphi_1 \vee \varphi_2$*,*
- *if* φ *is a formula, so are* $EX\varphi$, $EF\varphi$, $EG\varphi$*,*
- *if* φ_1, φ_2 *are formulas, so is* $E[\varphi_1\ U\ \varphi_2]$*.*

As usual, other operators can be defined as abbreviations, e.g. $true = (p \vee \neg p)$ for some arbitrary proposition p, $false = \neg true$, $\varphi_1 \wedge \varphi_2 = \neg(\neg\varphi_1 \vee \neg\varphi_2)$, $AG\ \varphi = \neg EF\ \neg\varphi$ and $AX\ \varphi = \neg EX\ \neg\varphi$[3]. CTL-X is the set of CTL formulas without the next-operators EX and AX.

For the interpretation of CTL formulas we look at *paths* of the Kripke structure.

[3] The formulae $AF\varphi_2$ and $A[\varphi_1\ U\ \varphi_2]$ have been omitted here since, under the interpretation given in Definition 5, they both coincide with φ_2.

Definition 4. *Let $K = (S, S_0, \rightarrow, L)$ be a Kripke structure over AP.*

- *A* path *is a nonempty finite or infinite sequence of states $\pi = s_0 s_1 s_2 \ldots$ such that $s_i \rightarrow s_{i+1}$ holds for all $i > 0$ (in case of finite paths up to some n). For a path $\pi = s_0 s_1 s_2 \ldots$ we write $\pi[i]$ to stand for s_i. We define the* length *of a finite path $\pi = s_0 \ldots s_{n-1}$, $|\pi|$, to be n and set the length of infinite paths to ∞.*
- *The set of paths starting in $s \in S$ is*

$$paths(s) = \{s_0 s_1 s_2 \ldots \mid s_0 s_1 s_2 \ldots \text{ is a path and } s_0 = s\}$$

Note that paths need not be maximal or infinite. In this, we deviate from the standard interpretation of CTL. Usually, only infinite paths are considered, whereas we also look at prefixes of such infinite paths. The interpretation we give below has, however, also been considered by [6] and [4].

Definition 5. *Let $K = (S, S_0, \rightarrow, L)$ be a Kripke structure and φ a CTL formula, both over AP. K satisfies φ ($K \models \varphi$) iff $K, s_0 \models \varphi$ holds for all $s_0 \in S_0$, where $K, s \models \varphi$ is defined as follows:*

- *$K, s \models p$ iff $p \in L(s)$,*
- *$K, s \models \neg\varphi$ iff not $K, s \models \varphi$,*
- *$K, s \models \varphi_1 \vee \varphi_2$ iff $K, s \models \varphi_1$ or $K, s \models \varphi_2$,*
- *$K, s \models EX\ \varphi$ iff $\exists\, \pi \in path(s), |\pi| > 1 \wedge K, \pi[1] \models \varphi$,*
- *$K, s \models EF\ \varphi$ iff $\exists\, \pi \in path(s), \exists\, k \geq 0 : K, \pi[k] \models \varphi$,*
- *$K, s \models EG\ \varphi$ iff $\exists\, \pi \in path(s), \forall\, k \geq 0, k < |\pi| : K, \pi[k] \models \varphi$,*
- *$K, s \models E[\varphi_1\ U\ \varphi_2]$ iff $\exists\, \pi \in path(s), \exists\, k \geq 0 : K, \pi[k] \models \varphi_2$ and $\forall\, j, 0 \leq j < k : K, \pi[j] \models \varphi_1$.*

This interpretation is more convenient for our study in two respects. One reason will be discussed later when we look at preserved properties: Under the standard interpretation fewer properties are inherited to specialised classes. For the second point consider again the class specification *BankAccount*. Its Kripke structure K satisfies the following properties:

S1 $AG\ (money \geq 0)$,
S2 $AG\ (enabled(Deposit))$,
L $AG\ EF\ (enabled(Withdraw))$.

The formula $AF\ (money > 0)$ does, however, not hold for the Kripke structure. This is due to the non-standard semantics of CTL given here since it does also consider non-maximal paths. The path consisting of just the initial state does not satisfy the formula; the formula would hold if we restrict paths to maximal (infinite) ones. Since the execution of methods, however, depends on calls of methods from the outside and is not under control of the class, we actually can never be sure that steps to next states are taken. Thus, an object of class *BankAccount* might actually never be used and thus we might never reach a state with $money > 0$.

2.3 Stuttering Equivalence

The last part of the background section concerns the result relating stuttering equivalence with CTL formulas. Stuttering equivalence will be the relation used to relate specialised classes with superclasses. Stuttering equivalence is the state-based analog of branching bisimulation [14].

Definition 6. *Let* $K_i = (S_i, S_{0,i}, \rightarrow_i, L_i)$, $i = 1, 2$, *be Kripke structures over* AP_1, AP_2, *respectively.* K_1 *and* K_2 *are* divergence blind stuttering equivalent *with respect to a set of atomic propositions* $AP \subseteq AP_1 \cap AP_2$ ($K_1 \approx_{AP} K_2$) *iff there is a relation* $B \subseteq S_1 \times S_2$ *which satisfies the following conditions:*

1. $\forall\, s_1 \in S_{0,1}\, \exists\, s_2 \in S_{0,2} : (s_1, s_2) \in B$, *and vice versa,*
   $\forall\, s_2 \in S_{0,2}\, \exists\, s_1 \in S_{0,1} : (s_1, s_2) \in B$,

and for all $(r, s) \in B$

2. $L_1(r) \cap AP = L_2(s) \cap AP$,
3. $r \rightarrow r' \Rightarrow$
   $\exists\, s_0, s_1, \ldots, s_n, n \geq 0$ *such that* $s_0 = s, \forall\, 0 \leq i < n : s_i \rightarrow s_{i+1}$
   $\wedge\, (r, s_i) \in B$, *and* $(r', s_n) \in B$, *and vice versa,*
   $s \rightarrow s' \Rightarrow$
   $\exists\, r_0, r_1, \ldots, r_n, n \geq 0$ *such that* $r_0 = r, \forall\, 0 \leq i < n : r_i \rightarrow r_{i+1}$
   $\wedge\, (r_i, s) \in B$, *and* $(r_n, s') \in B$.

Stuttering equivalence allows stuttering steps during the simulation of one step of K_1 by K_2 and vice versa. All the intermediate states in the stuttering step (e.g. $s_0, \ldots s_{n-1}$) still have to be related to the starting state of the other structure, i.e. the atomic propositions from AP may not change during stuttering. Since we allow stuttering a property holding in the *next* state of K_1 possibly does not hold in the next but only in later states of K_2. Thus concerning CTL formulas, stuttering equivalent Kripke structures only satisfy the same set of CTL-X formulas.

Theorem 1. *Let* K_1, K_2 *be two Kripke structures such that* $K_1 \approx_{AP} K_2$, *and let* φ *be a CTL-X formula over AP. Then the following holds:*

$$K_1 \models \varphi \quad \textit{iff} \quad K_2 \models \varphi$$

Proof. See [1, 4].

3 Inheriting Properties

Usually, inheritance of properties is studied for subclasses which are behavioural subtypes of their superclasses. Here we pursue a different line of thought. We allow to arbitrarily add new methods as well as change old methods and impose no a priori restrictions on these extensions and modifications. Since this may in general lead to a specialised class in which lots of properties of the superclass

do not hold anymore, we have to compute the set of formulas which are preserved. Instead of restricting the subclasses considered, as is usually done, we thus restrict the set of formulas.

First, we have to define what it means for a class to be a *specialisation* of another class. Instead of defining a particular operator we simply define a relation. In the following we always assume that the classes A and C have components $(V_A, I_A, M_A, (guard_m^A)_{m\in M_A}, (effect_m^A)_{m\in M_A})$ and $(V_C, I_C, M_C, (guard_m^C)_{m\in M_C}, (effect_m)^C_{m\in M_C})$, respectively.

Definition 7. *Let A and C be classes. C is a* specialisation *of A if $V_A \subseteq V_C$, $M_A \subseteq M_C$ and $I_C \mid_{V_A} = I_A$.*

In general, a subclass derived from a superclass by inheritance will be a specialisation of the superclass. C may change method definitions as well as introduce new methods: the changed methods are those methods m in M_A for which either $guard_m^A \neq guard_m^C$ or $effect_m^A \neq effect_m^C$, the new methods are those in $M_C \setminus M_A$. To keep matters simple we assume that the subclass does not change initial values of old attributes.

For computing the set of preserved formulas we have to find out which atomic propositions are affected by the modifications or extensions made in the specialised class. For this we need to compute the *influence set* of methods, i.e. the set of variables that may directly or indirectly be affected by method execution. First, we define the set of variables that a method modifies, uses and depends on. Intuitively, a method depends on a variable if the evaluation of its guard is affected by the value of the variable; it uses a variable if the value of the variable is used to determine output or changes to the attributes; and it modifies a variable if the value of the variable is changed upon execution of the method.

Note that in the definition below, $first(effect_m(..., ...))$ refers to the new state of attributes after execution of a method m (the first component of a pair from $R_V \times R_{Out}$).

Definition 8. *Let $A = (V, I, M, (guard_m)_{m\in M}, (effect_m)_{m\in M})$ be a class.*

A method $m \in M$ modifies *a variable $v \in V$ if there are $\rho \in R_V, \rho_{in} \in R_{Inp}$ such that $first(effect_m(\rho, \rho_{in}))(v) \neq \rho(v)$.*

Execution of $m \in M$ depends on *$v \in V$ if there are $\rho_1, \rho_2 \in R_V, \rho_{in} \in R_{Inp}$ such that $\rho_1 \mid_{V\setminus\{v\}} = \rho_2 \mid_{V\setminus\{v\}}$ and $guard_m(\rho_1, \rho_{in}) \neq guard_m(\rho_2, \rho_{in})$.*

A method $m \in M$ uses *$v \in V$ if there are $\rho_1, \rho_2 \in R_V, \rho_{in} \in R_{Inp}$ such that $\rho_1 \mid_{V\setminus\{v\}} = \rho_2 \mid_{V\setminus\{v\}}$ and $effect_m(\rho_1, \rho_{in}) \neq effect_m(\rho_2, \rho_{in})$.*

We let $mod^A(m)$ denote the set of variables m modifies in A, $depends^A(m)$ the set it depends on, and $uses^A(m)$ the set of variables it uses. When clear from the context we omit the index for the class. For the formalism used in the *BankAccount* example these sets can be (over-)approximated by using the following rules: $mod(m)$ is the set of variables appearing in the Δ-list of $effect_m$, $uses(m)$ are all variables appearing in $effect_m$, and $depends(m)$ are those appearing in $guard_m$. Thus we for instance have $mod(Withdraw) = \{money\}$ and $depends(Deposit) = \varnothing$. Over-approximation does not invalidate the result about preservation of properties proven below.

There is a tight connection between the variables a method uses or depends on and its guard and effect, as expressed by the next proposition.

Proposition 1. *Let $V' \subseteq V$ be a set of variables, $s, s' \in R_V$ two states such that $s \mid_{V'} = s' \mid_{V'}$ and let $m \in M$ be a method.*

If $depends(m) \subseteq V'$ then $\forall\, \rho_{in} \in R_{Inp} : guard_m(s, \rho_{in}) \Leftrightarrow guard_m(s', \rho_{in})$.

If $uses(m) \subseteq V'$ then $\forall\, \rho_{in}, \rho_{out}, \rho'_{out}$ such that $effect_m(s, \rho_{in}) = (t, \rho_{out})$, $effect_m(s', \rho_{in}) = (t', \rho'_{out})$, we get $t \mid_{V'} = t' \mid_{V'}$ and $\rho_{out} = \rho'_{out}$.

The set of methods which potentially influence the holding of formulas are those changed in the specialised class C (compared to the superclass A) or new in C. In the following we denote this set of methods by N. We are interested in the effect these methods in N may have wrt. the atomic propositions holding in (the Kripke structure of) A. If a method in N modifies a variable which is used in a guard of another method in A, then in C this method might be enabled at different states and its execution might produce different results. This leads to the following definition of *influence set*.

Definition 9. *Let A and C be classes such that C is a specialisation of A. The* influence set *of a set of methods $N \subseteq M_C$, $Infl^{A,C}(N)$, is the smallest set of variables satisfying the following two conditions:*

1. $\forall\, m \in N : mod^A(m) \cup mod^C(m) \subseteq Infl^{A,C}(N)$,
2. *if $v \in Infl^{A,C}(N)$ and there is some $m' \in M_A$ such that $v \in depends^A(m')$ or $v \in uses^A(m')$, then $mod^A(m') \cup mod^C(m') \subseteq Infl^{A,C}(N)$.*

Note that in 2. we refer to class A only since A and C agree on methods outside of N anyway. In the sequel we always write $Infl(N)$ instead of $Infl^{A,C}(N)$. The influence set is recursively defined since modifications made to a variable may propagate to other variables. At the end of this section an example illustrating this propagation of changes can be found.

The influence set computation is similar to the cone of influence computation used as an abstraction technique in hardware verification [2]. However, while the cone of influence is computed for the set of variables under interest (and the remaining variables can then be abstracted away), we take the opposite view here: the influence set tells us which atomic propositions might *not* hold anymore, and the set of preserved formulas has to be restricted to the remaining atomic propositions.

Given a set of non-modified variables V' and non-modified methods M' we define the atomic propositions over V' and M' to be

$$\begin{aligned} AP_{V',M'} = &\{v' = d \in AP \mid v' \in V'\} \\ &\cup \{enabled(m') \in AP \mid m' \in M' \wedge depends(m') \subseteq V'\} \end{aligned}$$

Starting from a temporal logic formula over atomic propositions in A and having given a specialised class C, the set of non-modified variables and methods is thus $V' = V_A \setminus (N)$ and $M' = M_A \setminus N$ (where N is the set of methods modified or added in C).

The following theorem now states the main result of this paper. Concerning the atomic propositions over non-modified variables and methods, a class A and its specialisation C are stuttering equivalent.

Theorem 2. *Let A and C be classes such that C is a specialisation of A. Let N denote the set of methods changed in C or new in C, and set V' to $V_A \setminus Infl(N)$, M' to $M_A \setminus N$. Then the following holds:*

$$K_C \approx_{AP_{V',M'}} K_A$$

Intuitively, all steps corresponding to the execution of new or changed methods are now stuttering steps under the restricted set of atomic propositions.

Together with Theorem 1 this gives us the following corollary:

Corollary 1. *Let A and C be classes such that C is a specialisation of A. Let N denote the set of methods changed in C or new in C, and set V' to $V_A \setminus Infl(N)$, M' to $M_A \setminus N$.*
Let φ be a CTL-X formula over $AP_{V',M'}$. Then the following holds:

$$K_A \models \varphi \;\Leftrightarrow\; K_C \models \varphi$$

In particular, class C preserves *all* of A's properties if it does not change existing methods and new methods only modify new attributes, as expressed by the following corollary.

Corollary 2. *Let A and C be classes such that C is a specialisation of A. Let N denote the set of methods changed in C or new in C.*
If $N \subseteq M_C \setminus M_A$ and $mod^A(N) \cup mod^C(N) \subseteq V_C \setminus V_A$ then the following holds for all formulas $\varphi \in$ CTL-X over AP_{V_A,M_A}:

$$K_A \models \varphi \Leftrightarrow K_C \models \varphi$$

In this case, class C can also be seen as a behavioural subtype of A in the sense of [10] (extension rule).

Proof of Theorem 2. Let $K_A = (S_A, S_{0,A}, \rightarrow_A, L_A)$ and $K_C = (S_c, S_{0,C}, \rightarrow_C, L_C)$ be the Kripke structures of A and C. The relation showing divergence blind stuttering equivalence between K_A and K_C is $B = \{(s_A, s_C) \mid s_A \mid_{V'} = s_C \mid_{V'}\}$, where $V' = V_A \setminus Infl(N)$. We have to check conditions 1 to 3.

1. Holds since C is a specialisation of A and hence $I_C \mid_{V_A} = I_A$ which implies $I_C \mid_{V'} = I_A \mid_{V'}$.

Let $(s_A, s_C) \in B$.

2. Concerning $AP_{V',M'}$ the same set of atomic propositions hold in related states:

$$
\begin{aligned}
L_A(s_A) \cap AP_{V',M'} &= \{v' = d \mid v' \in V', d \in D, s_A(v') = d\} \\
&\quad \cup \{enabled(m') \mid m' \in M', depends^A(m') \subseteq V' \\
&\qquad \wedge \exists\, \rho_{in} \in R_{Inp} : guard^A_{m'}(s_A, \rho_{in})\} \\
&= \{v' = d \mid v' \in V', d \in D, s_C(v') = d\} \qquad (1) \\
&\quad \cup \{enabled(m') \mid m' \in M', depends^C(m') \subseteq V' \\
&\qquad \wedge \exists\, \rho_{in} \in R_{Inp} : guard^C_{m'}(s_C, \rho_{in})\} \qquad (2) \\
&= L_C(s_C) \cap AP_{V',M'}
\end{aligned}
$$

Line 1 holds since $s_A \mid_{V'} = s_C \mid_{V'}$ and line 2 holds since $guard^A_{m'} = guard^C_{m'}$ and thus also $depends^A(m') = depends^C(m')$.

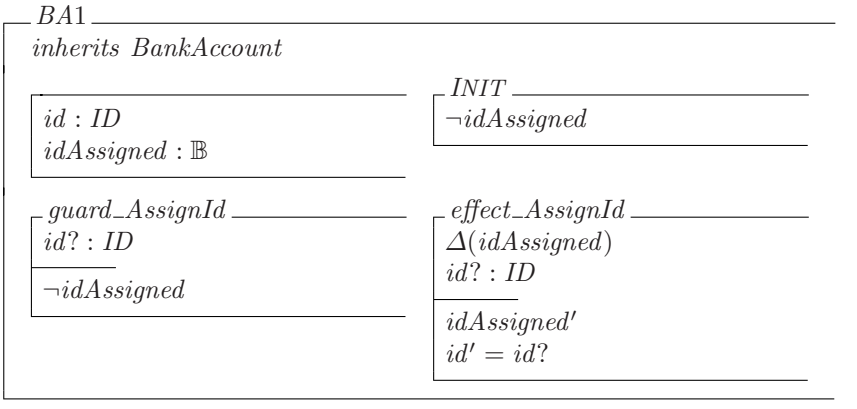
3. Assume $s_A \to s'_A$. Then there is some $m \in M_A$, some $\rho_{in} \in R_{Inp}, \rho_{out} \in R_{Out}$ such that $guard^A_m(s_A, \rho_{in})$ and $effect^A_m(s_A, \rho_{in}) = (s'_A, \rho_{out})$. There are now three cases to consider: m is a method which is unchanged in C and a) m does only depend on and does only use variables in V', or b) m uses or depends on variables in $Infl(N)$, and third case $m \in N$. In the first case the transition in A is matched by a corresponding transition in C (the same method is executed), in the other two cases the A-step is a stuttering step; it essentially changes atomic propositions not in $AP_{V',M'}$.
   (a) $m \in M_A \setminus N$ and $depends^A(m) \cup uses^A(m) \subseteq V'$ (and thus $depends^C(m) \cup uses^C(m) \subseteq V'$): By Proposition 1 $guard^C_m(s_C, \rho_{in})$ and $\exists\, s'_C : s'_C \mid_{V'} = s'_A \mid_{V'}$ and $effect^C_m(s_c, \rho_{in}) = (s'_C, \rho_{out})$, i.e. $s_C \to s'_C$, and by definition of B, $(s'_A, s'_C) \in B$.
   (b) $m \in M_A \setminus N$ and $depends^A(m) \not\subseteq V'$ or $uses^A(m) \not\subseteq V'$:
   Then there is some $v \in depends^A(m)$ or $v \in uses^A(m)$ such that $v \in Infl(N)$. Since m is unchanged, we get by definition of the influence set $mod^A(m) \subseteq Infl(N)$. Hence $s_A \mid_{V'} = s'_A \mid_{V'}$ and therefore $(s'_A, s_C) \in B$ (the A-step is matched by 0 C-steps).
   (c) $m \in N$ (a changed method):
   Then $mod^A(m) \subseteq Infl(N)$ and we can apply the same reasoning as in the last case.
4. Reverse case: $s_C \to s'_C$. Then there is some $m \in M_C$, some $\rho_{in} \in R_{Inp}, \rho_{out} \in R_{Out}$ such that $guard^C_m(s_C, \rho_{in})$ and $effect^A_m(s_C, \rho_{in}) = (s'_C, \rho_{out})$. We essentially get the same three cases as before (except for the fact that in the third case we can now have a new as well as a changed method) and can therefore reason analogously. □

We illustrate the result by our bank account example. Below two specialisations of class *BankAccount* are given. Both are defined using inheritance which here simply means that all definitions of *BankAccount* (i.e. attributes and methods) also apply for the specialised classes. The first class adds two new attributes and one method to *BankAccount*.

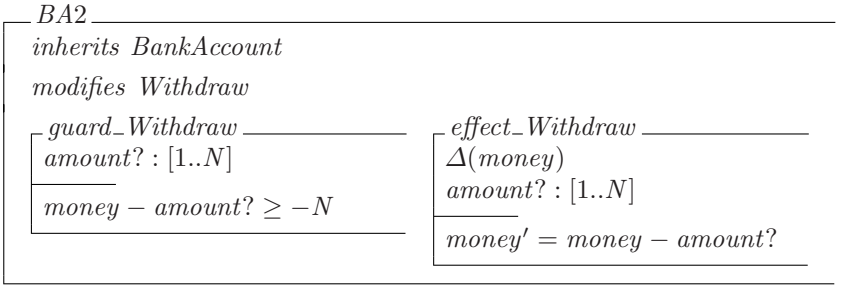
The new method *AssignId* assigns an identification number to a bank account. Once this number has been assigned method *AssignId* is disabled. The new method does not change the old variable:

$Infl^{BankAccount,BA1}(AssignId) = \{idAssigned, id\}$

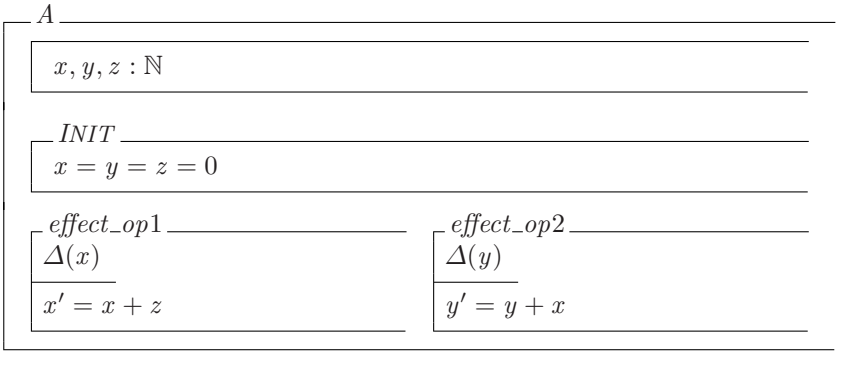
All of our properties S1, S2 and L are thus preserved.



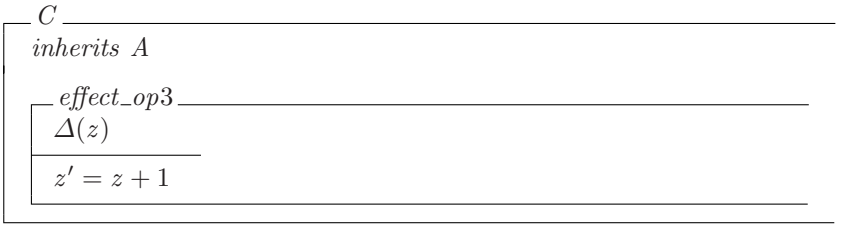
Consider on the other hand the second specialised class $BA2$: It modifies method *Withdraw* which means that the definition of *Withdraw* in *BankAccount* is replaced by the new definition. In this bank account the user is allowed to overdraw his account (up to $-N$).



We have $Infl^{BankAccount,BA2}(Withdraw) = \{money\}$. Thus our theorem only tells us that S2 is preserved but it tells us nothing about S1 and L. While the liveness property L does still hold, S1 is invalidated in $BA2$.



The last (rather artifical) example shows why we need the inclusion of variables into the influence set which are only indirectly affected by a new or changed method. Consider the following class A with three attributes x, y and z, where x and y are modified by operations $op1$ and $op2$, respectively.
The property $AG\ (y = 0)$ holds for this class. Next we make a specialisation of the class which adds one new method modifying z.



This method indirectly influences variable y: the value of y depends on x and x depends on z, $mod^C(op3) = \{z\}$, and $Infl^C(op3) = \{x, y, z\}$. Hence no formulas with atomic propositions talking about x, y or z are preserved, and it can in fact be seen that property $AG\ (y = 0)$ does not hold for class C anymore.

Finally, there is one issue which remains to be discussed: the choice for the specific non-standard interpretation of CTL. Usually, CTL is interpreted on maximal paths. Paths are always infinite which is achieved by requiring the transition relation $\rightarrow$ to be total. In order to get a result similar to Theorem 1 in this case, an additional condition is needed in the definition of stuttering equivalence (see [3]): whenever two states r and s are related via B, r has *infinite stuttering* if and only if s has. Infinite stuttering means that there is an infinite path $rr_1r_2r_3\ldots$ from r such that all r_i are related with s. Intuitively this corresponds to an infinite sequence of method applications which do not change the atomic propositions under consideration. This extended definition is referred to as *divergence sensitive stuttering equivalence*. It is needed for preservation of CTL-X properties under the standard interpretation since for every path in r we need a corresponding path in s. If we only consider maximal paths we may lack a corresponding path when r has infinite stuttering but s does not. When using the non-standard interpretation a path may also consist of a single state.

In principle, we could have used this extended definition of stuttering equivalence together with the standard CTL semantics. This has, however, one significant drawback: the specialised class C then has to be divergence sensitive stuttering equivalent to the superclass A in order to inherit properties from A. This is in general not the case: for instance, a new method in C (corresponding to the stuttering steps) that may be executed infinitely often (e.g. when its guard is true) leads to infinite stuttering. Since this method is not present in A there may be no corresponding infinite stuttering in A. Thus, divergence-blind stuttering equivalence is more adequate for comparing specialised class with superclass. If nevertheless the standard interpretation should be used then one possibility is to add one extra method to every class. This method should always be enabled (guard = true) and should not change any variable. Thus every state has infinite stuttering and the necessary conditions are trivially fulfilled.

4 Conclusion

In this paper, we investigated the question of property inheritance from superclasses to specialised classes. Unlike other approaches we did not assume any specific relationship as for instance subtyping between superclass and subclass. Instead, we showed how to *compute* the set of preserved properties when given a class and its specialisation.

Related Work. The work most closest to ours, both from a technical point of view and from its overall aim, is that of cone-of-influence reduction in hardware [2] and program slicing [7, 11] in software verification or specification analysis. Both techniques start from a large system (circuit, program or specification) and apply the reduction technique to obtain a smaller one on which verification/analysis takes place. We start with the small system (class) and compute the influence set to find out which properties are preserved in the larger system.

The techniques used to prove the correctness of the main result are nevertheless quite similar: the cone-of-influence reduction technique shows bisimilarity of large and reduced system and thus preserves all CTL formulas; program slicing, as employed in [7], achieves a trace-based variant of stuttering equivalence between large and reduced system and thus preserves all LTL-X formulas.

Future Work. This work is only a first step towards a practical use of the computation of preserved properties. The most important extension concerns the consideration of more than one class. This can be done by combining the technique presented here with *compositional* verification techniques. In the setting of Object-Z this could for instance be the work of [16]. Another important issue is the actual computation of the influence set: in order to be able to apply our technique the influence set should be as small as possible and thus purely syntax-oriented computations might not be practical.

Acknowledgement

Thanks to one of the anonymous referees for its detailed comments which helped to improve the paper.

References

1. M.C. Browne, E.M. Clarke, and O. Grumberg. Characterising Finite Kripke Structures in Propositional Temporal Logic. *Theoretical Computer Science*, 59:115–131, 1988.
2. E. Clarke, O. Grumberg, and D. Peled. *Model checking*. MIT Press, 1999.
3. D. Dams. Flat Fragments of CTL and CTL*: Separating the Expressive and Distinguishing Powers. *Logic Journal of IGPL*, 7(1):55–78, 1999.
4. Rocco De Nicola and Frits Vaandrager. Three logics for branching bisimulation. In *Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 118–129. IEEE, Computer Society Press, 1990.

5. E.A. Emerson and E.M. Clarke. Using branching time temporal logic to synthesize synchronisation skeletons. *Science of Computer Programming*, 2(3):241–266, 1982.
6. E.A. Emerson and J. Srinivasan. Branching time temporal logic. In J.W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, volume 354 of *LNCS*, pages 123–171. Springer, 1988.
7. J. Hatcliff, M. Dwyer, and H. Zheng. Slicing software for model construction. *Higher-order and Symbolic Computation*. To appear.
8. K. Huizing and R. Kuiper. Reinforcing fragile base classes. In A. Poetzsch-Heffter, editor, *Workshop on Formal Techniques for Java Programs, ECOOP 2001*, 2001.
9. G.T. Leavens and W.E. Weihl. Specification and verification of object-oriented programs using supertype abstraction. *Acta Informatica*, 32:705–778, 1995.
10. B. Liskov and J. Wing. A behavioural notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811 – 1841, 1994.
11. T. Oda and K. Araki. Specification slicing in formal methods of software development. In *Proceedings of the Seventeenth Annual International Computer Software & Applications Conference*, pages 313–319. IEEE Computer Society Press, 1993.
12. A. Poetzsch-Heffter and J. Meyer. Interactive verification environments for object-oriented languages. *Journal of Universal Computer Science*, 5(3):208–225, 1999.
13. G. Smith. *The Object-Z Specification Language*. Kluwer Academic Publisher, 2000.
14. Rob van Glabbeek and W. P. Weijland. Refinement in branching time semantics. In *Proc. IFIP Conference*, pages 613–618. North-Holland Publishing Company, 1989.
15. H. Wehrheim. Behavioural subtyping and property preservation. In S. Smith and C. Talcott, editors, *FMOODS'00: Formal Methods for Open Object-Based Distributed Systems*. Kluwer, 2000.
16. K. Winter and G. Smith. Compositional Verification for Object-Z. In D. Bert, J.P. Bowen, S. King, and M. Walden, editors, *ZB 2003: Formal Specification and Development in Z and B*, number 2651 in LNCS, pages 280–299. Springer, 2003.