Transformations for Pattern-based Forward-Engineering

Ismaïl Khriss
Département IRO
Université de Montréal
C.P. 6128, succursale Centre-ville
Montréal, Québec H3C 3J7, Canada
+1 514 343 6111, x3495
khriss@iro.umontreal.ca

Rudolf K. Keller
Département IRO
Université de Montréal
C.P. 6128, succursale Centre-ville
Montréal, Québec H3C 3J7, Canada
+1 514 343 6782
keller@iro.umontreal.ca

ABSTRACT
Software development raise the need for traceability, i.e. the ability to control the consistency between software artifacts produced at different stages of the software life-cycle. This traceability cannot be obtained without a systematic transformational approach to software development.

In this paper, we present a new approach to the correct stepwise refinement of UML static and dynamic design models based on refinement schemas. A refinement schema is composed of two compartments. The first compartment describes the abstract model of the design and the second compartment shows its corresponding detailed model after application of one design pattern. We propose also a number of correct smaller transformations called micro-refinements. These micro-refinements can be composed to produce correct refinement schemas. Our approach supports documentation and traceability, enhances evolvability, and helps automating detailed design.

KEYWORDS
Design patterns, transformation, refinement schema, correctness proof, Unified Modeling Language (UML).

1 INTRODUCTION

Software development raise the need for traceability, i.e. the ability to control the consistency between software documentations produced at different stages of the software life-cycle. This traceability cannot be obtained without a systematic transformational approach to software development.

This research was supported by the SPOOL project organized by CSER (Consortium for Software Engineering Research) which is funded by Bell Canada, NSERC (National Sciences and Research Council of Canada), and NRC (National Research Council of Canada).

A transformational approach has two major steps. First, A formal specification is derived from user requirements. Then, a set of transformations is applied to these formal specification to get an implemented system. This approach has many advantages [1]. It relieves developers from labor-intensive, knowledge-poor tasks. It eases the documentation of design choices. Finally, it ensures correctness of the resulting software in respect to its specification. Nevertheless, existing transformational approaches have been criticized because they are difficult to use and the high cost for obtaining formal specifications of large systems.

In the past years, there have been several attempts to combine o-o modeling techniques, such as the Unified Modeling Language (UML) [17], and formal languages. In fact, the o-o methods provide well-known and easy to use graphical diagrams for modeling. Some tools support round trip engineering between these o-o techniques and formal specifications. In the IFAD VDM++ toolbox [10], for instance, UML is used to provide the structural, diagrammatic overview of a model while VDM++ is used to provide the detailed functional behavior of a model. However, these tools lack the automatic support for transitions between high-level and low-level design.

In this paper, we propose a new approach to the correct stepwise refinement of UML design models. It is based on refinement schemas that are proven to be correct. Each refinement schema uses one design pattern to transform an abstract design model (structural and behavioral views) to a more detailed design. Design Patterns [3, 8] encapsulate good practice in o-o development. They add the reuse dimension to software development since they offer a way to use repeatedly the experience of the best designers and programmers in structuring systems. For proof correctness of our refinements, we use the formalism for UML proposed by Lano et Bicarregui [13] since the “official” UML semantic description is not sufficient for our needs. We present also a number of smaller transformations, called micro-refinements. These transformations can be composed to produce refinement schemas for many design patterns.

The incorporation of our approach in CASE tools will allow for the traceability of the artifacts produced at the different stages of the design life-cycle. Furthermore, these tools can record easily design choices due to different transformations performed over software design from high-level to low-level. Finally, the automation of the design process can be partially supported.

Organization of the paper

Section 2 describes our approach. A motivating example is also introduced in this section.In Section 3, we present the semantic framework for UML and a formal definition of refinement. Then, as an example, we demonstrate how proofs of correctness of micro-refinements can be reused to prove the refinement schema for the Observer pattern. Section 4 relates some works in the areas of refinement, correctness, and automatic application of design patterns. Section 5 discusses some important aspects of this work. Finally, Section 6 provides concluding remarks and points out some future works.

2 DESCRIPTION OF APPROACH

2.1 Motivating Example

As a running example, we use the design of part of a gas station system. The purpose of the system is to control the dispensing of gasoline, to handle customer payments, and to monitor tank levels. Figure 1 shows two classes, *Pump* and *Screen*, of the class diagram (ClassD) of the system. The class *Pump* is designed to control the dispensing of petrol, and the *Screen* class to show a volume of the petrol delivered. Figure 2 describes dynamic models (statechart diagrams (StateDs) of these classes.

The development of this system requires, among other things, a way to implement the constraints between the classes *Pump* and *Screen*. The Observer pattern [8] gives a good solution for this problem. Indeed, it defines a one to many dependency between objects so that when one object changes states, all its dependants are notified and updated automatically.

2.2 Overview of approach

The objective of our approach is to provide an automatic support to transition from high-level to low-level design. This transformation is carried out by a correct stepwise refinement. Each refinement is based on application of one design pattern. In each transformation step, the designer begins by selecting an excerpt of the design model to refine with an appropriate pattern. The model is presented as a ClassD, a StateD or a CollD (UML Collaboration Diagram). Then, he or she chooses the appropriate design pattern. Finally, its corresponding refinement schema is automatically applied on the different diagrams representing the system. Consequently, we allow for traceability between design models. Moreover, a tool supporting our approach can record easily design choices due to transformations performed over software design from high-level to detailed level.

A refinement is described graphically by a schema called *refinement schema*. A refinement schema is composed of two compartments. The first compartment describes the abstract model of the design and the second compartment shows its corresponding detailed model after application of a design pattern. Figure 5 shows the refinement schema that we have defined for the Observer Pattern.

A refinement schema is composed of correct small refinements that we have called *micro-refinements*. The micro-refinement is also described by a schema. A refinement schema is specific to one design pattern but a micro-refinement is general, and therefore can be reused by many refinement schemas. It plays a role similar to the role of a function in a library. However, the difference here is that not only its code that can be reused but also its proof of correctness as we will see later.

As an example of illustration, Figures 3 and 4 describe how the high-level design of the gas station system is transformed into a detailed design by application of the Observer refinement schema. This transformation is applied to implement the constraint between the classes *Pump* and *Screen*.

We have defined refinement schemas four other design patterns, such as Proxy, Mediator, Facade and Forwarder-Receiver [12]. They have been selected from the catalogues of Gamma et al. [8] and Buschmann et al. [3]. These patterns give useful solutions to some problems of objects communication, and they show well how design patterns can be used to integrate high-level with low-level design.

2.3 Micro-refinements used by Observer pattern

The Observer refinement schema can be composed by a sequence of four micro-refinements. Firstly, the abstract model is transformed using twice the micro-refinement *inheritance* (see Figure 6). This refinement adds a new class (*Subject* and *Observer*, respectively) that becomes a super class for one class (*Class1* and *Class2*, resp.). The StateD of *Class1* (*Class2*, resp.) is extended by adding the behavior of *Subject* (and *Observer* resp.).

Secondly, the micro-refinement *adding an action to transition* is applied twice to the StateD of *Class2*. This StateD is updated by adding an action for the transition t_i and another for the transition t_k. Thirdly, the binary association <*Class1*, *Class2*> is changed by two unidirectional associations <*Subject*, *Observer*> and <*Class2*, *Class1*>.

Finally, the constraints on the two associations are changed by an automatic notification to *Class2* to update its attribute as soon as its corresponding attribute of *Class1* has changed (see Figure 7). In fact, the instruction *notify()* is added at the end of the operation *setstate1* of *Class1*, in order to notify all

observers (and therefore objects of *Class2*) to update their attributes. Then the StateD of *Class1* is modified in order to accept any time the event *getstate1*. The transition *t2* of *Class1* StateD states that when an object, at any substate *S* of *S1*, receives the event *getstate1*, *t2* is triggered and the object remains in the same substate.

3 CORRECTNESS PROOFS OF TRANSFORMATIONS

In this section, we begin by introducing briefly a semantic framework for UML and a formal definition for refinement. Then, a proof of correctness of Observer pattern is given to illustrate how proofs of micro patterns are reused in proving refinement schemas of design patterns. The proofs of correctness for other design patterns are described in [12].

3.1 Semantic framework for UML and refinement

A Formalism for UML

For a formal representation of UML models, we use a formalism proposed by Lano et Bicarregui [13] called Real-time Action Logic (RAL). This logic is a synthesis of real time logic [12] with linear temporal logic [16] and the object calculus formalism of [6].

A UML class C is represented as a theory Γ_C. A theory is described by its name, types of its symbols, a set of attributes representing instance or class variables, a set of actions which may affect the data (such as operations, StateD transitions and methods) and a set of axioms which are logical properties and constraints between the theory symbols. Theories can be linked by theory morphisms, which allow to describe an UML model (a ClassD for example) from assembled theories of elements as classes or associations.

Theories use additional standard mathematical notation such as:

1. for each class or state X there is an attribute ext(X): F(X) denoting the set of existing instances of X (F denotes the "set of finite sets of").
2. if α is an action symbol and P a predicate, then $[\alpha]$ P is a predicate which means that P is a post-condition of α.
3. $\alpha \supset \beta$ (α calls β) is defined such that "every execution of α coincides with an execution of β.
4. temporal operators such as $\Diamond$ (sometime in the future), $\Box$ (always in the future) and $\bigcirc$ (next).

If D inherits from C then Γ_D is constructed by including Γ_C, adding symbols and axioms of new features of D, and adjoining the axioms $D \subseteq C \wedge ext(D) \subseteq ext(C)$.

A StateD for a Class is formalized by extending the theory of C as follow:

1. each state S is represented in the same manner as a subclass of C.
2. each transition in the StateD and each event for which the StateD defines a response yields a distinct action symbol. Also the occurrence of an event e is equivalent to the occurrence of one of its transitions: $t_1 \supset e \wedge \ldots \wedge t_n \supset e$
3. the axiom for the effect of a transition t with event e and guard condition G from state S1 to state S2 is: $\forall$ c: C. c $\in$ ext(S1) $\wedge$ a.G $\Rightarrow$[c!t] (c $\in$ ext(S2))
4. the transition only occurs if the trigger event occurs whilst the object is in the correct state: $\forall$ c: C. c $\in$ ext(S1) $\wedge$ a.G $\Rightarrow$ (c!e $\supset$ c!t)
5. the generated actions must occur at some future time: a!t $\Rightarrow$ $\bigcirc \Diamond$ c.Action

Formal definition of Refinement

A system CR refines a system C if there is a morphism σ from a theory Γ_C of C to a theory Γ_{CR} of CR such that every axiom φ of Γ_C under σ is provable in Γ_{CR}.

3.2 Correctness proofs for Observer refinement schema

As we have seen in section 2.3, the Observer refinement schema is composed of four small refinements. It is obvious that the third refinement (see section 2.3) is correct since *Class1* inherits from *Subject* the unidirectional association with *Observer* (and therefore with *Class2*). The correctness proofs of the first and the second refinements can be found in [12]. Below, we present, for illustration, the proof of correctness of the fourth refinement.

For the fourth refinement (see Figure 7) we have the axiom:

c: Class2. c $\in$ ext(Class2) $\Rightarrow$ c.state2 = c.subject.state1

In order to prove that this axiom holds in the refined model and on condition that *state1* is only modified by *setstate1*, we have to prove that:

$\forall$ c: Class2. c $\in$ ext(Class2) $\wedge$ c.subject!setstate1(s)$\Rightarrow$ $\bigcirc \Diamond$ c.state2 = s

$\forall$ c: Class2. c $\in$ ext(Class2) $\wedge$ c.subject!setstate1(s) $\Rightarrow$ $\bigcirc \Diamond$ (for all o in c.subject.observers do o!update())

$\Rightarrow$ $\bigcirc \Diamond$ (c!update()) since c $\in$ observers (note that the StateD of Class2 is designed in a such a way that at creation of an object o of Class2, o becomes an element of o.subject.observers)

$\Rightarrow$ $\bigcirc \Diamond$ (c.state2 = c.subject!getstate1())

$\Rightarrow$ $\bigcirc \Diamond$ (c.state2 = c.subject!getstate1()) (note also that the StateD of Class1 is designed in a such a way that an object of this class can always handle the event *getstate1* at reception)

$\Rightarrow$ $\bigcirc \Diamond$ (c.state2 = s)

4 RELATED WORK

In this section we review related work in the areas of refinement, correctness, and automatic application of design patterns.

The work of Lano et al. is highly related to ours. In [14], they use design patterns for reverse engineering. Reverse engineering is achieved by formally proved refinement transformations of VDM++ specifications using a version of Object Calculus [6]. They provide also a number of smaller transformations. Two of these smaller transformations, *abstraction* and *indirection*, are also identified in our work. The limitations of this work in respect to ours is that they do not consider dynamic aspects of design. Moreover, abstract specifications are defined in a way that are not general enough, which limits the applicability of the approach.

In [13], Lano et al. present a mathematical semantic representation of UML called Real-time Action Logic (RAL), the formalism we use in our present work. They provide also a set of refinements on class diagrams and statechart diagrams. Examples of refinements are: how to eliminate optional associations in ClassDs or how to strengthen transition guards in StateDs. These transformations are very useful since they are general and applicable in any UML design models; but they have to be composed to produce high-level refinements, which are more interesting for designers.

In traditional procedural approaches, there are many works that tackle the problem of the preservation of functional correctness or architecture correctness such as [9, 15]. Some of these works recognize also the importance of the notion of schematic transformation in stepwise refinement. For example, Moriconi et al. [15] provide schema transformations that keep architecture correctness. Different architecture styles are investigated in this work such as dataflow, pipe-filter and shared-memory styles. Architectures are described schematically and textually. They use first-order theories for correctness proofs. Moriconi et al. define also a syntactic form of correct composition for individual refinements. However, these works cannot be easily applied to o-o software development.

In o-o approaches, Shlaer et Mellor [18] define a method for design by transformations. The system to be built is first partitioned into domains, each of which will be analysed separately from the others. A domain is an abstract world inhabited by a distinct set of objects that behave according to rules and policies characteristic of the domain. Examples of domains are an application domain, a user interface domain or a software architecture domain. Domains are hierarchically classified from abstract to detailed level. Then a transformation engine is constructed to translate elements of one domain into corresponding elements of the next lower domain, according to rules defined by the system designer. We can say that Shlaer et Mellor give only a framework for transformations but do not give examples of transformation engines that can be reused in the design of systems.

There are some works that are interested in automating the application of design patterns at the implementation level such as [2, 4, 5, 7]. For example, Budinsky et al. [2] provide a tool that supports the application of design patterns by generating code according to specifications described in an ad-hoc scripting language called COGENT. Florijn et al. [7] present a prototype tool that performs reverse engineering by attaching roles to classes and relations. Eden et al. [4] use metaprogramming techniques to automate the application of design patterns. They defines also in [5] a catalogue of micro-patterns that can be composed for the application of design patterns at the implementation level.

5 DISCUSSION OF APPROACH

Below we will discuss our approach in respect to some interesting aspects: decomposition/composition of refinements, problems encountered, and relevance of approach in development process.

Decomposition/Composition

We have seen that refinement schemas of design patterns can be decomposed into correct small refinements. An important facet of this approach is that we allow not only the reuse of code refinements, but also guarantee their correctness. These micro-refinements can be composed to produce other correct refinement schemas.

Moreover, these micro-refinements can also be used alone since they provide solutions to some development tasks. For example, the micro-refinement *inheritance* synthesizes a new StateD of a class that has a new super-class. Recall, that the new StateD becomes a composite state, which contains two concurrent substates: the old StateD and the StateD of the super-class. Note that this solution is not general (to our knowledge, it is impossible to get a general solution) but it is sufficient to many problems, as we have demonstrated with the Observer refinement schema.

Problems encountered

In this paper, we have seen how a formal semantic for UML can help to prove transformations on UML models. Nevertheless, we have also encountered some problems with this semantic framework. The first problem is that the authors of the framework do not give a formal semantic for all UML models, in particular not for some interesting features of CollDs. The second problem is that refinement schemas often work on partial specifications of model elements. Yet, a formal prove needs complete specifications. As an example, in the Mediator refinement schema, we work on partial CollDs of object operations [12]. The solution that we have taken is to work on examples. Consequently, a deeper look is needed to overcome these limitations.

Relevance of approach in development process

Current object-oriented CASE tools support various graphical notations for modeling a system from different views, but lack the possibility of automatic transformations and traceability between models. In fact, coherence support between software artifacts of the software life-cycle is not ensured. The incorporation of our work into such CASE tools will allow this traceability. Moreover, tools can record easily design choices due to transformations performed over software design from high-level to detailed level.

Our approach, before incorporation into a CASE tool, needs to be populated with a large catalogue of patterns. We are aware that a set of micro-refinements cannot be complete without a deeper investigation of many other patterns. Thus, we plan to study other patterns in order to extract other micro-refinements and enrich our catalogue of supported design patterns.

6 CONCLUSION AND FUTURE WORK

We have described a new approach to the stepwise refinement for integrating UML high-level and low-level design with refinement schemas based on design patterns. These refinement schemas are proved to be correct. Our pattern-based transformational approach supports documentation and traceability, enhances evolvability, and ultimately leads to better o-o designs. The main contribution of our work can be summarized in two points. The first point is that we have demonstrated how refinement schemas can be composed of correct micro refinements. Not only, the code of these smaller transformations can be reused but also the proofs of their correctness. The second point is that different views of design (ClassDs, StateDs and CollDs) have been taken into account by our approach, and that their coherence is being ensured.

As future work, we want to study other design patterns in order to extend our library of micro-refinements. Also, we plan to investigate ways to prove refinements that perform partial specifications of a model element, such as a CollD of an operation.

REFERENCES

[1] R. Balzer, T. Cheatham Jr., and C. Green. Software technology in the 1990s: Using a new paradigm. IEEE Computer:(16)11:39-45, November 1983.

[2] F.J.M Budinsky, M.A. Finnie, J.M. Vlissides, and P.S. Yu. Automatic code generation from design patterns. Object Technology:35(2):172-191, 1996.

[3] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. Pattern-Oriented Software Architecture - A System of Patterns. John Wiley and Sons, 1996.

[4] A.H. Eden, J. Jil, and A. Yehuday. Precise Specification and Automatic Application of Design Patterns. In Proc. of the IEEE Int. Automated Software Engineering Conference, 1997.

[5] A.H. Eden, and A. Yehuday. Tricks Generate Patterns. Technical report 324/97, The Department of Computer Science, Schriber School of Mathematics, Tel Aviv University, 1997.

[6] J. Fiadeiro and T. Maibaum. Sometimes "Tomorrow" is "Sometime". In Temporal Logic, volume 827 of Lecture Notes in Artificial Intelligence:48-66, Springer-Verlag, 1991.

[7] G. Florijn, M. Meijers, and P. van. Winsen. Tool Support in Design Patterns. In European Conference On Object-Oriented Programming (ECOOP'97), volume 1241 of Lecture Notes in Computer Science:472-495, Springer-Verlag, 1997.

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, J. Design Patterns. Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.

[9] S.L. Gerhart. Knowledge about programs. In Proc. Int. Conf. Software Reliability:88-95, Los Angeles, CA, April 1975.

[10] IFAD. User Manual for the IFAD VDM++ Toolbox. IFAD-VDM-50, the VDM Tool Group, IFAD, August 1998.

[11] F. Jahanian and A.K. Monk. Safety Analysis of Timing Properties in Real-Time Systems. IEEE Transactions of Software Engineering:(12):890-904, September 1986.

[12] Ismaïl Khriss and Rudolf K. Keller. Integration between High-level and Low level Design with Refinement Schemas based on Design Patterns. Technical Report GELO-91, Université de Montréal, Montréal, Québec, Canada, January 1999.

[13] K. Lano and J. Bicarregui. UML Refinement and Abstract Transformations. ROOM 2 Workshop, Bradford University, May 1998.

[14] K. Lano, J. Bicarregui, and S. Goldsack. Formalising Design Patterns. RBCS-FACS Northern Formal Methods Workshop, 1996.

[15] M. Moriconi, X. Qian, and R.A. Riemenschneider. Correct architecture refinement. IEEE Transactions on Software Engineering: 21(4):356-372, April 1995.

[16] J.S. Ostroff . Temporal Logic for Real-Time Systems. John Wiley, 1989.

[17] J. Rumbaugh, I. Jacobson, and G. Booch . The Unified Modelling Language Reference Manual. Addison Wesley, Inc., 1999.

[18] S. Shlaer, and S.J. Mellor. A deeper look at the transition form analysis to design. Journal of Object-Oriented Programming, 5(9):16–21, 1993.

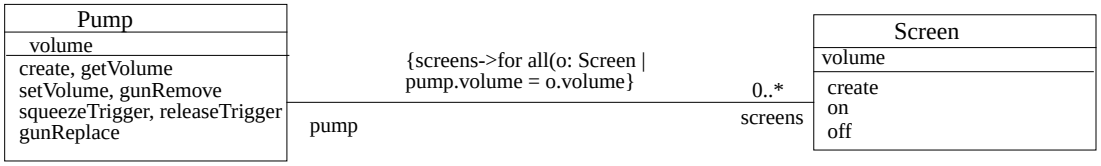

Figure 1: Part of ClassD of gas station system

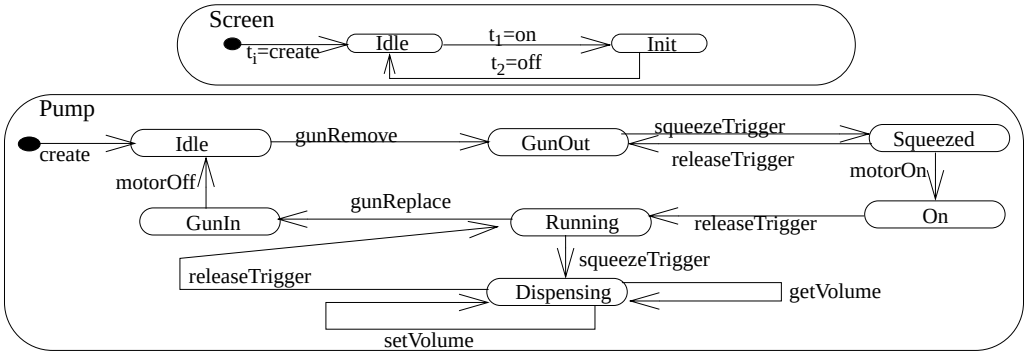

Figure 2: StateDs of Screen and Pump classes

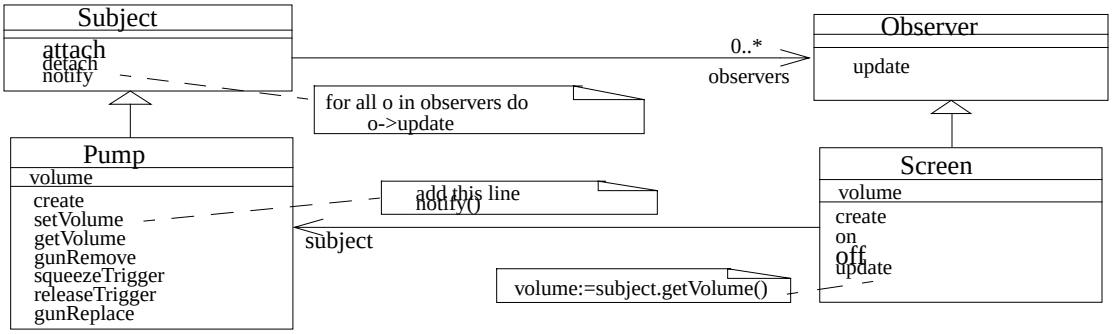

Figure 3: ClassD after application of Observer refinement schema

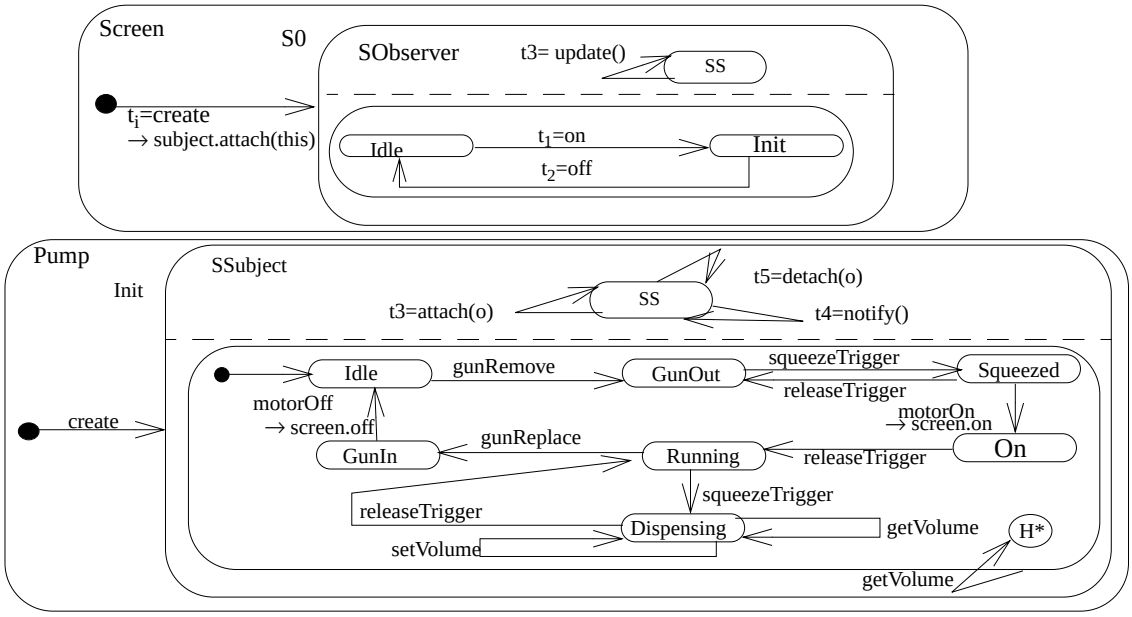

Figure 4: StateDs of Screen and Pump classes after application of Observer refinement schema

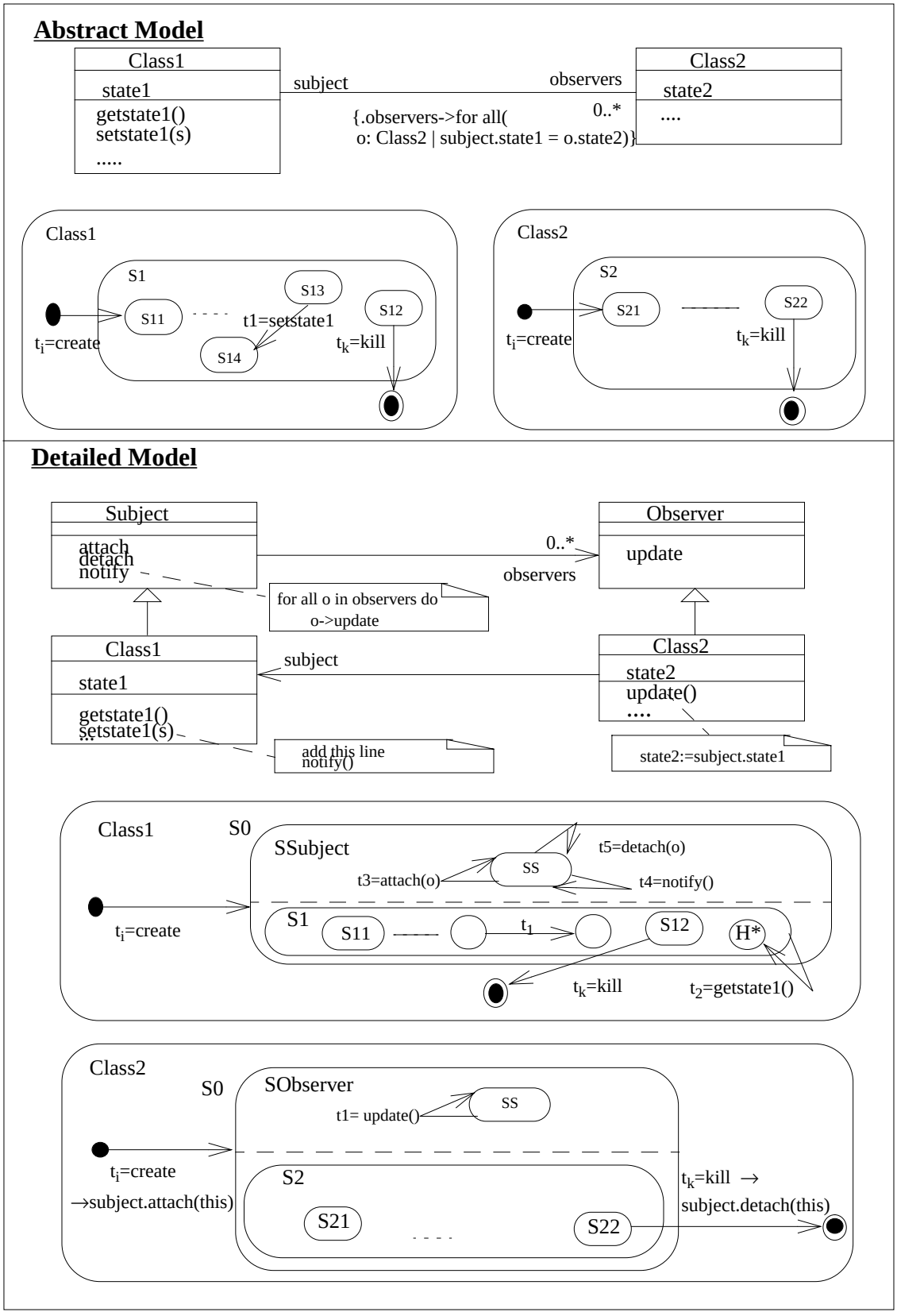

Figure 5: Refinement schema of the Observer pattern

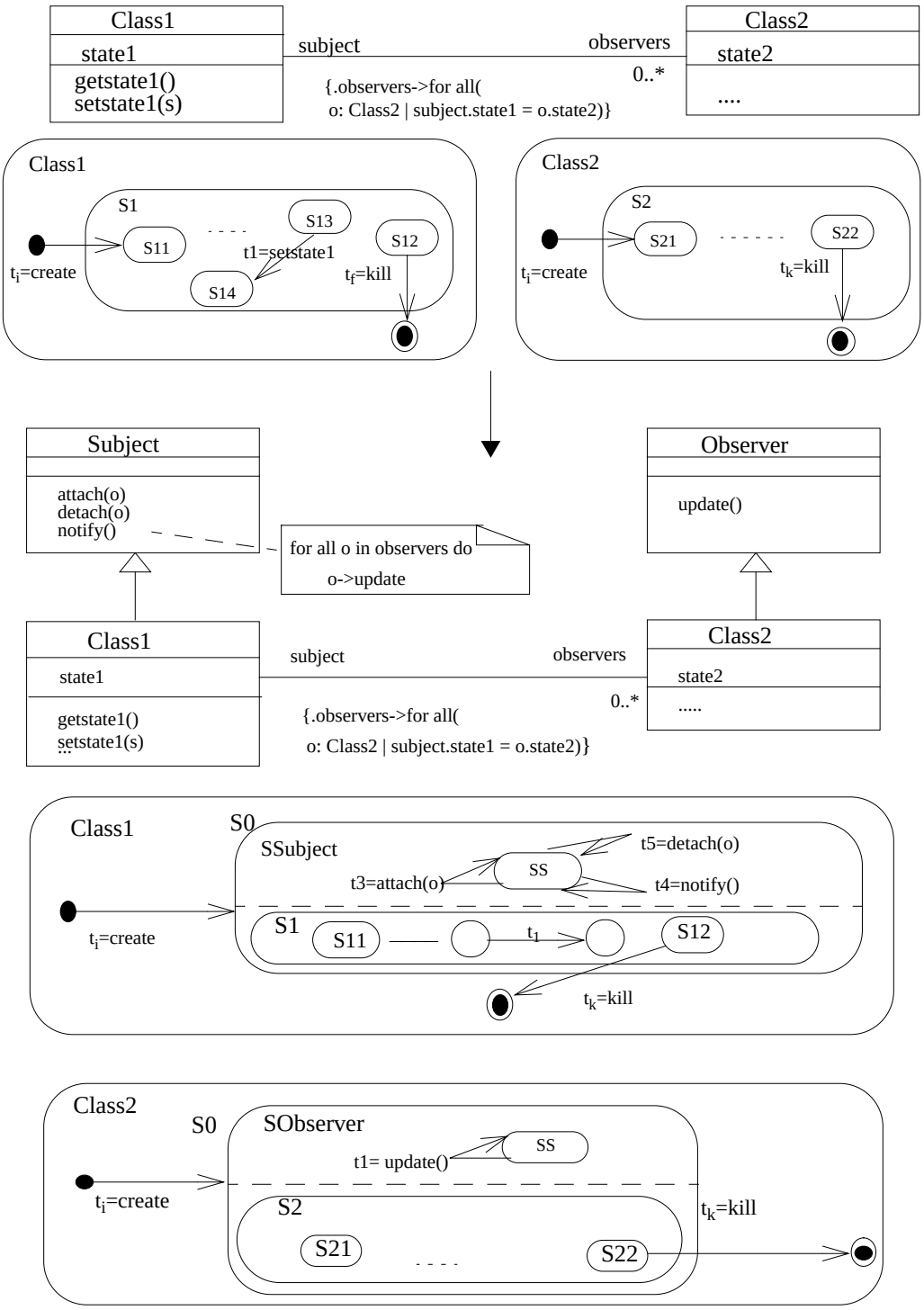

Figure 6: First micro-refinment used by Observer: “inheritance”

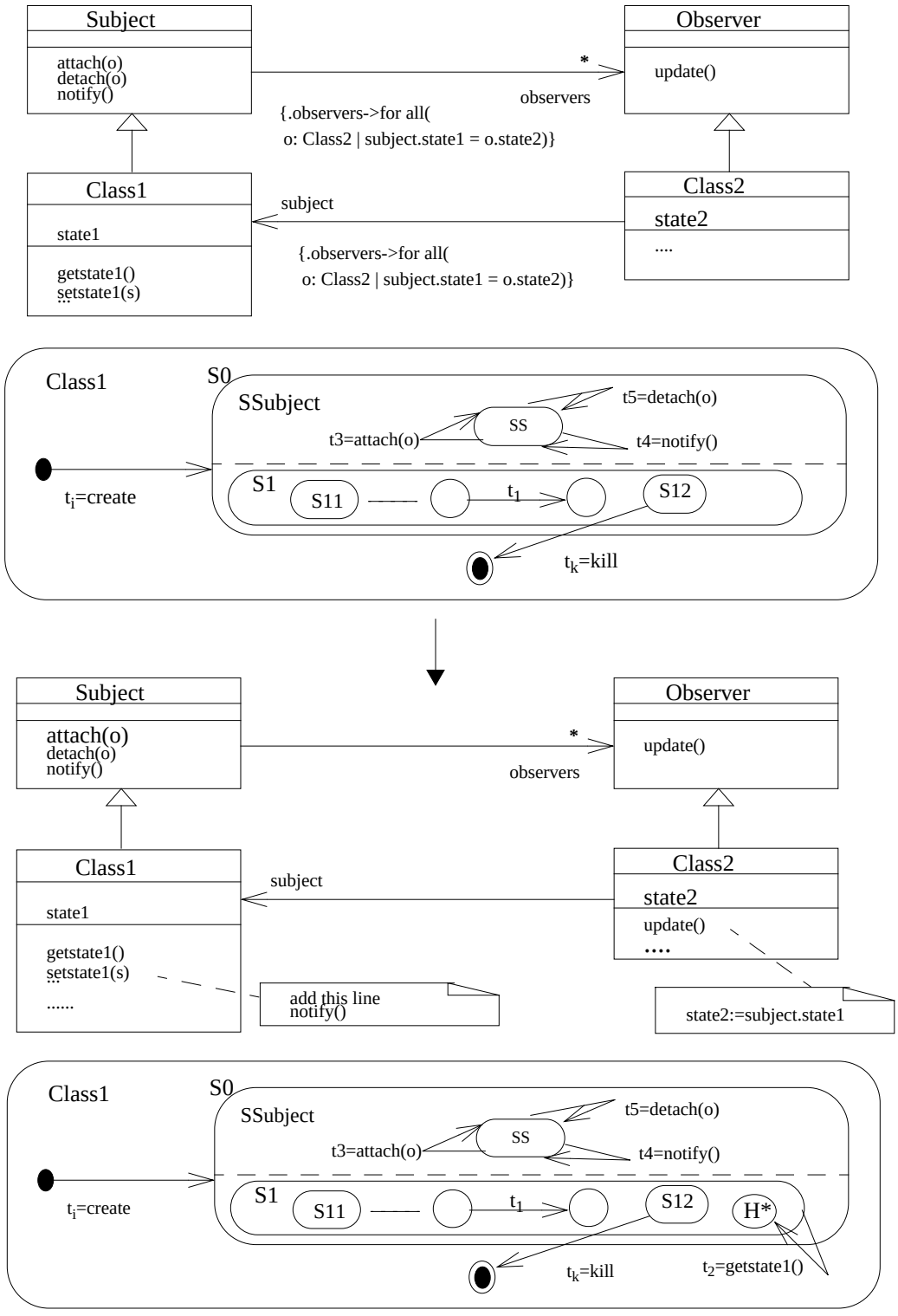

Figure 7: Fourth micro-refinement used by Observer: “Automated Notification”