Specifying and Verifying UML Activity Diagrams Via Graph Transformation*

Paolo Baldan[1], Andrea Corradini[2], and Fabio Gadducci[2]

[1] Dipartimento di Informatica, Università Ca' Foscari di Venezia, Italy
[2] Dipartimento di Informatica, Università di Pisa, Italy

Abstract. We propose a methodology for system specification and verification based on UML diagrams and interpreted in terms of graphs and graph transformations. Once a system is modeled in this framework, a temporal graph logic can be used to express some of its relevant behavioral properties. Then, under certain constraints, such properties can be checked automatically. The approach is illustrated over a simple case study, the so-called Airport Case Study, which has been widely used along the first two years of the AGILE GC project.

1 Introduction

The use of visual modeling techniques, like the UML [22], for the design and development of large applications is nowadays well established. In these approaches a system specification consists of several related diagrams, that represent both the statics and the dynamics of the system. Since the development process is made easier if it is possible to reason about the system under development at an early stage, in this paper we sketch a methodology which allows to express interesting behavioral properties of the system in a suitable logic, and, under certain constraints, to verify them automatically. We present our approach by applying it to a simple case study, which has been widely used along the first two years of the AGILE GC project [1], namely the Airport Case Study [2].

The first step of our methodology consists of representing a UML specification as a Graph Transformation System (GTS). Since the various kinds of diagrams used in a UML specification essentially are graphs annotated in various ways, it comes as no surprise that many contributions in the literature use techniques based on the theory of graph transformation to provide an operational semantics for UML behavioral diagrams (see, among others, [10–13, 17, 18]). We will stick to a tiny fragment of the UML, including (suitably restricted) class and instance diagrams for the statics, and activity diagrams for the dynamics of a system specification. The class diagram determines the shape of the graphs that will be used for modeling instance diagrams, called *instance graphs*, while each activity in a behavioral diagram will be represented as a graph transformation rule, describing the effect of such activity on the instance graph.

* Research partially supported by the EU FET – GC Project IST-2001-32747 AGILE and the EC RTN 2-2001-00346 SegraVis.

Next, following an approach to the verification of graph transformation systems developed during the last few years (see [3–5]), we shall introduce a temporal logic which allows for formulating relevant properties of a GTS. The logic $\mu\mathcal{L}2$ proposed in [5] (that we slightly modify in order to deal with a more general class of graphs) is a propositional μ-calculus where the basic predicates are monadic second-order formulae interpreted over graphs. For (fragments of) this logic, verification techniques have been proposed for finite and infinite-state systems, which exploit finite approximations of the unfolding of the GTS [4, 5].

The expressiveness of the proposed logic is tested against a collection of properties concerned with the Airport Case Study, which were collected by the members of the AGILE project in a meeting dedicated to modal and temporal logics (the proceedings are available at [1]). Even if monadic second-order logic is very expressive as far as graph properties are concerned, it turns out that some interesting dynamic properties of the Airport Case Study cannot be encoded directly in the proposed logic. Being the logic propositional at the temporal layer, there is no way to write formulae predicating about the properties of a specific object at different times. We briefly outline a more expressive graph logic, which, extending $\mu\mathcal{L}2$ with non-propositional features, allows for overcoming the mentioned limitations. Unfortunately, the verification techniques in [4, 5] do not directly apply to this logic, but we are confident that they can be suitably generalized, at least in the finite-state case.

In the next section, after a brief introduction to the Airport Case Study and to a partial specification of it using UML, we first show how the states of the case study can be represented by graphs, and the corresponding activities by graph transformation rules. Next we introduce the temporal logic for GTSs, and finally we discuss to what extent some relevant properties can be expressed in that logic, and which extensions of such logic would be needed.

2 The Airport Case Study

As anticipated above, in this paper we shall illustrate the main concepts using as running example a fragment of the Airport Case Study, described in [2].

The case study consists of a system representing planes landing and taking off from airports. The planes transport passengers. Departing passengers check in and board the plane; their luggage is loaded in the plane. The plane is ready to take off after all passengers have boarded the plane and their luggage is loaded. After the plane has reached its destination airport, passengers get off the plane and claim their luggage. On board, passengers may perform some activities, such as consuming a meal. The specification and modeling of several aspects of this case study using a variety of formalisms (UML, CommUnity, KLAIM and Graph Transformations) is presented in [2].

We shall consider here only the part of the case study related to the *Departure Use Case*, including the check-in and boarding of passengers, the loading of their luggage and the take-off of the plane. A UML instance diagram representing the initial state of the system is shown in Figure 1 (a), adopting the stereotypes *mobile* and *location* proposed in a recent extension for mobility of the language developed inside the AGILE project; instead, Figure 2 shows an activity diagram describing the relationships among the relevant activities. In the next subsections we discuss how to model this system using graph transformation, representing its states as graphs and the activities as rules.

2.1 Representing States as Hypergraphs

In order to model a UML specification as a graph transformation system, an obvious pre-requisite is the formal definition of the structure of the graphs which represent the states of the system, namely the *instance graphs*. However, there is no common agreement about this: we shall present a novel formalization, which shares some features with the one proposed in [14].

An *instance graph* includes a set of *nodes*, which represent all data belonging to the state of an execution. Some of them represent the elements of primitive data types, while others denote instances of classes. Every node may have at most one outgoing *hyperedge*, i.e., an edge connecting it to zero or more nodes.[1] Conceptually, the node can be interpreted as the "identity" of a data element, while the associated hyperedge, if there is one, contains the relevant information about its state. A node without outgoing hyperedges is a *variable*: variables only appear in transformation rules, never in actual states.

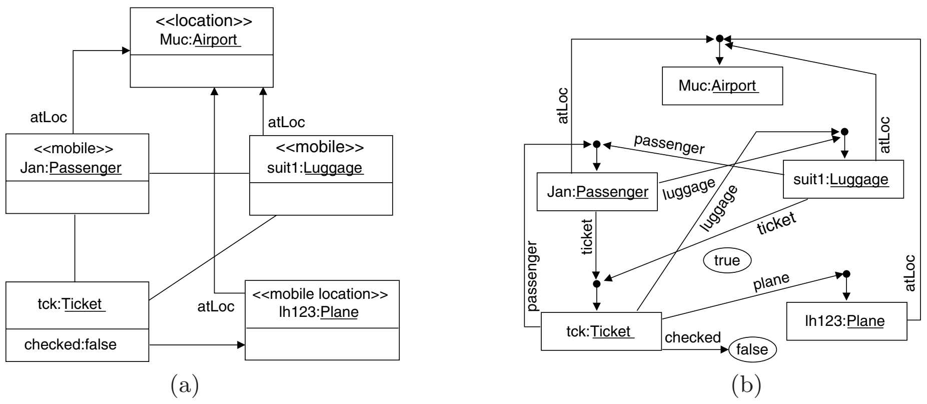

Fig. 1. An instance diagram (a) and the corresponding instance graph (b)

Typically, an instance of a class C is represented by a node n and by a hyperedge labeled with the pair $\langle \mathit{instanceName} : C \rangle$. This hyperedge has node n as its only *source*, and for each attribute of the class C it has a link (a *target tentacle*) labeled by the name of the attribute and pointing to the node representing the attribute value. For the logic presented later in Section 3, we assume that the *source tentacle* (linking a hyperedge to its source node) is implicitly labeled by

[1] Formally, these graphs are *term graphs* [21].

self. Every instance graph also includes, as unary hyperedges (i.e., hyperedges having only the self tentacle), all constant elements of primitive data types, like integers (0, 1, -1, ...) and booleans (true and false), as well as one edge null:C for each relevant class C.

Figure 1 (a) shows an instance diagram which represents the initial state of the Airport Scenario. As usual, the attributes of an instance may be represented as directed edges labeled by the attribute name, and pointing to the attribute value. The edge is unlabeled if the attribute name coincides with the class of the value (e.g., lh123 is the value of the plane attribute of tck). An undirected edge represents two directed edges between its extremes. The diagram conforms to a class diagram that is not depicted here.

Figure 1 (b) shows the instance graph (according to the above definitions) encoding the instance diagram. Notice that the graph contains two elements of a basic data type, true and false: these are depicted as ovals, which stands actually for a node attached through the self tentacle to a unary hyperedge. Up to a certain extent (disregarding OCL formulas and cardinality constraints), a class diagram can be encoded in a corresponding *class graph* as well; then the existence of a graph morphism (i.e., a structure preserving mapping) from the instance graph to the class graph formalizes the relation of conformance.

In the following we shall depict the states of the system as instance diagrams, which are easier to draw and to understand, but they are intended to represent the corresponding instance graphs.

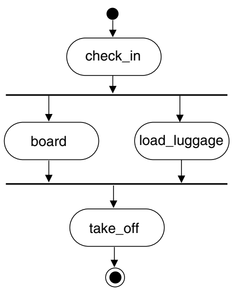

Fig. 2. The Activity Diagram of the Use Case Departure

2.2 Representing Activities as Graph Transformation Rules

Figure 2 shows the activity diagram of the Use Case Departure of the Airport Case Study. This behavioural diagram ignores the structure of the states and the information about which instances are involved in each activity, but stresses the causal dependencies among activities and the possible parallelism among them. More precisely, from the diagram one infers the requirement that board and load_luggage can happen in any order, after check_in and before take_off.

By making explicit the roles of the various instances in the activities, we shall implement each activity as a graph transformation rule. Such rules describe local modifications of the instance graphs resulting from the corresponding activities.

We will show that they provide a correct implementation of the activity diagram, in the sense that the causality and independence relations between the rules are exactly those prescribed in the activity diagram.

Let us first consider the activity board. Conceptually, in the simplified model we are considering, its effect is just to change the location of the passenger (i.e., its atLoc attribute) from the airport to the plane. In the rule which implements the activity, we make explicit the preconditions for its application: 1) the passenger must have a ticket for the flight using that plane; 2) the value of the checked attribute of the ticket must be true; 3) the plane and the passenger must be at the *same* location, which is an airport.

All the above requirements are represented in the graph transformation rule implementing the activity board, shown in Figure 3. Formally, this is a *double-pushout graph transformation rule* [7], having the form $L \overset{l}{\leftarrow} K \overset{r}{\rightarrow} R$, where L, K and R are instance graphs, and l and r are graph morphisms. In this case l and r are actually inclusions, represented implicitly by the position of nodes and edges in the source and target graphs.

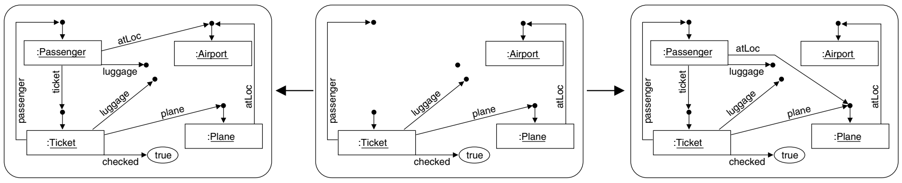

Fig. 3. The graph transformation rule for boarding

Intuitively, a rule states that whenever we find an occurrence of the *left-hand side* L in a graph G we may replace it with the *right-hand side* R. The *interface* graph K and the two morphisms l and r provide the *embedding information*, that is, they specify where R should be glued with the *context* graph obtained from G by removing L. More precisely, an *occurrence* of L in G is a graph morphism $g : L \rightarrow G$. The context graph D is obtained by deleting from G all the nodes and edges in $g(L - l(K))$ (thus all the items in the interface K are preserved by the transformation). The insertion of R in D is obtained by taking their disjoint union, and then by identifying for each node or edge x in K its images $g(x)$ in G and $r(x)$ in R: formally, this operation is a *pushout* in a suitable category.

Comparing the three graphs in the rule, one can see that, in order to change the value of the attribute atLoc of the Passenger, the whole hyperedge is deleted and created again: one cannot delete a single attribute, as the resulting structure would not be a legal hypergraph.[2] Instead, the node representing the identity of the passenger is preserved by the rule. Also, all the other items present in

[2] This is a design choice which forbids the simultaneous application of another rule accessing the Passenger. Conceptually, this is equivalent to putting a "lock" on the object whose attribute is changed.

the left-hand side (needed to enforce the preconditions for the application of the rule) are not changed by the rule.

In most cases, it is possible to use a much more concise representation of a rule of this kind, by depicting it as a single graph (the union of L and R), and annotating which items are removed and which are created by the rule. Figure 4 (a) shows an alternative but equivalent graphical representation of the rule of Figure 3 as a degenerate kind of *collaboration diagram* (without sequence numbers, guard conditions, etc.) according to [6].

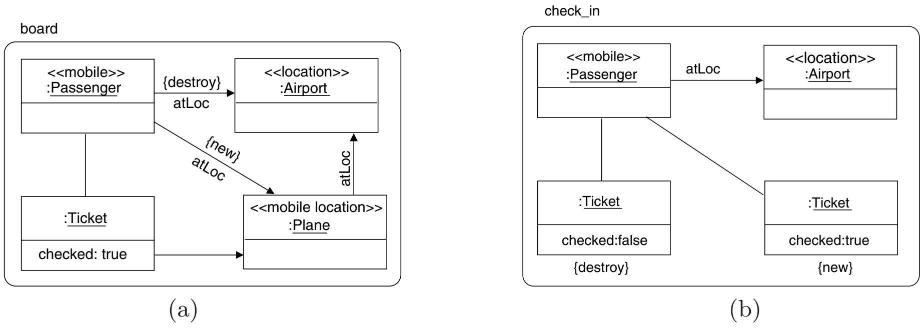

Fig. 4. The rules for boarding (a) and for checking in (b) as collaboration diagrams

Here the state of the system is represented as an instance diagram, and the items which are deleted by the rule (resp. created) are marked by {destroy} (resp. {new}: beware that these constraints refer to the whole Passenger instance, and not only to the atLoc tentacle). For graph transformation rules with injective right-hand side (and no shared variable, like all those considered here), this representation is equivalent to the one with explicit left-hand side, interface and right-hand side graph, and for the sake of simplicity we will stick to it.

Figure 4 (b) and Figures 5 (a, b) show the rules implementing the remaining three activities of Figure 2, namely check_in, load_luggage and take_off: the corresponding graphical representation can be recovered easily. Notice that the effect of the take_off rule is to change the value of the atLoc attribute of the plane: we set it to null, indicating that the location is not meaningful after taking off; as a different choice we could have used a generic location like Air or Universe.

The next statement, by exploiting definitions and results from the theory of graph transformation, describes the causal relationships among the potential rule applications to the instance graph of Figure 1 (b) (as depicted in Figure 6), showing that the dependencies among activities stated in the diagram of Figure 2 are correctly realized by the proposed implementation.

Proposition 1 (Causal Dependencies Among Rules Implementing Activities). *Given the start instance graph G_0 of Figure 1 (b) and the four graph transformation rules of Figures 4 and 5,*

– *the only rule applicable to G_0 is* check_in*, producing, say, the instance graph G_1;*

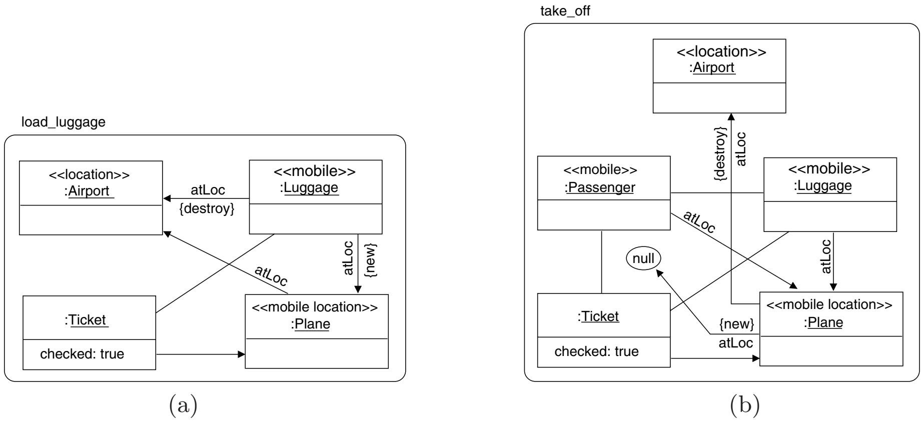

Fig. 5. The rules for loading the luggage (a) and for taking off (b)

$$\begin{tikzcd}
 & & G_1' \arrow[dr, "\text{board}"] & & \\
G_0 \arrow[r, "\text{check\_in}"'] & G_1 \arrow[ur, "\text{load\_luggage}"] \arrow[dr, "\text{board}"'] & & G_2 \arrow[r, "\text{take\_off}"'] & G_3 \\
 & & G_1'' \arrow[ur, "\text{load\_luggage}"'] & &
\end{tikzcd}$$

Fig. 6. Dependencies among the graph transformation rules of the Departure Use Case

– *both* board *and* load_luggage *can be applied to graph* G_1*, in any order or even in parallel, resulting in all cases in the same graph (up to isomorphism), say* G_2*;*
– *rule* take_off *can be applied to* G_2*, but not to any other instance graph generated by the above mentioned rules.*

2.3 Enriching the Model with Synchronized Hypergraph Rewriting

Quite obviously, the rule take_off fits in the unrealistic assumption that the flight has only one passenger. Let us discuss how this assumption can be dropped by modeling the fact that the plane takes off only when *all* its passengers and *all* their luggages are boarded.

We shall exploit the expressive power of Synchronized Hypergraph Rewriting [15], an extension of hypergraph rewriting, to model this situation in a very concise way. Intuitively, the plane has as attribute the collection of *all* the tickets for its flight, and when taking off it broadcasts a synchronization request to all the tickets in the collection. Each ticket can synchronize only if its passenger and its luggage are on the plane. If the synchronization fails, the take_off rule cannot be applied. This activity can be considered as an abstraction of the check performed by the hostess/steward before closing the gate.

Conceptually, a graph transformation rule *with synchronization* is a rule where one or more nodes of the left-hand side may be annotated with an *action*.

If the node is a variable, the action is interpreted as a synchronization request issued to the instance which will be bound to the variable when applying the rule. If the annotated node is the source of an instance, the action is interpreted as an acknowledgment issued by that instance. Given an instance graph, a bunch of such rules with synchronization can be applied simultaneously to it only if, besides satisfying the usual conditions for parallel application, all the synchronization requests are properly matched by a corresponding acknowledgment.

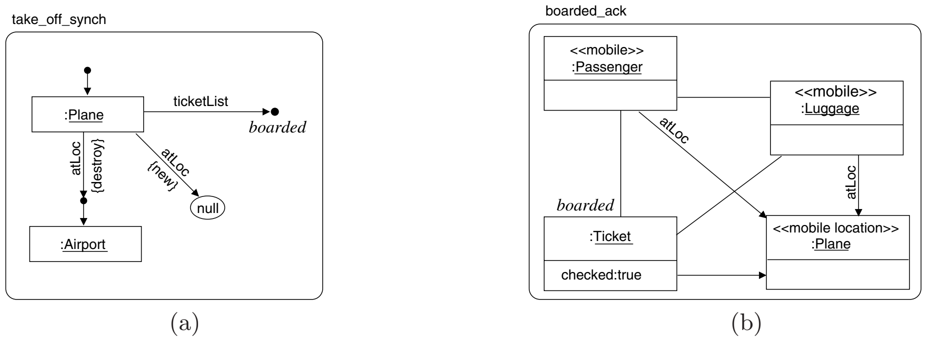

Fig. 7. The rules for taking off while checking that all passengers are on board (a), and for acknowledging the synchronization (b)

To use this mechanism in our case study, let us first assume that at the class diagram level we inserted an association Plane $\stackrel{1\quad *}{\Longleftrightarrow}$ Ticket with the obvious meaning: we call TicketList the corresponding attribute of a plane. Figure 7 (a) shows rule take_off_synch: the plane takes off, changing its location from the airport to null, only if its request for a synchronization with a *boarded* action is acknowledged by its collection of tickets. In this rule we depict the state as an instance graph, showing explicitly that a node representing the value of the attribute ticketList of the plane is annotated by the *boarded* action. On the other side, according to rule boarded_ack of Figure 7 (b), a ticket can acknowledge a *boarded* action only if its passenger and its luggage are both located on its plane. Here the state is depicted again as an instance *diagram*, and the *boarded* action is manifested on the node representing the identity of the ticket.

To complete the description of the system, we must explain how the tickets for the flight of concern are linked to the ticketList attribute of the plane. In order to obtain the desired synchronization between the plane and all its tickets, we need to assume that there is a subgraph which has, say, one “input node”(the ticketList attribute of the plane) and n “output nodes” (the tickets); furthermore, this subgraph should be able to “match” synchronization requests on its input to corresponding synchronization acknowledgments on its ouputs.

More concretely, this is easily obtained, for example, by assuming that the collection of tickets is a linked list, and by providing rules for propagating the synchronization along the list: this is shown in Figure 8, where the rules should be intended to be parametric with respect to the action *act*.

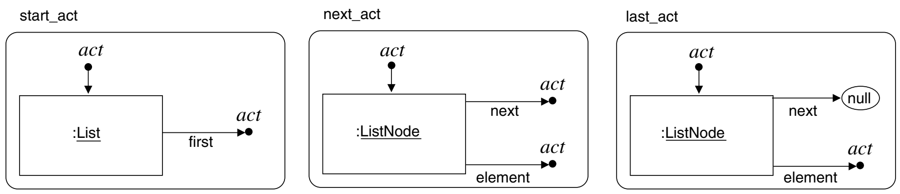

Fig. 8. The rules for broadcasting synchronizations along a linked list

3 A Logic for Graph Transformation Systems

This section presents a slight variation of the behavioral logic for graph transformation systems proposed in [5], adapted to deal with hypergraphs. It is essentially a variant of the propositional μ-calculus (i.e., a temporal logic enriched with fixed-point operators) where propositional symbols range over arbitrary *state predicates*, characterizing static graph properties, which are expressed in a monadic second-order logic.

3.1 A Monadic Second-Order Logic for Graphs

We first introduce the monadic second-order logic $\mathcal{L}2$ for specifying graph properties, i.e.,"static" properties of system states. Quantification is allowed over edges, but not over nodes (as, e.g., in [8]).

Definition 1 (Graph Formulae). *Let $\mathcal{X}_1 = \{x, y, w, \ldots\}$ be a set of (first-order) edge variables and $\mathcal{X}_2 = \{X, Y, W, \ldots\}$ be a set of (second-order) variables ranging over edge sets. The set of* graph formulae *of logic $\mathcal{L}2$ is defined as*

$$\begin{array}{rll} F & ::= & x = y \mid x.attrx = y.attry \mid \\ & & type(x) = \ell \mid x \in X \mid \qquad\qquad\qquad (Predicates) \\ & & F \vee F \mid F \wedge F \mid F \Rightarrow F \mid \neg F \mid (Connectives) \\ & & \forall x.F \mid \exists x.F \mid \forall X.F \mid \exists X.F \qquad (Quantifiers) \end{array}$$

where ℓ belongs to a set Λ of labels, *and attrx, attry to a fixed set of* attribute names*. We denote by free(F) and Free(F) the sets of first-order and second-order variables, respectively, occurring free in F.*

Let G be an instance graph, let F be a graph formula in $\mathcal{L}2$, and let σ : $free(F) \to Edges(G)$ and $\Sigma : Free(F) \to \mathcal{P}(Edges(G))$ be valuations for the free first- and second-order variables of F, respectively. The *satisfaction relation* $G \models_{\sigma,\Sigma} F$ is defined inductively, in the usual way; for instance

$$\begin{array}{rcl}
G \models_{\sigma,\Sigma} x = y & \iff & \sigma(x) \text{ and } \sigma(y) \text{ are the same edge;}\\
G \models_{\sigma,\Sigma} x.attrx = y.attry & \iff & \text{edges } \sigma(x) \text{ and } \sigma(y) \text{ have attributes (tentacles) } attrx \text{ and } attry\text{, respectively, and they point to the same node;}\\
G \models_{\sigma,\Sigma} type(x) = \ell & \iff & \text{the object represented by edge } \sigma(x) \text{ is an instance of class } \ell;\\
G \models_{\sigma,\Sigma} x \in X & \iff & \text{edge } \sigma(x) \text{ belongs to the set of edges } \Sigma(X).
\end{array}$$

If the formula F is closed then we will write $G \models F$ instead of $G \models_{\emptyset,\emptyset} F$. As an example, the formula $\exists p.\exists t.type(p) = Passenger \wedge type(t) = Ticket \wedge p.ticket = t.self$ holds true in the instance graph of Figure 1, using the assumption that the only source tentacle of each hyperedge is implicitly labeled by self.

We shall freely use the following obvious abbreviations for graph formulae

$$\begin{array}{rcl}
\forall x : T \,.\, \phi & \triangleq & \forall x \,.\, type(x) = T \Rightarrow \phi\\
\exists x : T \,.\, \phi & \triangleq & \exists x \,.\, type(x) = T \wedge \phi\\
x.attr = y & \triangleq & x.attr = y.self
\end{array}$$

and, in any context where a graph formula is expected,

$$\begin{array}{rcl}
x.attr & \triangleq & x.attr = true.self\\
\neg(x.attr) & \triangleq & x.attr = false.self
\end{array}$$

where the constants *true* and *false* are interpreted over the (unary) hyperedges encoding the booleans, which we assume to be included in every instance graph.

3.2 Introducing a Temporal Dimension

The behavioral logic for GTSs, called $\mu\mathcal{L}2$, is a variant of the propositional μ-calculus where propositional symbols range over formulae from $\mathcal{L}2$.

Definition 2 (Logic Over GTSs). *The syntax of $\mu\mathcal{L}2$ formulae is given by*

$$f ::= \; A \mid Z \mid \Diamond f \mid \Box f \mid \neg f \mid f_1 \vee f_2 \mid f_1 \wedge f_2 \mid \mu Z.f \mid \nu Z.f$$

where A ranges over closed formulae in $\mathcal{L}2$ and $Z \in \mathcal{Z}$ are proposition variables.

The formulae are evaluated over a *graph transition system* $T = (Q, \rightarrow)$, i.e., a transition system where the set of states Q consists of (isomorphism classes of) graphs. This can be thought of as the abstract representation of the behavior of a graph grammar $\mathcal{G}$: states in Q are (isomorphism classes) of graphs reachable in $\mathcal{G}$ and two states q_1 and q_2 are related, i.e., $q_1 \rightarrow q_2$, if q_2 is reachable from q_1 via a rewriting step in $\mathcal{G}$.

Intuitively, an atomic proposition A holds in a state q if $q \models A$ according to the satisfaction relation of the previous section. A formula $\Diamond f$ / $\Box f$ holds in a state q if some / any single step leads to a state where f holds. Note that (as in [19]) the operators $\Box$ and $\Diamond$ only refer to the next step and not (as defined elsewhere) to the whole computation. The connectives $\neg, \vee, \wedge$ are interpreted in the usual way. The formulae $\mu Z.f$ and $\nu Z.f$ represent the *least* and *greatest fixed point* over Z, respectively. When a transition system T has a distinguished *initial state* q_0, we say that *T satisfies a (closed) formula f*, written $T \models f$, if the initial state q_0 of T satisfies f. Since the logic is classical, $\Diamond$ and ν could be defined in terms of $\Box$ and μ.

Since properties of the form "eventually ϕ", i.e., $\mu Z.(\phi \vee \Diamond Z)$, and "always ϕ", i.e., $\nu Z.(\phi \wedge \Box Z)$, will be often used, we introduce the abbreviations

$$\begin{aligned} \Diamond^* \phi &\triangleq \mu Z\,.\,(\phi \vee \Diamond Z) \\ \Box^* \phi &\triangleq \nu Z\,.\,(\phi \wedge \Box Z) \end{aligned}$$

3.3 Specifying Some Properties of the Airport Scenario

In this section we discuss how some properties concerned with the Airport Case Study can be modeled in our logic. As mentioned before, the main limitation of the logic $\mu\mathcal{L}2$ resides in its propositional nature which prevents from describing the evolution of an object in time. We briefly discuss how this limitation can be overcome by considering a non-propositional extension of the temporal logic.

Using the Logic $\mu\mathcal{L}2$. The logic $\mu\mathcal{L}2$ can be used to express properties about the structure of system state, possibly at different instants.

- The plane leaves only if all passengers are aboard
  Fixed an edge pl : *Plane* representing a plane, the formula

$$\begin{aligned}\phi(pl) \quad \triangleq \quad & \exists p\colon \mathsf{Passenger}\,.\,(\exists tk\colon \mathsf{Ticket}\,.\,(p.\mathsf{ticket} = tk \wedge tk.\mathsf{plane} = \\ & pl \wedge tk.\mathsf{checked} \wedge p.\mathsf{atLoc} \neq pl))\end{aligned}$$

  means that there is a passenger p having a ticket tk associated with the plane pl, the ticket is checked but the passenger is not aboard. Hence the desired property can be expressed by saying that this can never happen for any plane which is on air, i.e., such that its atLoc attribute is *null*

$$\Box^*(\forall pl\colon \mathsf{Plane}\,.\,(pl.\mathsf{atLoc} = null \Rightarrow \neg\phi(pl)))$$

- A passenger can eat only on air
  This property is expressed by the formula:

$$\Box^*(\forall p\colon \mathsf{Passenger}\,.\,\forall pl\colon \mathsf{Plane}\,.\,((p.\mathsf{eat} \wedge p.\mathsf{atLoc} = pl) \Rightarrow pl.\mathsf{atLoc} = null))$$

  which says, assuming the existence of a tentacle eat, that it is always true that if a passenger is eating on a plane, then the plane is flying.

The validity of the above formulae over a finite-state system, like the Airport Case Study with a given initial state, can be checked by using a technique based on the unfolding semantics of GTSs and inspired to the approach originally developed by McMillan for Petri nets [20]. Recall that the unfolding construction for GTSs produces a structure which fully describes the concurrent behavior of the system, including all possible steps and their mutual dependencies, as well as all reachable states. However, the unfolding is infinite for non-trivial systems, and cannot be used directly for model-checking purposes. An algorithm proposed in [4] allows for the construction of a finite initial fragment of the unfolding of the given system which is *complete*, i.e., which provides full information about the system as far as reachability (and other) properties are concerned. Once it has been constructed, the prefix can be used to verify properties of the reachable states, expressed in the logic $\mathcal{L}2$. This is done by exploiting both the graphical structure underlying the prefix and the concurrency information it provides.

We mention that approximated techniques, also based on the unfolding semantics of GTSs, are available for systems which are not finite-state. In this case, finite under- and over-approximations of the unfolding can be constructed, which are used to check properties of a graph transformation system, like safety and liveness properties, expressed in suitable fragments of $\mu\mathcal{L}2$ [5].

A More General Logic. By experimenting with the Airport Case Study, it turns out that some interesting properties of the system cannot be expressed in $\mu\mathcal{L}2$ essentially because of its propositional nature. Take, for instance, the property "*All boarded passengers arrive at destination*". The corresponding formula should say that it is always true that, given any passenger, if *in a certain state* the passenger is boarded then *later, eventually* the passenger will arrive at its destination. This formula would have the shape

$$\Box^*(\forall p\colon \mathsf{Passenger}\,.\, p\ is\ boarded \Rightarrow \Diamond^*(p\ at\ destination))$$

which is not expressible in $\mu\mathcal{L}2$ due to the presence of the modal operator "$\Diamond^*$" in the scope of the quantifier "$\forall$".

The problem can be overcome by considering a more general, non-propositional temporal graph logic, where quantifiers and temporal operators can be interleaved. A possible syntax is given below, where the operators $\Box^*$ and $\Diamond^*$ are taken as primitive.

$$\begin{array}{rcll} F & ::= & x = y \mid x.attrx = y.attry \mid & \\ & & type(x) = \ell \mid x \in X \mid & \text{(Predicates)} \\ & & F \vee F \mid F \wedge F \mid F \Rightarrow F \mid \neg F \mid & \text{(Connectives)} \\ & & \forall x.F \mid \exists x.F \mid \forall X.F \mid \exists X.F & \text{(Quantifiers)} \\ & & \Diamond^* F \mid \Box^* F & \text{(Temporal Operators)} \end{array}$$

As before *attrx*, *attry* belong to a fixed set of *attribute names*, and x, X are first- and second-order variables, ranging over edges and set of edges, respectively.

The semantics of such logic can be defined by mimicking what is done, e.g., for first-order or second-order modal and temporal logics (see [16], or the more recent [9], where a graph logic is considered). Roughly, the logic is interpreted over a Kripke structure or a transition system where states are (first- or second-order) models. Since the logic allows to track the evolution of an individual, when a state q_1 can evolve to q_2, there must exist an explicit relationship among the elements of the models underlying such states.

More precisely, our logic could be interpreted over an *extended graph transition system* (Q, F), with Q a set of graphs and F a set of triples (q_1, f, q_2), where q_1, q_2 are states in Q and $f : q_1 \rightarrow q_2$ is a partial graph morphism. The presence of a triple (q_1, f, q_2) intuitively means that the graph q_1 can evolve to q_2. The function f relates any item in the graph q_1 which is *not* deleted by the rewriting step to the corresponding item in q_2.

By using this extended logic, several properties previously not expressible in $\mu\mathcal{L}2$ can now be easily modeled.

– All boarded passengers arrive at destination
  This property can be encoded by the following formula

$$\Box^*(\forall p\colon \mathsf{Passenger}\,.\,\forall pl\colon \mathsf{Plane}\,.\,((p.\mathsf{atLoc} = pl) \Rightarrow \exists a\colon \mathsf{Airport}\,.\,(pl.\mathsf{dest} = a \wedge \Diamond^*(p.\mathsf{atLoc} = a))))$$

  which says that, in any state, if a passenger p is boarded on a plane pl, whose destination is airport a, then the passenger p will eventually arrive at a.
– All passengers go on board
  This property can be encoded by the following formula

$$\Box^*(\forall p\colon \mathsf{Passenger}\,.\,\forall t\colon \mathsf{Ticket}\,.\,\forall pl\colon \mathsf{Plane}\,.\,(p.\mathsf{ticket} = t \wedge t.\mathsf{plane} = pl \Rightarrow \Diamond^*(p.\mathsf{atLoc} = pl)))$$

  which says that, in any state, each passenger with a ticket associated to a plane pl will eventually board on pl.
– Airports cannot move
  This property can be encoded by the following formula

$$\Box^*(\forall a\colon \mathsf{Airport}\,.\,\forall x\,.\,(a.\mathsf{atLoc} = x \Rightarrow \Box^*(a.\mathsf{atLoc} = x)))$$

  which says that an airport which is at some location will always stay there.
– Passengers change airport only by plane
  This property can be encoded by the following formula

$$\Box^*(\forall p\colon \mathsf{Passenger}\,.\,\forall a_1\colon \mathsf{Airport}\,.\,(p.\mathsf{atLoc} = a_1 \wedge \Box^*(\neg\exists pl\colon \mathsf{Plane}\,.\,(p.\mathsf{atLoc} = pl)) \Rightarrow \Box^*(\forall a_2\colon \mathsf{Airport}\,.\,((p.\mathsf{atLoc} = a_2) \Rightarrow a_2 = a_1))))$$

  which says that a passenger which is at an airport a_1 and which does not take any plane will be always in a_1.
– Baggage travels with passengers (each bag with its owner)
  This property can be encoded by the following formula

$$\Box^*(\forall p\colon \mathsf{Passenger}\,.\,\forall l\colon \mathsf{Luggage}\,.\,\forall pl\colon \mathsf{Plane}\,.\,(p.\mathsf{atLoc} = pl \wedge p.\mathsf{luggage} = l \Rightarrow \Diamond^*(p.\mathsf{atLoc} = pl \wedge l.\mathsf{atLoc} = pl))$$

which says that, in any state, if a passenger p has a luggage l and it boards on a plane pl then, eventually, also the luggage will be in pl together with the passenger.

– Passengers change location with their plane
  This property is interpreted as "a passenger on a plane will reach the same destination as the plane itself". This can be encoded by the following formula

$$\Box^*(\forall p\colon \mathsf{Passenger}\,.\,\forall pl\colon \mathsf{Plane}\,.\,\forall a\colon \mathsf{Airport}\,.\,(p.\mathsf{atLoc} = pl \;\wedge\; pl.\mathsf{dest} = a \Rightarrow \Diamond^*(p.\mathsf{atLoc} = a)))$$

  which says that, in any state, if a passenger p is on a plane pl with destination airport a then p eventually reaches airport a.

Unfortunately, the unfolding-based verification techniques mentioned for $\mu\mathcal{L}2$ do not immediately extend to this more general framework. Understanding to what extent such techniques can be adapted to the new framework is an open and stimulating direction of further research.

4 Conclusions

We presented a general approach to the specification and verification of systems modeled using UML diagrams, interpreted as graph transformation systems. More precisely, we discussed how to interpret instance diagrams as graphs, and how to implement the activities of an activity diagram as graph transformation rules. A temporal logic over monadic second-order graph predicates allows for formalizing relevant properties of the resulting system, and for fragments of such logic automatic verification techniques are available. The overall approach was presented quite informally, using a simple case study as running example.

The theoretical foundations of the methodology for verifying graph transformation systems presented in the second part of the paper are already quite well developed [3, 4, 5]. The most valuable contribution of this paper, in our view, is the idea of applying it for the verification of systems synthesized from a UML specification. This allowed us to identify some weaknesses of the approach, as described in the previous section, that we intend to address in the next future by generalizing the verification approach to more expressive logics and to graph transformation systems with synchronization.

Concerning the modeling of UML diagrams as graph transformation systems, the informal approach we discussed is clearly very preliminary, as it addresses only (restricted forms of) two kinds of diagrams. Nevertheless, we are confident, also on the basis of other contributions concerned with this topic [10, 11, 12, 13, 17, 18], that such an approach can be extended to cover a meaningful part of the UML. This represents another interesting topic for future research.

References

1. The AGILE project home page, `http://siskin.pst.informatik.uni-muenchen.de/projekte/agile/`, 2004.

2. L. Andrade, P. Baldan, H. Baumeister, et al. AGILE: Software architecture for mobility. In M. Wirsing, D. Pattinson, and R. Hennicker, editors, *Proceedings of the 16th International Workshop on Recent Trends in Algebraic Develeopment Techniques (WADT 2002)*, volume 2755 of *LNCS*, pages 1–33. Springer, 2003.
3. P. Baldan, A. Corradini, and B. König. A static analysis technique for graph transformation systems. In K.G. Larsen and M. Nielsen, editors, *Proceedings of the 12th International Conference on Concurrency Theory (CONCUR 2001)*, volume 2154 of *LNCS*, pages 381–395. Springer, 2001.
4. P. Baldan, A. Corradini, and B. König. Verifying finite-state graph grammars: an unfolding-based approach. In P. Gardner and N. Yoshida, editors, *Proceedings of the 15th International Conference on Concurrency Theory (CONCUR 2004)*, LNCS. Springer, 2004.
5. P. Baldan and B. König. Approximating the behaviour of graph transformation systems. In A. Corradini, H. Ehrig, H.-J. Kreowski, and G. Rozenberg, editors, *Proceedings of the First International Conference on Graph Transformation (ICGT 2002)*, volume 2505 of *LNCS*, pages 14–30. Springer, 2002.
6. A. Corradini, R. Heckel, and U. Montanari. Graphical operational semantics. In A. Corradini and R. Heckel, editors, *Proceedings of the ICALP Workshop on Graph Transformations and Visual Modeling Techniques (GT-VMT 2000)*, volume 8 of *Proceedings in Informatics*, pages 411–418. Carleton Scientific, 2000.
7. A. Corradini, U. Montanari, F. Rossi, H. Ehrig, R. Heckel, and M. Löwe. Algebraic Approaches to Graph Transformation I: Basic Concepts and Double Pushout Approach. In G. Rozenberg, editor, *Handbook of Graph Grammars and Computing by Graph Transformation. Vol. 1: Foundations*. World Scientific, 1997.
8. B. Courcelle. The expression of graph properties and graph transformations in monadic second-order logic. In G. Rozenberg, editor, *Handbook of Graph Grammars and Computing by Graph Transformation, Vol.1: Foundations*. World Scientific, 1997.
9. D. Distefano, A. Rensink, and J.-P. Katoen. Model checking dynamic allocation and deallocation. CTIT Technical Report TR–CTIT–01–40, Department of Computer Science, University of Twente, 2002.
10. G. Engels, J.H. Hausmann, R. Heckel, and S. Sauer. Dynamic meta modeling: A graphical approach to the operational semantics of behavioural diagrams in UML. In A. Evans, S. Kent, and B. Selic, editors, *Proceedings of the Third International Conference on the Unified Modeling Language (UML 2000)*, volume 1939 of *LNCS*, pages 323–337. Springer, 2000.
11. G.L. Ferrari, U. Montanari, and E. Tuosto. Graph-based models of internetworking systems. In B.K. Aichernig and T. Maibaum, editors, *Formal Methods at the Crossroads. From Panacea to Foundational Support*, volume 2757 of *LNCS*, pages 242–266. Springer, 2003.
12. M. Gogolla. Graph transformations on the UML metamodel. In A. Corradini and R. Heckel, editors, *Proceedings of the ICALP Workshop on Graph Transformations and Visual Modeling Techniques (GT-VMT 2000)*, volume 8 of *Proceedings in Informatics*, pages 359–371. Carleton Scientific, 2000.
13. M. Gogolla, P. Ziemann, and S. Kuske. Towards an integrated graph based semantics for UML. In P. Bottoni and M. Minas, editors, *Proceedings of the ICGT Workshop on Graph Transformations and Visual Modeling Techniques (GT-VMT 2002)*, volume 72 of *ENTCS*. Elsevier, 2003.

14. R. Heckel, J.M. Küster, and G. Taentzer. Confluence of typed attributed graph transformation systems. In A. Corradini, H. Ehrig, H.-J. Kreowski, and G. Rozenberg, editors, *Proceedings of the First International Conference on Graph Transformation (ICGT 2002)*, volume 2505 of *LNCS*, pages 161–176. Springer, 2002.
15. D. Hirsch and U. Montanari. Synchronized hyperedge replacement with name mobility. In K.G. Larsen and M. Nielsen, editors, *Proceedings of the 12th International Conference on Concurrency Theory (CONCUR 2001)*, volume 2154 of *LNCS*, pages 121–136. Springer, 2001.
16. G.E. Hughes and M.J. Cresswell. *A new introduction to modal logic*. Routledge, 1996.
17. S. Kuske. A formal semantics of UML state machines based on structured graph transformation. In A. Haeberer, editor, *Proceedings of the Fourth International Conference on the Unified Modeling Language (UML 2001)*, volume 2185 of *LNCS*, pages 241–256. Springer, 2001.
18. S. Kuske, M. Gogolla, R. Kollmann, and H.J. Kreowski. An integrated semantics for UML class, object and state diagrams based on graph transformation. In *Proceedings of the Third International Conference on Integrated Formal Methods (IFM 2002)*, volume 2335 of *LNCS*, pages 11–28. Springer, 2002.
19. C. Loiseaux, S. Graf, J. Sifakis, A. Bouajjani, and S. Bensalem. Property preserving abstractions for the verification of concurrent systems. *Formal Methods in System Design*, 6:1–35, 1995.
20. K.L. McMillan. *Symbolic Model Checking*. Kluwer, 1993.
21. D. Plump. Term graph rewriting. In H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors, *Handbook of Graph Grammars and Computing by Graph Transformation. Vol. 2: Applications, Languages, and Tools*. World Scientific, 1999.
22. J. Rumbaugh, I. Jacobson, and G. Book. *The Unified Modeling Language Reference Manual*. Addison Wesley, 1999.