ArchiTRIO: a UML-compatible language for architectural description and its formal semantics

Matteo Pradella[2], Matteo Rossi[1], and Dino Mandrioli[1,2]

[1] Dipartimento di Elettronica e Informazione, Politecnico di Milano
[2] CNR IEIIT-MI
Piazza Leonardo da Vinci 32, 20133 Milano, Italy
{pradella, rossi, mandrioli}@elet.polimi.it

Abstract. ArchiTRIO [14] is a formal language, which complements UML 2.0 concepts with a formal, logic-based notation that allows users to state system-wide properties, both static and dynamic, including real-time constraints. In this paper we present the semantics of the core concepts of the ArchiTRIO language. As the core elements of ArchiTRIO coincide with those of UML 2.0 (operation, interface, port, class), the semantics of ArchiTRIO provides also a formal definition for the basic concepts on which UML 2.0 is built.

Keywords: UML, software architecture, formal methods, real-time

1 Introduction

In the last few years, UML [8] has risen to the status of *de facto* standard for system modeling in industrial practice. Its appeal originates from a number of factors such as ease of use and a certain degree of intuitiveness and flexibility in the notation (probably rooted in the borrowing form previous, well-established notations), which reduce the effort needed to be able to write UML models to a minimum. In its 2.0 incarnation, UML includes constructs (e.g., component, connector, port) that were previously missing, which are necessary for describing system architectures. Alas, as with the previous versions, UML lack of formality hampers its applicability to critical systems, where precise and rigorous designs are of the utmost importance for the correct development of the application.

In [14] we sketched a novel approach to providing UML with the degree of formality that is necessary for rigorous modeling and verification, one that hinges on the idea of *complementing* the UML notation of class and composite structure diagrams [8] with a temporal logic-based notation. This combination of UML and logic-based notation results in a formal language, called ArchiTRIO. [14] presents the ArchiTRIO approach to system modeling, which falls essentially in the category of lightweight methods [16]; more precisely, ArchiTRIO allows developers to use standard UML 2.0 notation to describe non-critical aspects of systems, but it also offers a complementary formal notation, fully integrated with the UML one, to represent those system aspects that require precise modeling. ArchiTRIO, then, adds expressive power to UML diagrams, rather than replacing or modifying any of them: a user who at first does not need full-blown ArchiTRIO can start by drawing bare UML class diagrams, and only later, when the need arises for clarity and precision (especially for temporal constraints), introduce ArchiTRIO-specific notation.

ArchiTRIO is based upon few selected UML 2.0 constructs especially suited for describing architectures, it gives them a formal meaning, and precisely defines their composition. [14] mainly focuses on the principles behind ArchiTRIO and suggests guidelines for its application. This paper presents in some detail the semantics of the language. The semantics of ArchiTRIO is given in terms of HOT (Higher-Order TRIO), which is a higher-order extension of our previous first-order temporal logic TRIO [4]. We chose to found ArchiTRIO on a higher-order logic to allow for the concise representation of mechanisms such as the passing of parameters that have an ArchiTRIO/UML class for a type.

In our opinion, the distinguishing feature of HOT is its simplicity, rooted in the rigorous application of the principle of identifying the concepts of class and of abstract data type (see, e.g. [1]), which is seldom completely pursued in traditional object-oriented languages. Since ArchiTRIO has many concepts in common with UML (class, port, etc.), providing a semantics for the former amounts also to giving a formal definition for a number of UML elements.

This paper is structured as follows: Section 2 briefly summarizes the features of ArchiTRIO presented in [14]; Section 3 provides an overview of HOT and of its set-theoretic semantics; Section 4 builds upon it to define the semantics of ArchiTRIO; Section 5 compares the present work with some relevant literature, and especially with the OCL [7]; finally, Section 6 draws some conclusions and hints at future works in this area of research.

2 A brief overview of ArchiTRIO

In this section, we briefly summarize the ArchiTRIO approach originally presented in [14], and introduce a simple running example, an access control system for a building divided into areas having different security levels, which we will use throughout this article to illustrate the features of ArchiTRIO.

Consider an Access Control System used in one or more corporate buildings having three different security levels: *low*, *medium*, and *high*. The building may contain zero or more areas of a given security level. The access control is enforced essentially through two kinds of entities: a local mechanism based on the concept of *security gate*, and a *central control* connected to a user database.

Figure 1 shows the UML class diagram describing the situation above. It depicts a `CentralControl` class, the main entity which enforces the prescribed security policy for user access; a `UserDB`, that is a database containing users' sensible data and their actual security clearance; and three kinds of `Gate` classes: `SimpleGate`, `MediumSecurityGate`, and `HighSecurityGate`, in charge of man-

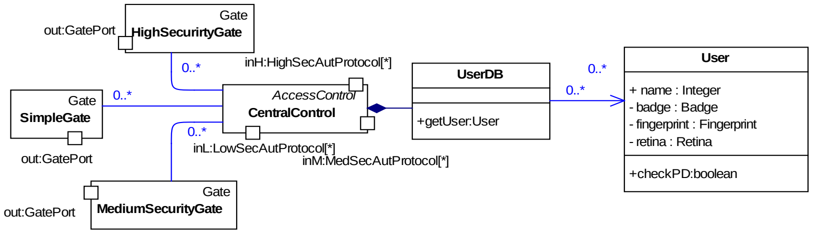

Fig. 1. Access Control System: the high-level class diagram.

aging the local access to areas with low, medium, and high security level, respectively. Every gate has a *port* of type `GatePort`, while `CentralControl` has three different ports, `LowSecAutProtocol`, `MedSecAutProtocol`, and `HighSecAutProtocol` that are used to communicate with `SimpleGate`s, `MediumSecurityGate`s, and `HighSecurityGate`s, respectively.

Moving in a top-down fashion, we now define the internal class structure of the gates (for space reasons, we omit the corresponding diagram; the interested reader can refer to [14]). A `SimpleGate` is an entity having one or more `BadgeReaders` (a subclass of `IdRecognizer`), managed by a local controller `LC_SimpleGate`. Communication between `BadgeReader` and `LC_SimpleGate` is based on the interface `LocalControl`, implemented by the latter. A `MediumSecurityGate` is based on a more sophisticated `IdRecognizer`, i.e. a `FingerprintsReader`, and has an `EntrySensor`. Analogously to the simple gate, a medium security gate is supervised by a local controller, `LC_MedSecGate`, and communication between the local controller and the sensors is based on the interface `LocalControl`. The most complex type of gate is the `HighSecurityGate`: it consists of two kinds of `IdRecognizers`, a `FingerprintsReader` and a `RetinaScanner`; an `EntrySensor`; and a local controller `LC_HighSecGate`. A high security gate is opened only after both the user's fingerprints and retina are successfully checked.

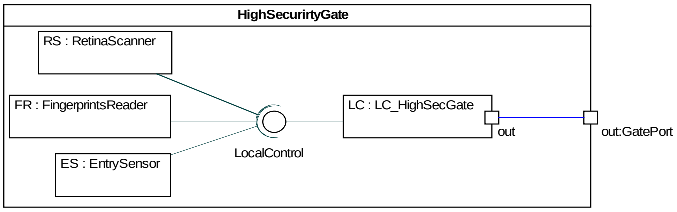

Fig. 2. Composite structure diagram of a high security gate.

Consider now for instance the structure of a high security gate (Figure 2). It consists of a retina scanner (`RS`), a fingerprints reader (`FR`), an entry sensor (`ES`), and a local control (`LC`). Every component is an instance of the corresponding class; LC exchanges data with the sensors by implementing the interface LocalControl, while communication with the remote central control happens through a replicated port of type GatePort.

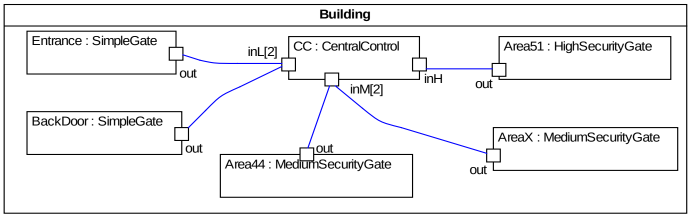

Fig. 3. The building structure: the high-level system architecture.

Finally, consider the system high-level architecture (Figure 3). It consists of: a central control (CC); two low security gates (Entrance and BackDoor); two medium security areas and their corresponding gates (AreaX and Area44); and one high security area reachable through a high security gate (Area51).

This concludes a first simple architectural description of the system, based exclusively on UML constructs. As we said in the introduction, UML *per se* does not precisely define many of the constructs we used for describing our system here. For instance, it lacks a precise definition of timeouts management and local control behavior. More generally, we would like to be able to precisely express a critical property and possibly to verify it. At this point the designer, e.g. of a critical system, could need something more than plain UML, to add desired properties and system requirements into its architecture. So ArchiTRIO appears in the picture: the designer needs a solid formal description of the used concepts (e.g. class, instance, interface, port, operation, connection, and so on), to state something more and more precisely of the system, well before implementing it.

The basic ArchiTRIO concepts mirror a subset of the elements one can find in UML 2.0. The core of the language is the *class*. A class defines *operations* and *attributes*, and can provide and require *interfaces*; *ports* are groups of required/provided interfaces, and can be used to define protocols. Classes can have *composite structures*, whose parts are connected by *connectors*. The graphical representation of those concepts that are common to both ArchiTRIO and UML is the same as in UML. Besides these UML elements, however, ArchiTRIO includes also concepts derived from temporal logic, which allow users to precisely define the behavior of a system modeled with ArchiTRIO.

For example, class LC_HighSecGate provides interface LocalControl and has a port of type GatePort; interface LocalControl defines two operations, incomingData and personEntered. In addition to the aforementioned UML port and interface, class LC_HighSecGate includes three logic items, inGate, lastUser and gate_open. Item inGate is time-independent (TI, meaning that its value is constant over time), and represents the identifier of the Gate to which the controller belongs; item lastUser is time-dependent (TD, that is its value depends on the time instant in which the item is evaluated) and models the data corresponding to the user who had either his/her fingerprints or his/her retina scanned; item gate_open, instead, is a state (which means that it is true/false in intervals of non-null duration), and models the intervals in which the gate is open.

In addition to the logic items explicitly declared in the class signature, an ArchiTRIO class includes a number of built-in items, which model the most significant features of the UML elements of the class (for example the parameters of an operation, an operation invocation, etc.). Then, the axioms of class LC_HighSecGate predicate over the logic items (explicitly declared or built-in) of the class to define its precise behavior. Axiom dataRelay shown below, for example, states that when an invocation of operation incomingData (exported through interface LocalControl) is received by the controller and the value of the rawData parameter is pd, within T time units in the future the controller will invoke (an instance of) operation sendPersData on port out, passing pd and the value corresponding to item inGate as parameters.

```
dataRelay:
  iD.inv_rec(pd) -> ex out.sPD(WithinF(out.sPD.invoke(pd, inGate), T));
```

In axiom dataRelay, iD and sPD are variables ranging over all possible *invocations* of operations incomingData and sendPersData, respectively. Then, ex out.sPD means that "there exists an invocation of operation sendPersData (within the scope of port out) such that...". inv_rec and invoke are built-in logic items (more precisely *events*, i.e. predicates that are true only in isolated time instants) modeling significant events of an operation invocation; in particular, event iD.inv_rec is true when invocation iD of operation incomingData is received by the local controller; similarly, event out.sPD.invoke is true when the controller issues invocation sPD on port out. WithinF is a temporal operator taken from the TRIO formal language (see [2] for its definition). pd is a variable of type PersonalData, where PersonalData is an ArchiTRIO class, not shown here for the sake of brevity, modeling either the badge, or the fingerprints, or the retina of a user.

Finally, a *port* is a collection of provided and required interfaces. It can be used to define a *protocol*, intended as a combination of invocations of operations that can be received (from a provided interface) or issued (to a required interface). Thus, an ArchiTRIO port can contain axioms defining the corresponding protocol in terms of the involved operation invocations. Consider, for example, port HighSecAutProtocol of Figure 1. It provides interface AccessControl, and requires one instance of interface FromAccessControl. The port defines the authentication protocol for gates that require that a user authenticates him/herself through both a fingerprint and a retina scan. More precisely, the two scans can occur in any order, but always within a maximum delay one from the other for the authentication to be successful (i.e. for the controller to allow the user to enter by opening the gate through an openGate command). Further details can be found in [14].

This concludes our simple and informal overview of ArchiTRIO. The next sections will cover these concepts more formally.

3 Higher-Order TRIO

Items are the founding elements of the HOT logic. HOT *items* correspond, in usual logic lore, to *constants*, *functions* or *predicates*. Items can have arguments (and return values), which are *typed* elements. The arguments (and returned values) of HOT items can be of any HOT type (see below). For example, we might define a HOT item `it` to be a time-dependent (`TD`) predicate with two arguments of type `T1` and `T2`:

```
items:  TD it(T1, T2) : boolean;
```

Items are the building blocks for HOT *formulas*. HOT formulas are, as usual, a combination of functions, predicates (that is, items), logical connectors (`&`, `|`, `->`, `<->`, `not`), temporal operators (`Dist`, `Futr`, `Past`, etc.) and quantifiers (`all`, `ex`). For example:

```
p1(f1, f2(c1)) -> Futr(all x(p2(x)), t);
```

Notice that every HOT variable (for example `c1`, `x` and `t` in the formula above) ranges over the values of some *type* (or *domain*), which is defined through a HOT *class*. A HOT class definition is essentially divided in two parts: the first part contains the local items; the second part contains the *axioms*. Axioms are formulas which model the behavior of the class; that is, they constrain its items. Given a HOT class, an element of the domain is called an *object*. The term object is synonym for *instance* (of a class) and *value* (of a type). A HOT object corresponds also to a *model* for the corresponding class (or, in TRIO terms, to a *history*); essentially, being HOT a temporal logic, an object is a function of time. As a consequence there is no notion of object creation and destruction as in operational languages. This approach differs from the usual related literature, but basically follows and extends the traditional TRIO class-oriented approach (see [2] for more details).

Modules TRIO has a primitive notion of *module* that sharply distinguishes it from the notion of item: a module is an instance of a class contained in an instance of another class. HOT instead, thanks to being higher order does not need such a separation. Rather, it has linguistic constructs that allow one to obtain the same semantics as TRIO modules in HOT from basic HOT concepts. Thus, HOT offers the keyword module as a shorthand notation to automatically introduce the axioms and definitions corresponding to the semantics of TRIO modules, where essentially a module is represented through a HOT item. As a brief example, consider a class `C` containing an array `m` of `n` modules of class `M`:

```
modules: m[1..n] : M;
```

This array corresponds in HOT to the following time-independent item:

```
items: TI m(1..n) : M;
```

Inheritance We distinguish two kinds of inheritance: *monotonic inheritance*, and *free, purely syntactic, inheritance*. Monotonic inheritance perfectly matches the notion of subtyping (in fact it is written C' *subtype of* C, or $C' \preceq C$):

1. every item and axiom defined in C is in C';
2. C' may add new items;
3. C' may add new axioms, thus more constraints w.r.t. C.

The subsection about semantics below shows how this simple notion of inheritance produces pure subtyping, as, e.g., in [1].

Instead, purely syntactic inheritance (written C' *redefines* C) is a free-form of inheritance: C' may modify, add, and delete any items and axioms of C.

We introduce both types of inheritance to make HOT a very free kind of logic language. Nonetheless, in our opinion the correct interpretation of inheritance is essentially subtyping. Although in this paper we do not deal with methodological aspects, we envisage a methodology where the specifier/designer could start with a class hierarchy in which both kinds of inheritance are used, being sometimes easier to work with a free form of inheritance, to later obtain, through some revision steps, a true tree in which only subtyping is used. For instance, consider the system presented in Section 2. The designer at first writes class `SimpleGate`, because it is the simplest kind of gate. Then, she decides to add another, more complex kind of gate: the `MediumSecurityGate`. She creates class `MediumSecurityGate` using syntactic inheritance from `SimpleGate`, replacing the `BadgeReader` component with one of type `FingerprintsReader`, then adding a new component of type `EntrySensor`. Later, rethinking about the relation between these two classes, she decides to collect all the concepts common in gates in a new class `Gate`, of which both `SimpleGate` and `MediumSecurityGate` become subtypes.

Genericity HOT classes can be parametric with respect to values of classes and with respect to classes. The header of a generic HOT class has the syntax `class <class_name> ( <par_decls> )` where parameters may be a type name or a value of a certain type.

Hints of HOT's set-theoretic semantics Given an item i of class C, let us call $sig(i)$ its signature[3]. Moreover, let us call $items(C)$ the set of items locally defined in C (e.g., if $items(C) = i_1, i_2$, and C' is a subtype of C that adds a single new item i_3 to C, then $items(C') = i_3$). Quite naturally, items are interpreted as (time dependent or independent) constants, functions or predicates, depending on their signature. Axioms are essentially constraints on the items. Classes are types, therefore are interpreted as sets of objects. An object x of class C is, in general, a function of time (τ):

$$x : \tau \rightarrow \bigtimes_{i \in I} sig(i), \text{ where } I = \bigcup_{C' \preceq C} items(C').$$

[3] For simplicity, in the following we do not consider homonyms and name clashes.

Therefore $x.i$ is interpreted as a projection of the range of x on the component $sig(i)$. For example, let C be a class with items n of type `natural`, and s of type `string`, with axioms stating that always $n = length(s)$. Let C' be a subtype of C, containing a new item c of type `char`, and a new axiom which states that $c = s[0]$. Then, every object in C or C' can be interpreted as a function with signature: $\tau \rightarrow \texttt{natural} \times \texttt{string} \times \texttt{char}$. The main difference is that C does not constrain in any way item c, therefore $C' \subseteq C$.

4 The semantics of ArchiTRIO

We now define the semantics of the core elements of ArchiTRIO on the stage of HOT: classes (possibly composite), operations, interfaces/ports. For reasons of brevity, we do not present the full semantics of ArchiTRIO, but only a significant subset thereof; the elements presented here, however, should provide a meaningful enough picture.

4.1 ArchiTRIO classes

An ArchiTRIO class is a HOT class, and defines a type; then, as in HOT, an ArchiTRIO object of type `AT` is an instance (i.e. a value) of class `AT`.

All ArchiTRIO classes are subtypes (in terms of HOT) of a HOT class `ArchiRootClass`; that is, all ArchiTRIO classes *implicitly* share a common root class, which thus defines a type that is common to all ArchiTRIO objects. A class can include *operations* and *attributes*. An attribute is, quite naturally, represented through an item modeling its value, and operations to get/set it. Then, its semantics does not raise specific issues besides those associated with the notion of operation.

4.2 Operations

The concept of ArchiTRIO (and, thus, UML) operation is defined through a HOT class `Operation`, which captures the core features shared by all operations. These features can be summed up as: 1) a set of items modeling the key aspects of an operation invocation (when the client object issues the invocation, when the server receives it, the parameters associated with the invocation, etc.), and 2) a set of axioms defining the constraints — time-related or not — over the aforementioned items (for example, the fact that a return must be preceded by the server object actually receiving the invocation, etc.). HOT class `Operation` defining the semantics common to all operations is sketched below.

```
class Operation
items:
  event invoke, inv_rec, reply; ...
axioms:
  Response_NC: reply -> SomP(inv_rec);
  ...
end
```

Class Operation introduces the logic items modeling the relevant features of an operation invocation (e.g. the invoke, inv_rec and reply events first introduced in Section 2), and the axioms defining the behavior that is common to all invocations. For example, axiom Response_NC defines a necessary condition for the reply event to occur: an operation invocation can return (occurrence of the reply event) only if the invocation was previously received by the called object (event inv_rec; see [2] for the definition of temporal operator SomP). Every instance o of class Operation corresponds to a *single* invocation of an operation. Then, for any instance o, the corresponding events invoke, reply, etc. are *unique*; that is, they can happen only *once* over the temporal domain. This property is defined by suitable axioms in HOT class Operation; for instance, formula invoke_unique states in an obvious way that, if event invoke occurs now, it cannot occur in any other instant of the temporal domain.

Every invocation is also characterized by a pair of objects: one that issues the invocation, and one that receives it. This is represented in the HOT semantics through a pair of items, src and tgt, both constants of type ArchiRootClass, modeling, respectively, the source of the invocation and its target.

A specific operation (e.g. sendPersData of interface AccessControl) is defined as a subtype of HOT class Operation. For example, class sendPersData below defines the semantics for the corresponding operation (see [14] for the complete declaration of the operation). Every instance spd of class sendPersData (i.e. every value of type sendPersData) is an invocation of the corresponding operation.

```
class AccessControl.sendPersData
subtype of: Operation;
items:
  TI rawData : PersonalData;
  TI gate : GateId;
  TI partial returned : User;
  TI partial raised : UserNonExistentException; ...
axioms: ...
end
```

The parameters of an operation are represented through constants having the same type of the parameter. For example parameter rawData of operation sendPersData corresponds to a constant with the same name in the HOT class sendPersData; then, given an instance sPD of class sendPersData, sPD.rawData is the value of parameter rawData for that invocation. Similarly, if an operation returns a value (resp. raises an exception), this is represented through a constant returned (resp. raised) of the same type as the returned value (resp. raised exception). Constant item returned is declared as partial, meaning that its value can be undefined, for example if the invocation ends with an exception (similarly, item raised is partial since its value is undefined if the invocation ends correctly). When the instance of an operation appears in a formula, the server object which it refers to is included, as a prefix, in the term identifying the instance (if the server object is not included, it defaults to this, i.e., the current object in which the formula is defined). For example, if sPD is a variable of type sendPersData and ac is a term corresponding to an instance of a class providing interface AccessControl (that is, operation sendPersData), to refer to an invocation of operation sendPersData on object ac we have to write ac.sPD (see for example formula dataRelay of class LC_HighSecGate). This corresponds to stating that the target object of invocation sPD is ac or, using the HOT semantics, that sPD.tgt = ac.

4.3 Interfaces and ports

From a semantic point of view, an ArchiTRIO interface is a class that exports operations (and attributes), but cannot include other logic items (such as, for example, state gate_open of class LC_HighSecGate), nor can be decomposed into parts (i.e. it cannot be composite, but merely simple). The only possible associations that an interface can have are a generalization relationship with other interfaces, and a "provided by" relationship with an ArchiTRIO class; it cannot, for example, require an interface.

A class providing an interface I is a subtype of HOT class I. ArchiTRIO allows a class (resp. interface) to provide (resp. specialize) more than one interface. Then, the corresponding HOT class is a subtype of every and each one of the provided (resp. specialized) interfaces. An ArchiTRIO class requiring an interface I is a HOT generic (i.e. parametric) class with respect to a parameter of type I. For example, class RetinaScanner provides interface IdRecognizer and requires an interface LocalControl. The corresponding HOT class is shown below.

```
class RetinaScanner (lc : LocalControl)
subtype of: IdRecognizer; ...
end
```

As detailed above, HOT class RetinaScanner has one parameter, lc, of type LocalControl; that is, every object rs of class RetinaScanner must be instantiated with an object providing interface LocalControl. One way to provide an instance c of a class C requiring an interface I with the necessary instances of I is by *connecting* c with an object providing I in a Composite Structure Diagram, as explained in Section 4.4. A port in ArchiTRIO is a class that provides a (possibly empty) set of interfaces PI, and requires a (possibly empty) set of interfaces RI. In addition, an ArchiTRIO port can include a set of axioms, which define a protocol associated with the port. A port, like an interface, cannot include logic items, nor be decomposed further into parts; it may specialize another port, but not other kinds of classifiers (classes and interfaces). Only ArchiTRIO classes (neither interfaces, nor other ports) may offer a port.

Being an ArchiTRIO class, a port is defined as a HOT class that requires and provides the corresponding interfaces. For example, the HOT semantics of port HighSecAutProtocol shown in Figure 1 is the following:

```
class HighSecAutProtocol (fac : FromAccessControl)
subtype of: AccessControl; ...
end
```

The HOT semantics of an ArchiTRIO class `C` that offers a port `p` of type `P` is that of a class having a module `p` of type `P`. The multiplicity of every port component (i.e. how many instances of a port `P` an instance of `C` actually offers) is a parameter of the class offering it, if it is left open in the class definition (e.g., when defined as `[1..*]`). For example, class `CentralControl` of Figure 1 has three ports: `inH` of type `HighSecAutProtocol`, `inM` of type `MedSecAutProtocol` and `inL` of type `LowSecAutProtocol`. The actual multiplicity of these ports is decided when class `CentralControl` is instantiated (for example in the example of building of Figure 3, instance `CC` has two instances each of ports `MedSecAutProtocol` and `LowSecAutProtocol`, and one instance of port `HighSecAutProtocol`). Then, the HOT semantics of class `CentralControl` is the following[4]:

```
class CentralControl (N_inH : Natural, N_inM : Natural, N_inL : Natural,
                      inH_fac : FromAccessControl[1..N_inH], ...)
modules:
  inH[1..N_inH]: HighSecAutProtocol(inH_fac);
  inM[1..N_inM]: MedSecAutProtocol(...);
  inL[1..N_inL]: LowSecAutProtocol(...); ...
end
```

Note that the number of ports `N_inH`, `N_inM` and `N_inL` are parameters of class `CentralControl`, and are set when the class is instantiated.

4.4 Composite classes

The parts of an ArchiTRIO composite class are defined, in a natural manner, through HOT modules. For example, the semantics of class `HighSecurityGate` of Figure 2 is the following (note the definition of port `out` as a module of the class, in accordance to the discussion of Section 4.3):

```
class HighSecurityGate (out_ac: AccessControl)
modules:
  out: GatePort(out_ac);
  LC: LC_HighSecGate;
  RS: RetinaScanner(LC);
  FR: FingerprintsReader(LC);
  ES: EntrySensor(LC); ...
end
```

As shown above, the HOT semantics of a connection between a provided and a required interface (such as the one between components `LC` and `RS` in Figure 2) is that of parameter instantiation. Then, for example, in object `RC` of class

[4] Parameter `inH_fac` is a sequence of `N_inH` objects of type `FromAccessControl`.

`RetinaScanner` (which requires an interface of type `LocalControl`) parameter `lc` is instantiated with object `LC`, which is precisely of type `LocalControl` (similarly for object `FR` and `ES`). There are two kinds of connectors between ports. The first one corresponds to the situation in which two ports of the same kind, one belonging to a composite class, and one belonging to one of its components, are connected with each other (this is, for example, the case of ports out of class `HighSecurityGate` and of its part `LC`). The second one, instead, corresponds to the configuration in which a port `P` providing interfaces `PI1, ..., PIn` and requiring interfaces `RI1, ..., RIm` is connected to a complementary port `Pc` *requiring* interfaces `PI1, ..., PIn` and *providing* interfaces `RI1, ..., RIm`. This second case occurs, for example, in class `Building` of Figure 3, where port `inH` of component `CC`, which has type `HighSecAutProtocol`, is connected to port out of `Area51`, which has type `GatePort`; in fact, port `HighSecAutProtocol` provides interface `AccessControl` and requires interface `FromAccessControl`, while port `GatePort` requires interface `AccessControl` and provides interface `FromAccessControl`, and is thus complementary to the former. Informally, in the first kind of connection between ports (the one exemplified by class `HighSecurityGate` and its part `LC`) the composite class relays *instantly* all signals arriving at the outermost port `p_out` to the innermost one `p_in` (and vice-versa). Then, all traces of port `p_out` are also traces for `p_in`. This corresponds to `p_out` and `p_in` actually being the *same* object. In the case of class `HighSecurityGate`, for example, this corresponds to stating that `out = LC.out`. The second kind of connection is instead an extended version of the connection between provided and required interfaces described above. Then, the connection between components `CC` and `Area51` in class `Building` has the following semantics:

```
class Building ...
modules:
  CC: CentralControl(1, 2, 2, [Area51],...);
  Area51: HighSecurityGate(CC.inH); ...
end
```

where `[Area51]` is a sequence of exactly one object, `Area51` (which has type `FromAccessControl`, as required by the definition of parameter `inH_fac` of class `CentralControl` above).

5 Related Works

ArchiTRIO is a formal language that includes a number of concepts from UML 2.0, and assigns them a precise semantics. As a consequence, it is related to a number of works that have appeared in the literature in recent years. In this section, we take into account some of the aforementioned works, and briefly analyze how our approach differs from previous ones.

ArchiTRIO is a logic-based language, and indeed the UML notation already includes a logic language, the Object Constraint Language (OCL) [7]. With respect to OCL, however, ArchiTRIO has larger scope and greater expressiveness.

In fact, OCL is a language for specifying "[...] invariants on classes and types in the class model [...] pre- and post conditions on Operations and Methods [...] constraints on operations [...]" [7]. With ArchiTRIO one can express all of these properties and some more; for example, axiom dataRelay shown in Section 2, which defines neither a class invariant, nor a pre/post condition (nor a constraint) on an operation, but, rather, a dynamic relationship between two different operations, cannot be expressed as an OCL constraint[5]. In addition, OCL expressions are forbidden to "alter the state of the corresponding executing system" (i.e. they are *side-effect-free*), and they can describe the computation of an operation only if this is side-effect-free (in UML terms, only if it has an *isQuery* tag). ArchiTRIO formulas, on the other hand, do not have any of these restrictions, and can easily formalize properties such as "as a result of an invocation of operation Op, the value of attribute A becomes X".

Also, it is well-known that OCL cannot express real-time constraints. A real-time extension to OCL has been proposed in [15] to express real-time constraints on Statecharts. In the approach of [15], real-time OCL formulas can state that, for example, "if the class is in state X, then on all execution traces of the underlying Statechart state Y must follow after no less than T1 and no more that T2 time units" (where T1 and T2 must be either integer constants or the keyword inf); that is, RT-OCL formulas make sense only when interpreted with respect to the Statechart associated with the class. ArchiTRIO formulas, instead, are more expressive as far as temporal constraints are concerned (they allow quantifications over temporal variables, which can range not only over discrete, but also over continuous temporal domains), and have a higher level of abstraction. In fact, while RT-OCL formulas refer to a specific computational environment (i.e. the one given by the Statechart of the enclosing class), ArchiTRIO ones assume very little (for example that a reply must be preceded by an invoke), and they themselves define the possible computations of the class they refer to.

[9] presents a formal semantics for object systems with particular emphasis on how objects react to the stimuli (called *requests*) coming from other objects. In addition, it introduces a notion of *substitutability* between objects based on behavioral conformity. The present work exhibits some similarities with [9], in that we also provide a formal semantics for systems composed of communicating objects, and introduce a notion of *subtyping* that hinges on the principle that a subtype can be used wherever a parent type appears (in HOT/ArchiTRIO terms, it guarantees that the axiom formulas of the parent still hold). Notice, however, that while [9] refers to state-based specifications (for example ones given through Statecharts), ArchiTRIO belongs to the category of axiom systems, hence the two notions of substitutability and subtyping are inherently different, even if

[5] One could argue that such a property (minus the real-time constraint) could be expressed in UML by means of a Statechart or an Activity Diagram. However, this only highlights the fact that while in ArchiTRIO there is a unique formalism for all aspects of the model (static and dynamic), basic UML relies on a number of overlapping views, which often express similar properties and can be difficult to reconcile with one another.

related (the former is based on the concept of trace containement/simulation, while the latter on the concept of subset/logical implication). In addition, while [9] basically offers a semantics of Statecharts describing communicating objects, ArchiTRIO has a wider reach, as it encompasses the definition of the whole system, in both its structural and dynamic features.

How to add formality to existing UML is a widely acknowledged problem. In this regard, a number of works in the literature have proposed an approach based on *translating* UML behavioral diagrams (especially Statecharts and sequence diagrams) into an existing formalism (be it π-Calculus [10], TRIO [11], Promela [13], and many others not listed here for the sake of brevity), or, alternatively, into an ad-hoc model [12]. The ArchiTRIO approach is different in that we do not translate any UML dynamic diagram into an existing formalism; on the contrary, we developed a formal language that is *integrated* into the UML 2.0 notation, which allows one to precisely describe both the structure and the dynamics of a system, of its components and their interactions, with particular attention to their temporal constraints.

Finally, [6] presents an approach to the analysis of system architectures based on a subset of UML 2.0 concepts and a formal semantics for time-annotated Statecharts. Again, with respect to this work, the scope of ArchiTRIO is wider, as it is intended for use in the whole system design phase, from modeling to verification. In fact, one could see the techniques presented in [6], and associated notations, as a target model, to be obtained through a suitable method from an ArchiTRIO design to perform subsequent verification.

6 Conclusions

We presented the semantics of the ArchiTRIO language [14]. Since ArchiTRIO shares many concepts with UML, its semantics effectively corresponds to a formal definition of a number of important UML concepts. The semantics of ArchiTRIO is given in terms of a higher-order temporal logic, HOT, which is endowed with a notion of subtyping built upon the simple and intuitive concept of subset.

Our further work on the ArchiTRIO language will follow a number of directions. First and foremost, we are currently developing an integrated tool-set, called TRIDENT, which is based on the Eclipse [3] platform, to support writing ArchiTRIO models. This tool, of which an early prototype exists, will be able to import UML diagrams from external tools, both commercial and noncommercial, and will allow users to add ArchiTRIO-specific details to those parts of the model that require a greater level of rigor and precision. Secondly, we will investigate verification techniques (to be supported by TRIDENT) to complement the modeling features presented in this paper. In this regard, the semantics of ArchiTRIO in terms of HOT suggests an encoding of ArchiTRIO classes into the higher-order logic of a theorem prover such as PVS, along the lines already followed for the TRIO language [5].

The ultimate goal of our research is the development of a complete "UML-compliant and compatible", fully tool supported methodology that allows one to move smoothly from a purely logic high-level specification to architectural design to implementation through a sequence of refinement steps, the correctness of each one being rigorously verified by exploiting several complementary methods.

Acknowledgements

The authors would like to thank the anonymous reviewers for their useful comments and suggestions.

References

1. H. Balsters and M. M. Fokkinga. Subtyping can have a simple semantics. *Theoretical Computer Science*, 87(1):81–96, 1991.
2. E. Ciapessoni, A. Coen-Porisini, E. Crivelli, D. Mandrioli, P. Mirandola, and A. Morzenti. From formal models to formally-based methods: an industrial experience. *ACM TOSEM*, 8(1):79–113, 1999.
3. Eclipse Foundation. `http://www.eclipse.org`.
4. C. A. Furia, D. Mandrioli, A. Morzenti, M. Pradella, M. Rossi, and P. San Pietro. Higher-order TRIO. Technical report, DEI, Politecnico di Milano, 2004.
5. A. Gargantini and A. Morzenti. Automated deductive requirements analysis of critical systems. *ACM TOSEM*, 3(3):225–307, 2001.
6. H. Giese, M. Tichy, S. Burmester, and S. Flake. Towards the compositional verification of real-time uml designs. In *Proc. of ESEC/FSE 2003*, pages 38–47, 2003.
7. Object Management Group. UML 2.0 OCL specification,. Technical report, OMG, 2003. ptc/03-10-14.
8. Object Management Group. UML 2.0 superstructure specification,. Technical report, OMG, 2003. ptc/03-08-02.
9. D. Harel and O. Kupferman. On object systems and behavioral inheritance. *IEEE Transactions on Software Engineering*, 28(9):889–903, 2002.
10. V. S. W. Lam and J. Padget. Formalization of uml statechart diagrams in the π-calculus. In *Proc. of the 2001 Australian Soft. Eng. Conf.*, pages 213–223, 2001.
11. L. Lavazza, G. Quaroni, and M. Venturelli. UML and formal notations for modelling real-time systems. In *Proc. of ESEC/FSE 2001*, pages 196–206, 2001.
12. X. Li, Z. Liu, and H. Jifeng. A formal semantics of UML sequence diagram. In *Proc. of the 2004 Australian Soft. Eng. Conf.*, pages 168–177, 2004.
13. W. E. McUmber and B. H. C. Cheng. A general framework for formalizing UML with formal languages. In *Proceedings of the 23rd ICSE*, pages 433–442, 2001.
14. M. Pradella, M. Rossi, and D. Mandrioli. A UML-compatible formal language for system architecture description. In *SDL 2005: Proc. of 12th SDL Forum*, volume 3530 of *Lecture Notes in Computer Science*, pages 234–246. Springer-Verlag, 2005.
15. S. Flake S. and W. Mueller. Formal semantics of static and temporal state-oriented OCL constraints. *Software and Systems Modeling*, 2(3):164–186, 2003.
16. H. Saiedian, J. P. Bowen, R. W. Butler, D. L. Dill, R. L. Glass, D. Gries, A. Hall, M. G. Hinchey, C. M. Holloway, D. Jackson, C. B. Jones, M. J. Luts, D. L. Parnas, J. Rushby, J. Wing, and P. Zave. An invitation to formal methods. *IEEE Computer*, 29(4):16–30, 1996.