A UML-compatible formal language for system architecture description

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

[1]Dipartimento di Elettronica ed Informazione, Politecnico di Milano and
[2]CNR IEIIT-MI
via Ponzio 34/5,
20133 Milano, Italy
{pradella, rossi, mandrioli}@elet.polimi.it

1 Introduction

Nowadays, UML is the *de facto* standard for system modeling in industrial practice. Its popularity derives from a number of factors such as simplicity, ease of use and a certain degree of intuitiveness and flexibility in the notation, which reduce the effort needed to be able to write UML models to a minimum. UML is evolving, and its 2.0 incarnation introduces some new constructs (e.g., component, connector, port), crucial for describing system architectures, that were previously missing [11]. 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. To overcome these deficiencies, a number of approaches use existing formal languages to give some chosen UML constructs, typically statecharts and sequence diagrams, a precise semantics (e.g. [8, 9]).

This work presents a new temporal logic language, ArchiTRIO (*Architectural TRIO*), which combines a subset of the UML notation with a precise formal semantics inspired from our experiences with the TRIO and TC (TRIO-CORBA) languages [2]. To better suit industrial practices, ArchiTRIO follows a lightweight approach to the problem of formal modeling [17]; 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 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. It differs from the aforementioned formal approaches to UML in that it exploits a logic-based approach that, given a UML 2.0 composite structure diagram [14], allows one to define the dynamic properties (including possible temporal constraints) of the system components and their mutual interactions at a high abstraction level. ArchiTRIO adds expressive power to UML diagrams, rather than replacing or modifying any of them; then, a user who at first does not need full-blown ArchiTRIO can start by drawing bare UML composite structure diagrams, and only later, when the need arises for clarity and precision (especially for what concerns critical system temporal constraints), introduce ArchiTRIO-specific notation.

Given the formal nature of the language, from an ArchiTRIO model a number of developments are possible: an obvious one is to apply formal verification techniques to check the correctness of the design against high level requirements (similarly to the experience of TC [16]); in addition, to move from a high-level architectural design to a lower level closer to implementation, we envision the possibility of translating ArchiTRIO formulas into operational notations such as Statecharts or SDL diagrams [1, 12, 18]. The ultimate goal of our research, in fact, is to support the full life cycle by allowing the developer to move smoothly and safely from the high phases of requirements analysis and specification down to final implementation and verification. Thus, an operational version of architectural system design can be further refined into an executable implementation possibly exploiting a (semi)automatic code generator such as, e.g. [18].

To provide tool support to ArchiTRIO, a plugin of the TRIO-based TRIDENT integrated development platform is currently being developed. To fully support the above methodological approach, TRIDENT will allow the user to import "pure UML documents" produced through any UML tool and to augment it with the appropriate level of formality expressed in terms of ArchiTRIO.

This paper is structured as follows: Section 2 presents the ArchiTRIO approach to system development, which combines informal UML models with precise temporal logic formulas; Section 3 presents the ArchiTRIO language through a running example, and briefly hints at its formal semantics; Section 4 describes the tool being developed to support the aforementioned language and methodology; finally, Section 5 presents a selection of related works and draws some conclusions.

2 Overview of the approach

In this section, we will sketch our approach, along with a simple running example, an access control system for a building divided into areas having different security levels. Our methodological trip will start from the high level system description, written in natural language and pure UML, and will go through the architectural design, by means of the ArchiTRIO language. This section will stop right before actually presenting ArchiTRIO concepts – this will be the purpose of Section 3.

Our aim is to offer a methodology and tools that, starting from standard UML, may include a formally sound temporal logic-based technique. Ideally, our methodology follows the following route: a user would start by drawing a UML diagram (at present, we take into account class diagrams, and leave behavioral diagrams out of the picture), and then refine/specialize/complete it until (s)he obtains a complete specification/architecture, consisting of Composite Structure diagrams and their ArchiTRIO semantics, possibly augmented with exclusively ArchiTRIO concepts, on which formal verification can be carried out.

Let us now consider the example system. The Access Control System is 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 through essentially two kinds of entities: a local mechanism

based on the concept of *security gate*, and a *central control* connected to a user database.

As in current UML-based industrial practice, we start by drawing a class diagram, in which we depict the relations among these higher-level entities (see Figure 1).

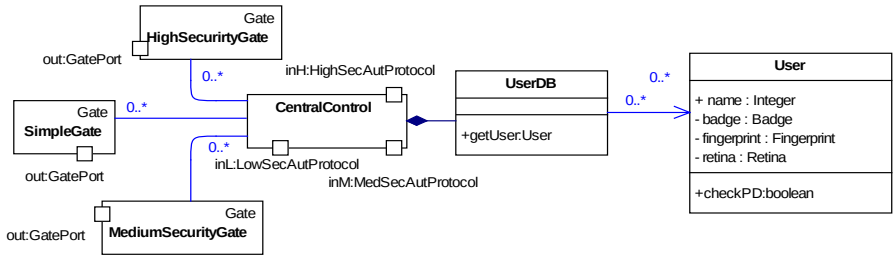

Fig. 1. Access Control System: the high-level class diagram.

The diagram in Figure 1 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 managing the local access to areas with low, medium, and high security level, respectively.

UML 2.0 introduces the useful concept of port, which is essentially an interface container. In this example ports are used to define the protocols used by the `CentralControl`, to get from, send data to, and manage gates. In Figure 1, every gate has a port of type `GatePort`, while `CentralControl` has three different ports, `LowSecAutProtocol`, `MedSecAutProtocol`, and `HighSecAutProtocol` that will be used to communicate with `SimpleGates`, `MediumSecurityGates`, and `HighSecurityGates`, respectively.

Moving in a top-down fashion, we now define the internal class structure of the gates (see Figure 2). As the reader can see, the low security gate is the simplest one, and it is depicted on the left part of the diagram. 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.

The medium security level gates are described in the central part of the diagram. A `MediumSecurityGate` is based on a more sophisticated `IdRecognizer`, a fingerprints reader (class `FingerprintsReader`), and has an entry sensor (class `EntrySensor`). In the typical usage scenario of a medium security gate, the user approaches the gate and his/her fingerprints are scanned; his/her data is then sent to the central control to be checked. If everything is ok, the gate remains open either for a short fixed time interval, or until the entry sensor actually detects the user getting in. This scenario could typically be described in UML by a sequence diagram, not reported here. 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`.

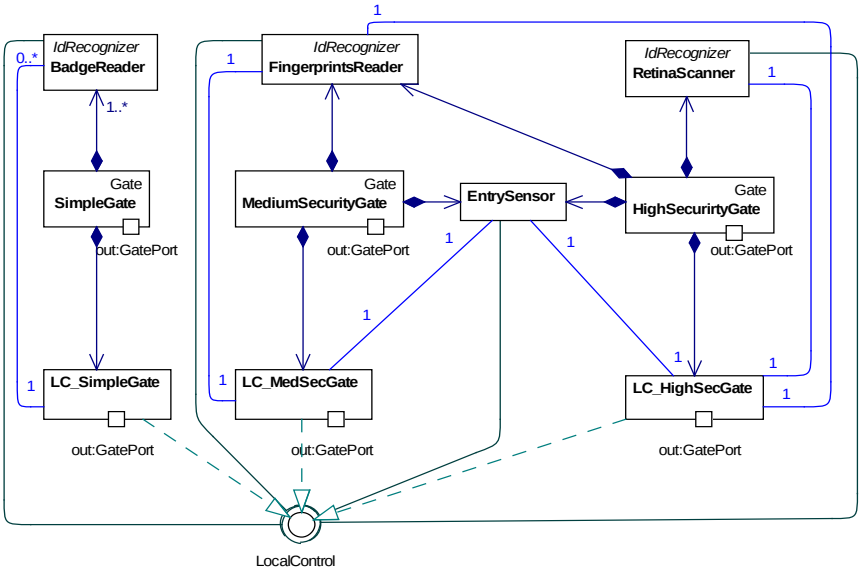

Fig. 2. Access Control System: the local-level class diagram.

The most complex type of gate is the `HighSecurityGate`, on the right side of Figure 2: it consists of two kinds of `IdRecognizers`, a `FingerprintsReader` and a `RetinaScanner`; an `EntrySensor`; and a local controller `LC_HighSecGate`. Its behavior is basically analogous to the medium security level one, but for the retina scanner: the access control has to check both the user's fingerprints and retina to open the gate.

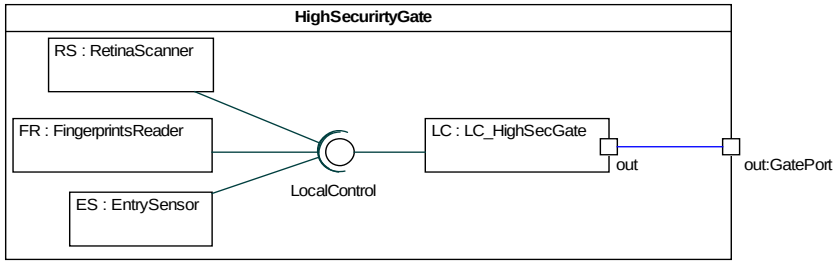

Fig. 3. Composite structure diagram of a high security gate.

To move towards the high-level system architecture, we have now to describe how instances of the classes sketched in the previous diagrams are actually interconnected and structured. As far as UML is concerned, the new composite structure diagrams, a welcome addition in version 2.0, are quite useful. Let us consider for instance a high security gate (Figure 3).

A high security gate consists of a retina scanner (`RS`), a fingerprints reader (`FR`), an entry sensor (`ES`), and a local control (`LC`). Every one 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`. Details of this aspect will be provided in the next section.

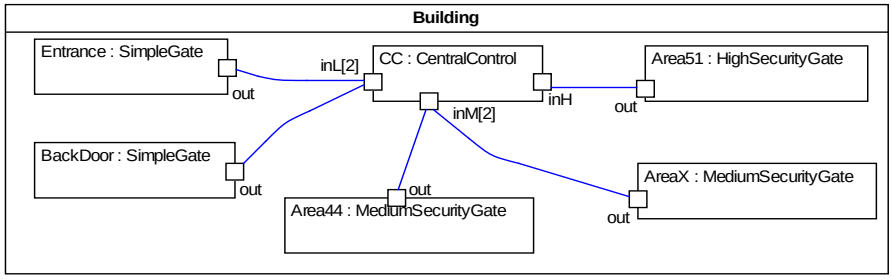

Fig. 4. The building structure: the high-level system architecture.

Last, we consider the system high-level architecture (Figure 4): our example building is made of a central control (`CC`); two low security gates (`Entrance` and `BackDoor`); two medium security areas and their corresponding gates (`AreaX` and `Area44`); finally, a 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, in our brief description above, a precise definition of timeouts management and local control behavior is nowhere to be found. More generally, we would like to be able to precisely express a critical property and possibly to verify it. In our example an unwanted behavior like the following should not be possible: Alice has clearance to enter Area51 and authenticates herself at the gate, at the same time a malicious Bob is waiting for her authentication behind a corner nearby, trying to enter into the restricted area right behind her. On one hand it is easy to correctly model the local control by using behavioral diagrams (e.g. statecharts or SDL); on the other hand however, stating and verifying general properties, such as “the entry sensor must signal a single entrance after a valid authentication, and it must occur not before k ms and not after $k+n$ ms”, is almost impossible, if one uses pure UML, even taking into account OCL. OCL *per se* has limited aim and has been designed to express static constraints like guards and pre-/post-conditions on operations without side-effects. We will consider OCL and some of its proposed variants later.

It is at this point that the designer, e.g. of a critical system, could need something more than plain UML, to seamlessly incorporate 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.

3 The ArchiTRIO language

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*.

Next to these UML elements, however, ArchiTRIO includes also concepts derived from temporal logics, which allow users to precisely define the behavior of a system modeled with ArchiTRIO. In fact, every UML element featured in ArchiTRIO is given a formal semantics in terms of the temporal logic (HOT, Higher-Order TRIO [5]) on which ArchiTRIO is founded. This, in turn, allows one to attach a precise meaning to the formulas describing the dynamics of the components (taken separately or as a whole) of the system being modeled.

Let us now illustrate some of the most significant syntactic features of ArchiTRIO through the example of system shown in Section 2. Section 3.1 will briefly hint at the semantics of some of the elements shown here, without pretense of being exhaustive.

The graphical representation of those concepts that are common to both ArchiTRIO and UML is the same as in UML. However, every ArchiTRIO element (UML- and logic-derived ones alike) is also given a textual representation detailing its ArchiTRIO-specific features. For example, class `LC_HighSecGate` introduced in Figure 2 provides interface `LocalControl` and has a port of type `GatePort`; interface `LocalControl` defines two operations, `incomingData` and `personEntered`. The corresponding textual declaration of Figure 5 defines that, 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. Notice that the `temporal domain` clause defines that temporal variables range over real values (that is, time is dense).

```
class LC_HighSecGate
temporal domain: real;

provides LocalControl ...
ports:
  out : GatePort; ...
```

```
items:
  TI inGate : GateId;
  TD lastUser : User;
  state gate_open;
constructors:
  LC_HighSecGate(GateId g) : inGate = g;
 axioms: ...
end
```

Fig. 5. Sketch of the textual declaration of class `LC_HighSecGate`.

As we will show later through some examples, 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` are formulas that 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` (see Figure 7 for its signature) on port `out`, passing `pd` and the value corresponding to item `inGate` as parameters.

```
vars: iD  : incomingData;
      sPD : sendPersData;
      pd  : PersonalData;
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` (whithin 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 [2] (it stands for *within the future*). Finally, `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.

As another example of ArchiTRIO formula, let us focus on axiom `gate_open_Def` below, which defines precisely when the controller leaves the gate open, thus allowing a user to enter. `gate_open_Def` states that, in the current instant, the gate is open if and only if there is another instant, within the past `Topen` time units (where `Topen` is a system-dependent constant), in which the controller received an invocation `oG` of operation `openGate` from port `out`, and no invocation of operation `personEntered` has been received since (see [2] for the precise definition of temporal operators `Since` and `WithinP`).

```
vars: pE : personEntered;
      oG : openGate;
gate_open_Def:
  gate_open <->
    Since(not ex pE(pE.inv_rec), ex out.oG(out.oG.inv_rec)) &
    WithinP(ex out.oG(out.oG.inv_rec), Topen);
```

Notice that as a consequence of axiom `gate_open_Def` the gate cannot stay open longer than `Topen` time units if the `openGate` command is not refreshed (i.e. received anew from the central controller); in fact if, after `Topen` time units from the last `openGate`, no person has yet entered (i.e. a `personEntered` command has not been received), subformula `WithinP(ex out.oG(out.oG.inv_rec), Topen)` does not hold any more, thus `gate_open` becomes false (i.e. the gate closes).

Let us now focus on the concept of *port* in ArchiTRIO. Syntactically, a port is just a collection of provided and required interfaces. From a semantic point of view, instead, a port 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). Then, an ArchiTRIO port can contain axioms defining the corresponding protocol in terms of the involved operation invocations .

Consider, for example, port `HighSecAutProtocol` mentioned in Figure 1. It provides interface `AccessControl`, and requires one instance of interface `FromAccessControl` (the details of the operations defined by the two interfaces can be found in Figure 7).

```
interface AccessControl
operations:
 User sendPersData(in PersonalData rawData,
                   in GateId gate)
      raises UserNonExistentException;
 enterPerson(in User user,
             in GateId gate)
      raises UserNonExistentException;
end
```

```
interface FromAccessControl
operations:
 openGate();
end
```

Fig. 6. Declaration of interfaces `AccessControl` and `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 authenciation to be successful (i.e. for the controller to allow the user to enter by opening the gate through an `openGate` command).

Figure 8 shows axiom `openGate_SC` of port `HighSecAutProtocol` defining a sufficient condition for the `openGate` command to be sent to the gate through interface `FromAccessControl`. Formula `openGate_SC` states that if there are two invocations (`sPD1` and `sPD2`) of operation `sendPersData` of interface `AccessControl` (`ac`) that are completed successfully within a maximum delay of `Tprot` time units one from the other, and such that

- the `gate` input parameter is the same for both and
- the `rawData` input parameter has type `Fingerprints` for one of them and `Retina` for the other,

then operation `openGate` is invoked on interface `FromAccessControl` (`fac`) no later than `T` time units after the instant in which the second invocation (represented in the formula by `sPD1`) ended.

```
vars: sPD1, sPD2 : sendPersData;
      oG : openGate;
openGate_SC:
  ac.sPD1.reply(u) & ac.sPD1.rawData = rd1 & ac.sPD1.gate = g &
  ex ac.sPD2(WithinP(ac.sPD2.reply(u) &
                     ac.sPD2.rawData = rd2 & ac.sPD2.gate = g,
                  Tprot)) &
  (type(rd1, Fingerprints) -> type(rd2, Retina)) &
  (type(rd2, Fingerprints) -> type(rd1, Retina))
  ->
  WithinF(ex fac.oG(fac.oG.invoke), T);
  ...
end
```

Fig. 7. Axiom `openGate_SC` of port `HighSecAutProtocol`.

Finally, the textual declaration of a composite ArchiTRIO class defines the elements composing each instance of the class, and how they are connected with each other (e.g. which part provides the interface required by another part, and so on).

3.1 ArchiTRIO semantics (hints)

From a semantic point of view, ArchiTRIO is founded on a higher-order temporal logic, Higher-Order TRIO (HOT for short [5]). The choice of a higher-order logic was dictated by the need to allow an easy representation of mechanisms such as the passing of parameters of complex types (to be precise, of parameters that can be ArchiTRIO/UML objects).

In HOT terms, a class is a *type*. An *object* in HOT is an instance of a class, that is a *value* of a type. ArchiTRIO is based on the same concepts: an ArchiTRIO class is a HOT class, so it defines a type; an ArchiTRIO object is an instance of the class.

An ArchiTRIO operation also corresponds to a HOT class. All operations share a core group of features (built-in items and behavior), which is modeled by a HOT class `Operation`. This class introduces the logic items modeling the relevant features of an operation invocation (e.g. the `invoke`, `inv_rec` and `reply` events presented above), and the axioms defining the behavior that is common to all invocations.

A specific operation (e.g. `incomingData`) is also defined as a class. For example, a class `incomingData` defines the semantics for the corresponding operation. Class `incomingData` is a subtype of class `Operation`: in short, if a class `S` is a subtype of a class `P` then `S` inherits all the elements of `P`, and all axioms of `P` are still valid in `S`. Every instance `i` of class `incomingData` (i.e. every value of type `incomingData`) is an invocation of the corresponding operation.

An ArchiTRIO interface is just a HOT class exporting operations. A class providing an interface, from a semantic point of view, is a subtype of that interface. An ArchiTRIO class requiring an interface `I` is a HOT generic (i.e. parametric) class with respect to a parameter of type `I`. A connection between a provided and a required interface (like the one between modules `LC` and `RS` of Figure 3, for example) corresponds, semantically, to a parameter instantiation (in the case of Figure 3, the parameter of type `LocalControl` of module `RS` is instantiated with object `LC`).

Finally, since a port is a collection of provided and required interfaces (plus a set of axioms), an ArchiTRIO class that has a port of type `P`, which provides interfaces `PI1`...`PIn` and requires interfaces `RI1`...`RIm`, also provides and requires the same interfaces. In addition, a class that has a port `P` includes the axioms of `P`.

4 Tool support

Our experience of several decades with the TRIO language brought the construction of a long series of prototypical tools, every one with a different slight variant of the language, and different verification or editing capabilities. From this situation the decision of a couple of years ago, to build up an industrial-strength integrated tool for supporting our methodologies and languages.

TRIDENT (short for *TRio Integrated Development EnvironmeNT*) is a tool for the development and analysis of time-critical systems based on the TRIO formal language. TRIDENT is implemented on the Eclipse platform [3], and is currently being developed jointly by Politecnico di Milano and CEFRIEL.

As typical with Eclipse-based tools, TRIDENT is plugin-based, so it is by itself an open and evolving product. The environment is still in a prototypical stage, so many of the intended features are still incomplete.

Some of the most notable present features of the tool are the ability of editing complex TRIO specifications and *histories* (i.e. execution traces that may be used as test cases), and check their mutual compatibility.

More recently, a plugin for supporting model-checking of TRIO specifications has been implemented [13]. This plugin, also called TRIO-PROMELA, is based on the well-known model-checker SPIN, and uses a novel translation technique based on alternating automata. We intend to use this very same technique for model-checking modular and mixed logic/operational specifications (e.g. having components written in some automata-based notation, say for example SDL), but this feature is not yet implemented.

As far as ArchiTRIO is concerned, currently there is an advanced-stage prototypical plugin, which supports class and composite structure diagrams editing, and some of the basic ArchiTRIO characteristics. In addition, a prototype plugin capable of partially transforming XMI files into TRIDENT objects has been developed and should be available in the TRIDENT distribution in a short time.

5 Related works and conclusions

In this paper we presented a formal language, ArchiTRIO, suitable for describing system architectures. It combines a subset of the UML 2.0 graphical notation with a higher-order temporal logic, which allows users to precisely express both the structural (static) and the behavioral properties of the modeled system. ArchiTRIO is designed to let users draw models in a subset of the usual UML notation (to be precise, using class diagrams and the new composite structure diagrams) and then, if and when necessary, add precise details about the behavior of the target system using a temporal logic-based formalism.

ArchiTRIO combines UML and formal languages to provide a powerful means to model system architecture and, as a consequence, is related to a number of works that have appeared in the literature in recent years. Let us briefly analyze how our work on ArchiTRIO differs from previous ones.

[11] shows how UML pre-2.0, if taken by itself, lacks concepts that are necessary for modeling system architectures, and proceeds to introduce profiles for a pair of Architecture Description Languages (ADLs) to cover for these deficiencies. The approach of [11] presumes that users will then use these profiles, and the ADL-specific concepts they define, to model architectures. The ArchiTRIO approach, instead, does not introduce any new graphical notation to UML 2.0: the user who does not need the full expressiveness of ArchiTRIO can still use the plain UML notation and ignore the underlying logic altogether; the user in need of rigor and precision, on

the other hand, can seamlessly introduce formal definitions of the behavior of the system in his/her model, without altering the original UML description.

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 (see, [8] or [10], and many others not listed here for the sake of brevity), or, alternatively, into an ad-hoc model [9]. With ArchiTRIO there is no translation into any other language; on the contrary, it is a formal language *integrated* into the UML 2.0 notation, which allows one to precisely describe both the structure and the behavior of a system, of its components and their interactions, with particular attention to their temporal constraints.

Indeed, UML already has an associated logic language, the Object Constraint Language (OCL), for which temporal extensions have been proposed [4]. However, OCL, and RT-OCL in particular, is a language with limited scope, as its intended use is mostly for expressing constraints on behavioral diagrams such as statecharts. On the contrary, the ArchiTRIO approach is a comprehensive one, which aims at supporting the whole system specification and design process by modeling all aspects of a system architecture, both structural and behavioral.

Finally, [7] 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 [7], and associated notations, as a target model, to be obtained through a suitable method from an ArchiTRIO design to perform subsequent verification.

This work opens the way to a variety of future developments. First and foremost, we will complete the development of the tool sketched in Section 4, which we plan to release for free use by both academic and industrial communities.

Secondly, we will investigate verification techniques (to be supported by the tool-set mentioned above) to complement the modeling features presented in this paper. In this regard, the semantics of ArchiTRIO in terms of HOT suggests a fairly straightforward 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 [6]. Other approaches will also be explored, for example translating ArchiTRIO classes into automata-based formalisms (like, e.g., those used in [7]) to exploit model checking techniques.

Finally, we plan on developing a method that allows one to move from the purely logic notation of ArchiTRIO to an operational formalism closer to implementation such as SDL [12] (as mentioned in Section 4, techniques to translate TRIO temporal operators into Promela communicating processes have already been explored in [13], and many concepts of Promela can also be found in SDL). This would open up the possibility of using existing tools (e.g. [18]) to perform automatic generation of code that complies with the properties and the behavior precisely defined by an ArchiTRIO model (and, in particular, by the axioms contained in its classes).

References

1. Sergio Cigoli, Philippe Leblanc, Salvatore Malaponti, Dino Mandrioli, Marco Mazzucchelli, Angelo Morzenti, Paola Spoletini: An Experiment in Applying UML2.0 to the Development of an Industrial Critical Application, Proceedings of the UML'03 workshop on Critical Systems Development with UML, San Francisco, CA, October 21 2003.
2. Coen-Porisini A., Pradella M., Rossi M., Mandrioli D., A Formal Approach for Designing CORBA based Applications, ACM TOSEM, vol. 12, n. 2 (2003) 107–151
3. Eclipse Foundation, http://www.eclipse.org
4. Flake, S., Mueller, W. Formal Semantics of Static and Temporal State-Oriented OCL Constraints, Software and Systems Modeling, vol. 2, n. 3, Springer (2003) 164–186.
5. Furia, C. A., Mandrioli, D., Morzenti, A., Pradella, M., Rossi, M., San Pietro, P., Higher-Order TRIO, Technical Report 2004.28, Dipartimento di Elettronica ed Informazione, Politecnico di Milano (2004).
6. Gargantini, A., Morzenti, A., Automated Deductive Requirements Analysis of Critical Systems, ACM TOSEM, vol. 3, no. 3, (2001) 225–307.
7. Giese, H., Tichy, M., Burmester, S., Flake, S., Towards the compositional verification of real-time UML designs, Proc. of ESEC/FSE 2003, Helsinki (2003) 38–47
8. Lavazza, L., Quaroni, G. Venturelli, M., Combining UML and formal notations for modelling real-time systems, Proc. of ESEC/FSE 2001, Vienna (2001) 196–206
9. Li, X., Liu, Z., Jifeng, H., A Formal Semantics of UML Sequence Diagram, Proceedings of the 2004 Australian Software Engineering Conference, (2004) 168–177
10. McUmber, W. E., Cheng, B. H. C., A general framework for formalizing UML with formal languages, Proceedings of the 23rd ICSE, (2001) 433–442
11. Medvidovic, N., Rosenblum, D. S., Redmiles, D. F., Robbins, J. E. Modeling Software Architectures in the Unified Modeling Language, ACM TOSEM, vol. 11, no. 1, (2002) 2–57
12. Mitschele-Theil, A., System Engineering with SDL – Developing Peerformance-Critical Communication Systems, John Wiley (2001)
13. Morzenti, A., Pradella, M., San Pietro, P., Spoletini, P., Model-checking TRIO specifications in SPIN, 12th Int. FM Symposium, LNCS 2805, Pisa (2003) 542–561
14. Object Management Group, UML 2.0 Superstructre Specification, Technical Report, OMG, ptc/03-08-02 (2003).
15. Object Management Group, UML 2.0 OCL Specification, Technical Report, OMG, ptc/03-10-14 (2003).
16. Rossi M., Mandrioli D., A Formal Approach for Modeling and Verification of RTCORBA-based Applications, ISSTA, Boston (2004) 263–273
17. Saiedian, H., Bowen, J. P., Butler, R. W., Dill, D. L., Glass, R. L., Gries, D., Hall, A., Hinchey, M. G., Holloway, C. M., Jackson, D., Jones, C. B., Luts, M. J., Parna, D. L., Rushby, J., Wing, J., Zave, P. An Invitation to Formal Methods. IEEE Computer, vol. 29, no. 4, (1996) 16–30
18. Telelogic Tau Generation2 Tools, http://www.telelogic.com/products/tau/tg2.cfm