Model, Notation, and Tools for Verification of Protocol-Based Components Assembly

Pascal Rapicault[1,2], Jean-Paul Rigault[1], and Luc Bourlier[1]

[1] Rainbow and Sports Projects, I3S Laboratory,
University of Nice Sophia Antipolis and CNRS (UMR 6070),
F-06902 Sophia Antipolis Cedex, France
[2] Object Technology International, Inc. (OTI),
`firstname.lastname@essi.fr`

Abstract. Assembly of blackbox components is made difficult by the lack of precise information on the way components interact. What is needed is a behavioral model of the component, at the input and output interface levels. This paper introduces the notion of behavioral points of view and an associated graphical notation, SyncClass, to represent such a model. The underlying semantics of SyncClass makes it possible to automatically verify component assembly, either for individual components or for a whole system.

1 Introduction

We are currently working on the definition and the implementation of a CASE environment, named Co2[1], which provides notations and tools to specify, develop, and use components.

This communication focuses on the notations and tools devoted to the *user* of components, that is the person who assembles existing components to build an application. More specifically it addresses the problem of verifying components assembly, either for individual components or for a whole system.

The ideal way of assembling components should rely on the sole knowledge of the component interfaces (*blackbox reus*e [19]). However the reality is different, and the developers often require knowledge about component internals (*glassbox reuse* [19]). Indeed the user is generally provided with a static description of the interface (a simple list of operations) whereas information about the valid sequences of operation calls would be needed. The latter information is what we call the component *protocol of use*.

The situation is even worse when components are organized into a component framework [19]. In order to respect a given component protocol, the user cannot just consider his/her own calls to the component but also the calls originating from other components and targeted to the component under study. Thus the user has to guess the part of the protocol to which he/she must comply.

[1] Co2 stands for *Components and Composition*

© Springer-Verlag Berlin Heidelberg 2002

This reveals that to use a component framework the knowledge of individual component protocols is not sufficient and that describing the messages exchanged among components is also required. Thus we need to accompany the input interface with the description of an *output interface*. In the same way as the input interface comes with a protocol of use, the output interface should sport a *protocol of composition*.

In order to describe these protocols without showing too much of the inner behavior, we propose a protocol model and a lightweight notation that can be used from analysis to reuse time. This graphical notation, called SyncClass, and the associated tools to verify the assembly of components constitute the major topics of this paper. SyncClass diagrams are used to represent both the input and the output protocols of components; they are to be embedded into the components. Since our main objective is to make the verification automatic, SyncClass has to rely on formal semantics. Because of our cultural background as well as for the many tools it proposes, we chose the so-called Synchronous Model [10,4].

This paper is organized as follows: section 2 describes existing techniques used to represent protocols, including the synchronous model itself. Section 3 describes the model and the language of SyncClass, and an example of this notation is used in section 4. Section 5 briefly presents the relationship between the synchronous model and our component model, and section 6 details static and dynamic verification.

2 Models for Components Protocols

In the following, we only mention the most common techniques for representing protocols. To get further information on emerging works in the domain (and even broader) one may refer to the workshop on "Specification and Verification of Component-based Systems" at OOPSLA'01 [1].

Automata-Based Models. A popular choice for representing protocols (of any kind) is finite state machines and their derivatives (e.g., regular expressions, path expressions [7]...). However, it is not suitable to describe complex behavior because automata lack readability when the number of states and/or transitions becomes large.

The work on "regular types" [15] follows the same line. It aims at checking type substitutability [12] and uses automata to specify the protocol of a type. However, regular types do not describe the effect of method calls; thus they are not appropriate to express interactions between components.

Architectural Description Languages. ADLs generally represent the architecture of a system as a combination of two kinds of entities: components and glue [2]. A component is a unit of computation, a connector describes a connection protocol. This model is flexible in that it permits to reuse components or connectors independently. However, even if the component/connector distinction can be twisted to represent components and frameworks [18], we do not think that ADLs are appropriate for blackbox components. Indeed, ADL components have no context dependencies and thus they need connectors to establish links with other components. In the blackbox reuse model, a component is aware of the components to which it connects.

UML-Based Models. UML component diagrams are purely structural and do not depict dynamic information like operation calls and protocols. Sequence diagrams, collaboration diagrams and statecharts specify the dynamic aspects of a system. However, the first two ones are meant to represent only one execution of the system and many diagrams are necessary to cover (even partially) the all the possibilities; the third ones are used to detail the inner states of objects and can also describe protocols [16, p. 2-175]. However, because Statecharts semantics is not satisfactory to build a proof system, most works interested in proof systems have to introduce their own ad hoc formalism[2].

Synchronous Model. In order to represent protocols, we chose the synchronous model [9,4]. It is a specialization of the theory of automata. The reasons for this choice are multiple: the model relies on formal semantics allowing automated proofs [6], and comes with a complete development platform which provides model checkers and simulators. There exists several textual or graphical languages supporting it (Esterel [5], Lustre [10], Argos [13], SyncCharts [3]), and there is a research and commercial support for the tools.

To the best of our knowledge, synchronous languages have never been applied to software component specification, despite their proof capabilities. The description of the synchronous model is out of the scope of this paper; we shall just introduce the concepts we use in section 5.

3 A Model and a Language for the Protocol

3.1 What Do We Mean by a Component?

Our definition of a component includes the following characterisitics: it is a unit of composition, it is encapsulated (it promotes black box reuse), it provides an input interface (a set of operations described by their signatures), it provides an output interface (a set of called operations together with the external components which sport them). These four first items are closed to Szyperski's component definition where "a component is a unit of composition with contractually specified interfaces and explicit context dependencies" [19].

The following three characteristics are related to the execution model that we consider. A component constitutes an unit of execution, acting as a reactive entity, it contains one thread of execution (there might be several but method reentry is not authorized), its methods run to completion. This may appear as

[2] To get convinced, one just has to look at the session(s) on Statecharts semantics in the yearly UML Conference.

limitations. Indeed, our work cannot presently support the general concurrency issues. Thus we restrict to a pure sequential use of components, an approach which is still useful in many applications (see for instance [14]).

Finally a component embeds its own documentation. This last characteristics is essential for our work. We claim that a component is not only a piece of code, but that it must contain the documentation for using it. Embedding the documentation constitutes the last step of component development, just before final releasing.

In our view, this documentation must be usable by automatic verification tools and still be readable by human beings. Thus it should externally describe the protocol of the component, without displaying internal details. Moreover, writing this documentation should not constitute extra work ; it should be derivable from the analysis and design models.

3.2 Behavioral Points of View

We introduce the notion of behavioral points of view to describe the protocol of a component from a given perspective. For each component, we identify two such perspectives: the *client point of view* expresses how the component should be used, the *composition points of view* describe how the component uses other components.

To illustrate these points of view, we consider the hypothetical system made of three components presented on figure 1.

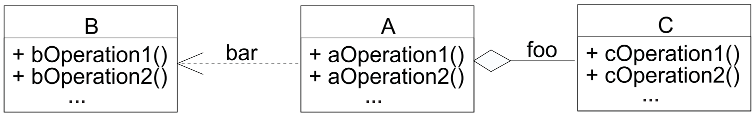

Fig. 1. A system made of three components.

The Client Point of View. The client point of view describes the protocol of use of a component. It is the dynamic counterpart of the static interface: it specifies in which order the component operations can be called. Thanks to this view, a user knows when an operation can be called or when it must not be.

There is one client point of view per component. Thus, in the three components example, there are three client points of view. For instance the client view of `A` describes the valid order of calls to `aOperation1`, `aOperation2`...

Composition Points of View. Whereas the client point of view focuses on the protocol of one component, composition points of view abstract the protocol with which one component uses an other one. Thus, a given component has one composition point of view for each other component to which it is associated.

The composition point of view of component A with respect to component B specifies both the operations of B used by A and the order of the corresponding calls. Thus it constitutes a part of the dynamic specification of the output interface.

For the given example, we have one composition point of view describing how A uses B. It gives the protocol used by A to communicate with B. It represents a relation order over B's operations. A second composition point of view does the same for components A and C. Two more composition points of view are needed to reflect the reverse associations (i.e., B w.r.t. A, and C w.r.t. A).

Relation between Client and Composition Points of View. These two kinds of points of view are not independent. Indeed, the composition point of view of A with respect to B depends on the list of operations called by A and on the order of the corresponding calls. Since these B operations are called from A, it also depends on the order with which A operations are called (that is the client point of view of A).

The corresponding descriptions should not be duplicated; instead, they should be consolidated into a unique (graphical) representation. This representation extends the client point of view of a component A by adding to any operation of A the operations it calls on other components. This synthetic representation, that we call the *component protocol*, not only describes the input interface behavior but also completely specify the dynamics of the output interface. As a consequence, it makes it possible to know the overall communication protocol among components.

The composition points of view are just restrictions of the component protocol. In the example, the composition point of view of A w.r.t. B corresponds to the set of operations that are called by component A and that are defined in B's protocol. The composition points of view can be automatically built from the component protocol. It is important to note that, dualy, the composition point of view of A w.r.t. B is also a restriction of the client point of view of B.

Of course, the number of views increases quickly. However, in a real system, all the views are not required for every component. The component developers will only provide the views for those components for which there is an interest in precise documentation or verification.

Those views are provided with the component they describe. In our three components example, the client point of view of A comes with component A, as does the composition point of view of A w.r.t. B and the one of A w.r.t. C.

3.3 SyncClass: A Graphical Language for Behavioral Points of View

In order to represent those views, we chose to introduce a graphical language, close to what is already known by designers, so it can easily be used and understood. It is named SyncClass[3]. SyncClass inherits from SyncCharts [3], a synchronous extension of Harel's StateCharts [11].

[3] We write SyncClass to describe the model, and syncClass to describe an instance of it.

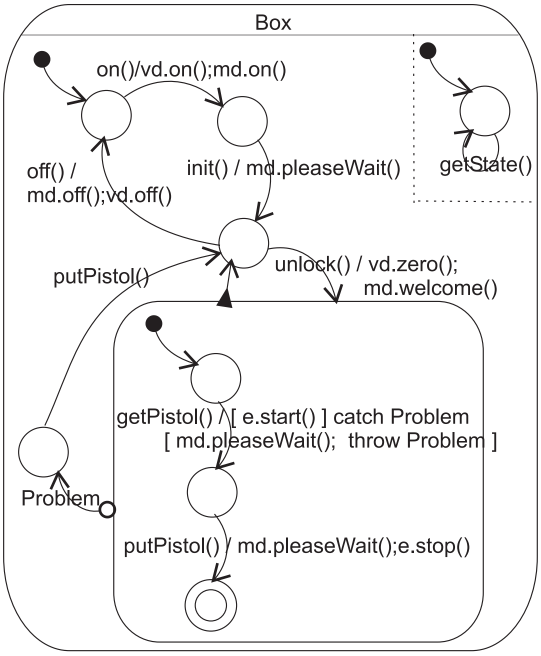

Fig. 2. Example of a SyncClass.

Graphically SyncClass is an automata-based representation, made of states and transitions, as represented on figure 2. Macrostates make it possible to encapsulate a subautomaton and provide a hierarchical decomposition feature. Note that a transition cannot cross the border of a macrostate. The concurrency, expressed in SyncClass by dashed lines splitting a macrostate into several parts, does not represent true runtime concurrency. It indicates that the operations contained in the subparts are not depending on each other. Such a macrostate is said to be *composite*. The notation defines four kinds of transitions. An *initial transition* starts with a black circle; its target is the initial state. A *regular transition* is represented by a simple arrow and corresponds to an operation call. An *exception transition* is represented by an arrow starting with a circle and corresponds to an exception raised by the component itself; it means that the component protocol enters some exception mode and it will be the responsibility of the user to catch the exception and to ensure a valid continuation. Finally a *normal termination* transition starts with a triangle; it is automatically triggered when a macrostate reaches a final state (a final state is represented by a double circle).

Only exception transitions and regular transitions can and must be labeled. For regular transitions, the label is divided into two parts separated by a slash. The left part represents the signature of the operation that will *trigger* the transition, and the right part (also called the *action* part) is an abstraction of the operation behavior: it uses a small language to describe the control flow of the operation designated in the left part. This language offers optionality[4] (indicated by a `#` prefix), iteration (a list of instructions enclosed within braces), exception catching (`[...] catch` *exception* `[...]`) and raising (operator `throw`), and operation call (using the dot notation to specify the target component). Semicolons separate instructions and indicate sequentiality. The label of exception transi-

[4] The condition expression itself is not represented: the inner state is not shown and thus encapsulation is not broken.

tions is only made of a left part representing the exception that causes this transition to be triggered.

Thus a syncClass represents the overall protocol of a component: the left part of transitions corresponds to the client point of view whereas the right part corresponds to the composition ones.

The operational semantics of SyncClass is the following: the initialization of a syncClass is done by activating all top level macrostates, which is equivalent to triggering their initial transition, that is to activate their initial state. This is done recursively. After the initialization has been done and all along the syncClass "execution", several states may be active. Regular and exception transitions are triggered when their origin state is active and the corresponding operation [resp. exception] is called [resp. raised]. When a transition leaving an active macrostate[5] is triggered, the macrostate is no longer active. When an active simple macrostate reaches its final state, its normal termination transition (if any) is triggered. The end of a composite macrostate is reached when every subpart reaches its end. When an unexpected trigger occurs, it is considered as an error.

When a transition is triggered, the associated action part is executed which causes its messages to be sent. If an instruction is marked as optional, this indicates the possibility of several exclusive execution paths.

4 An Example: A Petrol Pump

This section gives an example of syncClass to represent the behavioral points of view of a petrol pump system. The petrol pump is made of five components: a box, a pistol, an engine, a volume display, and a message display. The class diagram on figure 3 describes the associations among the components. The role names on the associations will be used to denote target components when sending messages.

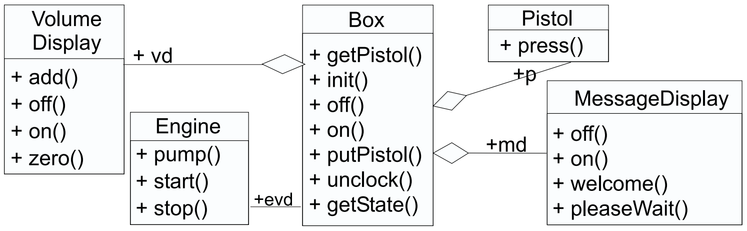

Fig. 3. Class diagram of the petrol pump.

[5] A macrostate is said to be active if it contains an active (macro)state.

In fact, figure 2 presents the component protocol of the Box. The Box is considered as the "main" component since it switches the others on. The client point of view indicates that operation on needs to be called first. One can also see that operation on of the box calls operations on of the volume and message displays. Only once the on operation of the box has been executed, can the operation init be called. The action part on the getPistol transition, shows an example of exception catching.

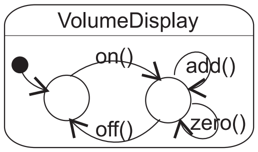

Fig. 4. Component protocol of the volume display.

The component protocol of the volume display presents the sequence of authorized operations (figure 4). The absence of action on the transitions in the protocol indicates that this component can be used alone.

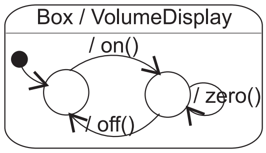

Fig. 5. Composition view of the box with respect to the volume display component.

The composition point of view of the box w.r.t. the volume display (figure 5) specifies the order in which the operations of VolumeDisplay are called by the Box. It is a restriction of the protocol of Box. By comparing this composition point of view to the client point of view of VolumeDisplay (figure 4), one can deduce that VolumeDisplay is probably used by something else than Box. Indeed the box only uses a subset of VolumeDisplay operations (operation add is not called). Facing such a situation, the user needs to determine whether the missing operations are called by other components or whether he/she is required to call them.

5 SyncClass and the Synchronous Model

As indicated in section 2, we use the synchronous model as a formal basis. This model allows us to use its model checkers, its simulators, and its languages either textual like Esterel [5] or its graphical equivalent, the SyncCharts [3]. We could not use the usual synchronous notation since our model differs from the synchronous one on the following points: an operation call cannot be considered as instantaneous (we need to distinguish beginning and end of methods); we do not need a general broadcast ; we have to make our SyncClass model deterministic, even though the application is not. Thus we have an automatic translation of SyncClass to SyncCharts. However we are short of space to describe the translations.

6 Verification of Components Assembly

6.1 Static Verification

The mapping to a semantically sound model allows to use model checkers for static verification. We support two kinds of static verification. One checks the compatibility of one component with respect to another one and the other checks a complete system of components.

For both kinds of verification we use model checking [6,9]. Indeed, it provides automatic tools to prove properties on automata. These properties are classically represented by observers, which are also synchronous (SyncCharts), composed in parallel with the system to prove. The proof relies on the exploration of the overall state space (system and observers).

Checking the Compatibility of One Component with an Other One. Here the objective is to check whether a given component, say `A`, correctly uses an other component, say `B`, with which it is associated. In our model this means to verify the compatibility between the composition view of `A` w.r.t. `B` and the client view of `B`. In the example of the petrol pump, one could check whether the box correctly uses the volume display. To do so, we consider the composition point of view of the box w.r.t. the volume display (figure 5), and the client point of view of the volume display (figure 4).

We associate an observer to each operation. Its role is to check that each call is correctly received. For this the operation is instrumented, so that it sends an acknowledge signal (`ack`) when it starts. Owing to synchronous signals broadcast, the observer has just to check that the call and the acknowledge match each other. Figure 6 presents the observer for operation `on` of component `Box`: if the operation is called and no acknowledge is received, an error signal is emitted (`KO`). The observers as well as the needed instrumentation are automatically generated.

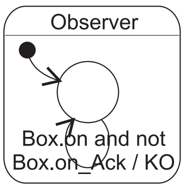

Fig. 6. SyncCharts of the observer used for static verifications.

Then the client point of view of the box and one of the composition point of view are composed in parallel together with the observers for all the operations involved in the composition point of view. The result is fed into the model checker which is then asked to find whether the `KO` message can possibly be emitted.

System Checking. The goal of this second kind of static verification is to check whether an assembly of components can cooperate so that each component respects the others protocol. For example, one can check whether the five components constituting the petrol pump can work together.

This verification requires a specification of the (overall) protocols of all the components. They contain all the information required: the protocol, and the interaction of a component with the others. The verification also requires an observer which listens to all operation calls and check that they are all correctly received. This observer is thus a parallel composition of simple observers such as the one in figure 6.

It is necessary to constrain the exploration order of the model checker, so that call sequences that are known to be erroneous are not considered–otherwise the checker will automatically fail since unexpected operation will be called. Such sequences can be automatically computed from component protocols. If an error occurs (`KO` gets emitted) the model checker is able to display a sequence of operation calls causing the problem. If no error appears, then it is sure that the system under test will behave correctly provided that it is used correctly.

Instead of being computed from component protocols, the input sequences can be produced by the user. An interesting case is when they are derived from the user glue code. The check will then indicate whether or not the system fails when used by the glue. This can be seen as a kind of integration test.

Another benefit is to verify component replacement. If we substitute a component with another one in a validated system, the same verification techniques apply to check whether this substitution is correct.

Both kinds of static verification have been implemented and the petrol pump has been checked. The verification of the whole system took approximately 26 seconds on a Pentium III 1Ghz.

6.2 Run-Time Verification

The goal of run-time verification is to detect operation calls which do not respect component protocols. For this, we use the syncClass embedded within the component. Each operation call is dynamically checked against this syncClass.

To avoid two versions of each component, one for debug and one for the final release, we have to trap operation calls: we use meta-programming techniques when applicable. We successfully implemented dynamic verification [17], first for simple classes using Javassist [8], then for JavaBeans components using their built-in meta-facilities. For these examples, the documentation has been embedded into the component most suitable format. For JavaBeans, we enhanced the meta information associated with the beans so that it may contain the protocol. For the classes, we added the documentation in the user attribute zone of the classfile.

7 Conclusion

This paper introduces a graphical notation, named SyncClass, to describe component protocols of use. This protocol is composed of two kinds of behavioral points of view: the client view describes the valid sequences of operations that can be applied to the component; the composition view specifies the way one component uses another one. In this approach, the documentation is fully integrated into each component and SyncClass constitute an essential part of it.

SyncClass relies on the Synchronous Model, which permits to take advantage of synchronous platforms and tools. In particular we show how model checkers can use the embedded syncClasses to automatically verify components assembly, at the individual component level as well as at system level. The same documentation may also be used at run-time to dynamically check proper component usage.

The Co2 environment, the context of this work, provides tools to ensure the consistency between component implementation and their protocol documentation. This is even more true when the syncClasses can be derived from the design documents. Most of the Co2 tools (SyncClas editor and generator, code generator, interface with model checker...) have been implemented, only a test generator is missing. Co2 addresses not only the component developer's task but also the component user's one. The embedded documentation (especially SyncClass) bridges the gap between the two activities.

Some features of SyncClass were not presented here. The most important one is callback specification, which is part of the composite view. Another is the possibility to handle a restricted form of concurrency instead of assuming a pure sequential usage of components.

Other features would be desirable, such as supporting a general model of concurrency. An other important issue is related to the graphical representation of component interfaces (both input and output) and component interconnections. A third improvement would be to handle component inheritance and substitutability. These features will constitute the topic of future work.

In a near future we shall also experiment our notation and tools on bigger examples, such as the framework Blocks [14] in order to demonstrate the applicability of our approach to real life systems.

References

1. Specification and verification of component-based systems workshop at OOPSLA 2001. Workshop ISU TR 01-09a, Department of Computer Science, 2001.
2. Robert Allen and David Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*, 6(3):213–249, July 1997.
3. Charles André. Representation and analysis of reactive behaviors : a synchronous approach. In *CESA*, pages 19–29, july 1996.
4. Gérard Berry. *Proof, Language and Interaction: Essays in Honour of Robin Milner*, chapter The foundations of Esterel. MIT Press, 2000.
5. Gérard Berry and Georges Gonthier. The ESTEREL synchronous programming language: design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.
6. Amard Bouali. Xeve: An esterel verification environment (version v1.3). Technical Report RT-214, INRIA, October 1997.
7. R. Campbell and N. Habermann. The specification of process synchronization by path expressions. In *Proc. Int. Symp. on Operating Systems*, LNCS 16, pages 89–102. Springer-Verlag, 1974.
8. Shigeru Chiba. Load-time structural reflection in Java. In *ECOOP 2000, Sophia Antipolis and Cannes, France*, LNCS 1850, pages 313–336. Springer-Verlag.
9. Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts, 1999.
10. Nicolas Halbwachs. *Synchronous Programming of Reactive Systems*. Kluwer Academic, 1993.
11. David Harel. Statecharts: A visual formalism for complex system. *Science of Computer Programming*, 8(3):231–274, 1987.
12. Barbara Liskov and Jeannette M. Wing. A new definition of the subtype relation. In *ECOOP '93, Kaiserslautern, Germany*, LNCS 707, pages 118–141. Springer-Verlag.

13. Florence Maraninchi. Operational and compositional semantics of synchronous automaton compositions. In *CONCUR '92*, LNCS 630, pages 550–564. Springer-Verlag, 24–27 1992.
14. Sabine Moisan, Annie Ressouche, and Jean-Paul Rigault. BLOCKS, a Component Framework with Checking Facilities for Knowledge-Based Systems. *Informatica*, 25(4), 2001.
15. Oscar Nierstrasz. Regular types for active objects. In Oscar Nierstrasz and Dennis Tsichritzis, editors, *Object-Oriented Software Composition*, pages 99–121. Prentice Hall, 1995.
16. Object Management Group (OMG). OMG Unified Modeling Language Specification. URL: http://www.omg.org/, February 2001. version 1.4 (draft).
17. Pascal Rapicault and Frédéric Mallet. Behavioral specification of Java components using SyncCharts. In *Workshop on pervasive component systems*, June 2000.
18. Joãn Pedro Sousa and David Garlan. Formal modeling of the enterprise JavaBeans component integration framework. In *FM'99*, LNCS 1709, pages 1281–. Springer Verlag, September 1999.
19. Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. ACM Press and Addison-Wesley, New York, NY, 1998.