Behavioral specification of Java component using SyncCharts.

Pascal Rapicault – Frédéric Mallet
Laboratoire I3S UPRES_A 6070 du CNRS
Pascal.Rapicault@essi.fr* - Frederic.Mallet@unice.fr**

Abstract.

Nowadays, using a component is not easy when one does not know the method call linking. This is due to the fact that "Component specification focuses on the interface and fails to capture the behavior". In this position paper, we propose an approach using SyncCharts, to describe the required sequences of method to call in order not to misuse a component. We also present a transparent architecture and implementation of a runtime verifier based on the latter.

Since the late 60's software community is in crisis [NATO68] and tries to maximize the quantity of code reused by many ways. In these ways one can list, Object oriented or component paradigms. In the early days of this last approach, people envisioned a software component to be reused as an electronic component. However one quickly realized this would be difficult, because of the under specification of software component in comparison of electronic ones. So, a software component can't be just plugged in or replaced by another one, because it relies on a strong assumption of the sequence of things that needs to happen to make a correct use of it, and those sequences are not the same from one component to another one, even if they offer the same services. The other problem with this sequence, is that it cannot be guessed, and it is very hard to be explained clearly with a paper documentation or some examples which will never be exhaustive.
In fact, like stated in the workshop description, "component specification focuses on the interface and fails to capture the behavior" whereas components become more and more intricate, and a user may wish he could get some information, in order to know how to use the component. We identify this information as the "protocol of a component". This information could be very helpful for a user to develop or debug a component-based application.
For example, let's assume one get the media player component exposing the following interface:

`Public void init()`	to initialize the player
`Public stream load(File)`	to load a file to play
`Public void play(Stream)`	to play the file
`Public void getPosition()`	to get the current position
`Public void stop()`	to stop the play
`Public void quit()`	to release resources (close files, …)
`Public void loadAndPlay(File)`	to load, then play a file

In this component that offers two ways to play a file (one advanced, one simple), how can a user guess that the simplest way - simple mode - to use this component is by the method `loadAndPlay` which is sufficient; and one can also use - advanced mode – the `init`, `load` and `play` methods. In that mode, he must know the right execution sequence: `init` has to be called before `load` and `play`. Here the knowledge of the component protocol would be helpful.
Basically, there are three separate times in the software life cycle where information on how to use a component (according to its protocol) is useful:

- At development time, in order to know how to use a component;
- At compile time, to check as soon as possible if a component is correctly used;
- At runtime, to help developers to understand misusages of the component detailing errors.

* Rainbow Project : http://www.essi.fr/~rainbow
** Sports Project : http://www.i3s.unice.fr/~map/WEBSPORTS/index.htm

At development time, the user needs to access the protocol of the component in order to know how to use it. The sure detection of a protocol usage at compile time is theoretically impossible since this problem is related to problems of code reachability in a full program, which is not decidable. However, a less sure help which should only give partial errors may be provided at compile-time, but we have to make additional investigation in this field.
At runtime, a sure detection of the good use of a protocol is easier, however it just covers one execution trace at a time. So a correct execution does not guaranty that another one will be correct too. However an unexpected method being called can cause the execution to fail and an error message indicating the reason can be displayed. For example, calling the `load` method before calling the `init` one will raise an error. The error message displayed in case of protocol misuse is very useful when a user doesn't understand a failure due to a badly used component in his code. This proposal will emphasize locations of protocol misuses, and give which message sequences are expected before being able to send such a message.

To make such checks possible, data describing the protocol of the component are required, and need to be in somehow connected with it.
This paper describes such a method, using real-time systems mechanisms to describe in a formal way the component behavior. Before describing its architecture and its implementation, we first explain the formalism used to describe the protocol and its usage. Next, we terminate by a conclusion and the coming work.

Part 1 - Data representation for the protocol.

What we want our data to capture is not the whole usage of a component, and so its inner working, but only the user exposed part of its interface, the protocol. So, the nature of our protocol made us choose the StateCharts [Har87, UML97] as a representation of it. Because StateCharts "are used to model the situations during the life of an entity in which it satisfies some condition, performs some activity, or waits for some occurrence", they exactly give the required information to feed our system that needs to know which method it is authorized to call after another one. Another point is that since their usage is recommended by the RUP [Kru98] to represent the behavior of entities, so one can imagine that instead of loosing this information after the development, it could be used to document (decorate) the component.
Before using this graphical representation we thought using path expressions. However, the expression of complicated behavior quickly becomes painful. Moreover, to be interpreted, path expressions are often translated into automata which building time may be exponential.
Instead of using StateCharts which interpretation may be ambiguous, we decided to use a representation of reactive behaviors from the real-time / synchronous systems called SyncCharts [And96]. SYNCCHARTS is an acronym for Synchronous Charts. It inherits from StateCharts [Har87] and Argos [Mar90]. It is based on the synchronous paradigm and it offers enhanced preemption capabilities. Like Esterel, it deals with sequence, concurrency, preemption and communication in a fully deterministic way. The main differences with StateCharts are that SynchCharts has a stricter mathematically-defined semantics and a richer preemption management. This strict semantics allows, in relation with the synchronous framework, some reachability or safety proofs. Moreover, it has the ability to be safely compiled into a system of boolean equations. This ability allows its integration into a software component [MB99].

SyncCharts is designed to represent the behavior of reactive systems. Those systems must react instantaneously when an input signal occurs. The instantaneously reaction means that the resulting behavior has to be computed and output signal must be emitted in the same synchronous instant. Our purpose is to consider software components as reactive components. They should react instantaneously to a method call (signal input) and must instantaneously computes their ability or not to execute the method due to the protocol definition. For example, a media player will say, he refuses to compute the `load` method until the `init` method has not been called. Obviously, the method execution is not supposed to be instantaneous.

Let us now represent the SyncCharts[1] of our media player component described previously :

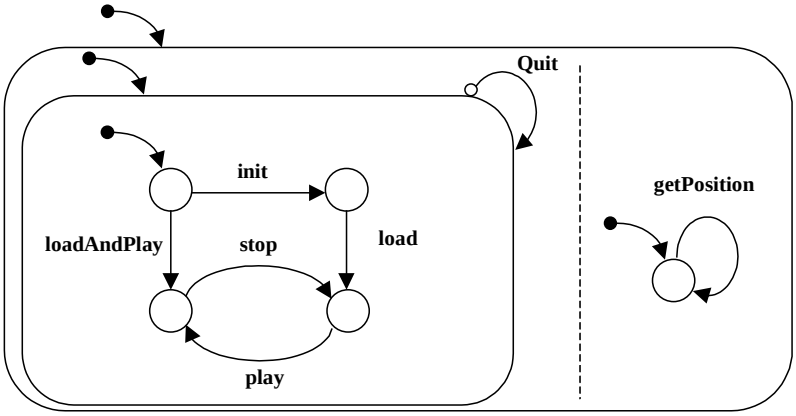

Figure 1: Media player SyncCharts

The SyncCharts semantics is to ignore any unknown input signal (here a method call) when it is not expected. So, if we conform to it, in the example diagram calling the `load` method before calling `init` there will not have any behaviour whereas we expected to have one (cause an error).
The solution would have been to add from every state all the unexpected transition to a common "dead" state, but this would have made the diagram unnecessarily complex. To avoid that, we extended the SyncCharts graphical semantics for the previous statement to be part of it. This is only a syntactic extension which has as an immediate consequence, the simplification of the drawn diagram and the conformance to what we expect. In order to take into account that some methods are authorized at any instant without any restriction, we provide a concurrent SyncCharts added in parallel of the previous one. Practically, this is just a state with a looped reaction to itself on specific method calls. Our diagrams only represent usage of the exposed part of an interface, every method call from an non exposed one will generate errors when used. To avoid this problem, we chose to provide a list of all methods that need to be taken care of.

The use of this notation permits to express in a clear way complex behaviors like preemption. Moreover it sums up in a unique place and easy to understand format, the usage interface of a component.

Part 2 – Embedding the protocol

While designing the architecture of our system to do checks, we kept in mind as an objective to make things happen transparently for a user, and not to change the source code. As a consequence, the resulting architecture keeps the diagrams and the initial components separated.

So, the programmer of a component who wants to document his component usage interface will have to put the corresponding SyncCharts in a separate file and may reference it in the code comments. This happens at development time of the component itself.

In order to avoid problems like the one with documentation (one never knows where it is, understands it,…) SyncCharts are directly embedded in the class itself. For all this embedding, diagrams and components are still separated, because the diagram never comes to pollute the component code.

To implement this transparent integration of a component and a diagram, we used the flexibility of the class file to store them and so put them in the user attribute [LY96]. A user attribute is in a class file, a

[1] The SyncCharts representation is used in 'Esterel studio' an industrial product distributed by Simulog.

non definite place where one may record any information under a given name. Usually it's used to store debug attributes.
Inserting the SyncCharts into the class file is the only non automated preparation (transformation) the component explicitly needs to receive, and that must be done by a human being.

Currently, a diagram only embeds the component usage interface. However we can imagine embedding another SyncCharts describing the inner work of the component. This would allow checks, not only from an external usage point of view, but also from an internal one (enabling checks for inheritance usage).

Since SyncCharts (component behavioral description) and components are now always gathered in the same place, it's easy to use these new data at development time into a component protocol visualizer, at compile time to give recommendations, or at runtime to check the good use of the protocol.

Part 3 – Runtime protocol checking.

With the SyncCharts, we introduced a way to describe the usage of a component. This section presents the architecture of a solution based on these diagrams to perform checks, and describes an implementation of it.

An extra component with the ability to understand a SyncCharts is introduced. It will be used to check if a method call is authorized, and plays the role of a guard. We'll refer to it as the behavior verifier. This verifier is independent from the component and the SyncCharts it needs to check. There exists one instance of behavior verifier by component and SyncCharts instance. All those separated components are related as follow: a SyncCharts is embedded in a component, and the behavior verifier is bound to a component via method calls. This connection is done catching the message send on the component and asking the behavior verifier if the method is expected. The system initialization is as follow: when the component is loaded, the corresponding SyncCharts is loaded and prepared to create a behavior verifier instance. Every component keeps track of its associated verifier using a specific reference.
Instead of using a specific verifier we could have used assertions [Mey92], however even if could have been used, they are not really adapted because dealing with the internal state of component data, whereas with SyncCharts we deal with the behavioral state. Moreover, assertions are designed to be directly inserted in the code, and so are not as transparent as we wished.

On the implementation side, a transparent integration of the previously described architecture requires to use a Meta Object Protocol, and to find a component being able to operate on SyncCharts.
The MOP we used is Javassist [Chi00]. It allows us to perform behavioral reflection like field addition or method wrapping. The first feature is used to add a field keeping the reference to the verifier and the second to automatically call the verifier before a method is executed. It should be noticed wrappers are only installed on methods that requires a check (this information is contained in the SyncCharts). Since we have as many behavior verifier instances as component instances, the instantiation was changed so that a new behavior verifier is created when a new component instance is created.

These transformations on the component are done at load time by using a specific loader provided with the MOP (it needs to be recalled that in a Java, before being able to be instantiated, classes are loaded in the JVM by a class loader). This loader is the cornerstone of our verification system, since it is required to load the component to "debug" through it. Concretely, the only thing to do to check a component is to change the program invocation command line so that classes are loaded with this specific loader.

The following figure depicts the transformation done at load time:

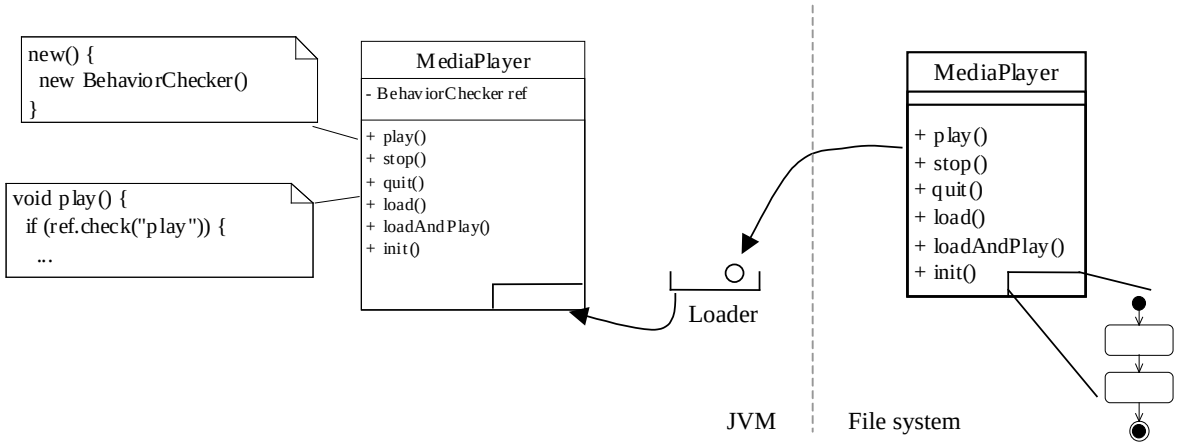

Figure 2: Load time modification of a class.

Because operating on SyncCharts, and in fact on a set of boolean equation is not easy, in order to implement the behavior verifier, we decided to reuse a component previously designed in SEP [MB99].

Thanks to this clear separation between data, component, and behavior verifier, one doesn't have to change anything in his program, and over all, he doesn't have to manage two sets of components i.e.: one for the verification mode, and one for the regular use.

Let's now see the runtime cost of such modifications. When the component is used in a normal mode, since the mechanisms described here above are not installed, there is no over cost, even generated by the presence of the SyncCharts at the end of the class file. Indeed user attributes are only loaded on demand. When used in a "verification mode", an over cost occurs in the following phases:

- Class loading: the over cost generated is due to the class reification and the modifications previously described.
- Instance creation of the component: the creation of a new instance of a component implies the creation of a new instance of the verifier.
- Method call: For every method call interested in checking, a call to the behavior verifier is done. We remind you that internal methods are not checked.

Conclusion and future work

With this architecture based on a clear and easy to understand formalism to describe states of a component (SyncCharts), we proposed a way to describe a component behavior in a tool usable format which can be embedded in the component. Moreover, this solution fits in the component models (but eXtreme ones) since StateCharts (SyncCharts like diagrams) are advised to use.
So, this way a developer finds an extra reason to do those diagrams, since it may avoid him to write a documentation and his specification are directly re-used. We could imagine that he designs its SyncCharts in parallel of its code, and refers it into his code. When the code is compiled, the diagram is automatically embedded in the class file. This solution represents an interesting alternative to the part of the documentation that shows examples.
Thanks to its clear separation with the component (no source code needs to be modified), the component decoration can be done afterwards.

The extensions we are thinking of are:
- On the model side:

- To study an extension of this architecture to manage a protocol shared between several components;
- The description of a component inner behavior;
- The meaning of inheritance for such diagrams.

- On the implementation side :
  - The management of parallelism;
  - The component instance selection, because one don't especially want to check the behavior of every component instance;
  - SyncCharts verification at compile time.

References.

[And96] : C. André, "*Representation and analysis of reactive behaviors : a Synchronous Approach"*, Invited paper, CESA'96, pp.19-29, Lille, France.

[Chi00] : S. Chiba, "Load-time Structural Reflection in Java", in ECOOP '00, to be published.

[Har87] : D. Harel "*StateCharts, a Visual Formalism for Complex Systemes"*, Sc. of Computer Prog., Vol 8, pp. 231-274.

[Kru98] P. Kruchten, "The Rational Unified Process", Addison-Wesley, December 1998

[LY96] : T. Lindholm, F. Yellin, "The Java Virtual Machine Specification".

[Mar90] : F. Maraninchi, "*ARGOS: un langage pour la conception, la description et la validation des systèmes réactifs"*, PhD thesis, Université Joseph Fourier, Grenoble I, France.

[Mey92] : B. Meyer, "Applying design by contract", IEEE Computer, vol. 25, num. 10(October 1992), page 40-51.

[MB99] : F. Mallet, F. Boéri, "*Esterel and Java in an Object-oriented framework for Heterogeneous Software and Hardware system Modelling and Simulation : The SEP approach"*, Euromicro'99, Vol I, pp.214-222, Milan, Italie.

[NATO68] : NATO Software Engineering Conferences in 1968 and 1969.

[UML97] : G. Booch, I. Jacobson, J Rumbaugh, Unified Modeling Language, Rational Corporation, 1997.

in whereas component becomes more and more intricate.