JTN: A Java-Targeted Graphic Formal Notation for Reactive and Concurrent Systems*

Eva Coscia and Gianna Reggio

DISI, Università di Genova – Italy
Fax: 39-010-3536699
{coscia,reggio}@disi.unige.it

Abstract. JTN is a formal graphic notation for Java-targeted design specifications, that are specifications of systems that will be implemented using Java.

JTN is aimed to be a part of a more articulated project for the production of a development method for reactive/concurrent/distributed systems. The starting point of this project is an existing general method that however does not cover the coding phase of the development process. Such approach provides formal graphic specifications for the system design that are too abstract to be transformed into Java code in just one step, or at least, the transformation is really hard and complex.

We introduce in the development process an intermediate step that transforms the above abstract specifications into JTN specifications, for which the transformation into a Java program is almost automatic and can be guaranteed correct. In this paper we present JTN on a simple toy example.

Introduction

In this paper we present a part of a more articulated project we are currently working on: a development method for reactive/concurrent/distributed systems (shortly systems from now on) that are finally implemented in Java. Such development process should be supported by formal tools and techniques, whenever possible, and by a set of user guidelines that describe in detail how to perform the various tasks. The formal bases and the main ideas come from previous work of one of the authors about the use of formal techniques in the development of systems that, however, did not ever considered the final coding step, see, e.g., [1,2,10,11]. We chose Java as the implementation language since it is OO, widely accepted for its simplicity and, at the same time, for its richness. It is considered a language for the net, for its portability, but also a language for concurrency and distribution. Moreover, there exists a precise, even if informal, reference [4] for the semantics of the core language.

[11] presents a general method for giving formal graphic design specifications of systems, but such specifications are too abstract to be transformed into Java

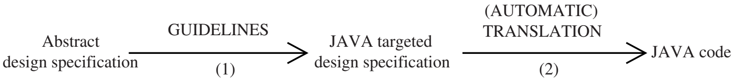
* Partially funded by the MURST project: Sistemi formali per la specifica, l'analisi, la verifica, la sintesi e la trasformazione di sistemi software.

J.-P. Finance (Ed.): FASE'99, LNCS 1577, pp. 77–97, 1999.

code in just one step, or at least, the transformation is really hard and complex. For example, following [11] you can specify systems with n-ary synchronous communications, where the components can exhibit any kind of non-determinism and can be coordinated by complex scheduling policies, which cannot have a direct implementation into Java.

Moreover, the complexity of this transformation into Java does not allow to check the correctness of the generated code, and there is no way to automatize it. Furthermore, [11] does not take into account the relevant, good characteristics of Java, as the OO features.

We think that it is useful to introduce an intermediate step in the development process that transforms an abstract design specification into a Java-targeted one, whose transformation into a Java program is really easy.



Step (2) can be automatized and guaranteed correct whereas step (1) cannot be automatized, but we are working to give a rich set of guidelines for helping the user in such task.

Here we present JTN, a graphic formal notation for the Java targeted design specifications, obtained by targeting [11] to Java, that is by modifying the specification language to take into account the features and the limitations of Java.

JTN is graphic because every aspect of system (e.g., components, global architecture and behaviour) is described only by diagrams. But it is formal, because the diagrams composing the specifications are just an alternative notation for logic specifications as in [11] (the formal semantics of JTN is presented in [3]).

We think that JTN, with the associated method and software support tools, could help the development of reliable systems implemented in Java.

– We can describe the system design graphically, and that is helpful to grasp the system characteristics. However, the JTN graphic specifications are structured and that avoids one of the possible drawbacks of graphic notations: to handle very large diagrams that could be not understood.
– The level of the JTN descriptions is not too low; the designer avoids to specify too much
  details and the drawings are simple enough; for example, in JTN there are user defined data-types and abstract communication mechanisms, as synchronous and asynchronous channels.
– JTN specifications are formal, with the usual advantages to use formal methods without bothering the specifiers with too much formalities.
– The automatizable translation to Java reduces the time to a working system and also gives a prototyper for such specifications.

There are neither theoretical nor practical problems to realize a full tool-set for supporting the use of JTN using the current technology (e.g., interactive editor, static checker, hyper-textual browser, translator to Java, debugger); it is possible to realize them within a reasonable amount of time, we just need some human resources.

Here, for lack of room, we consider only a rich subset of JTN applied on a toy running example and give some ideas about its translation into Java; a detailed presentation of JTN and further examples are in [3]. In Sect. 6 we present the relations with other works as well as some hints on our future work.

The running example We specify the design of a Java program simulating a pocket calculator that computes and interacts with a keyboard, a display and a printer; think, for example, of a small application simulating a calculator on the desktop of a computer. The functionalities of the calculator are quite obvious: it can receive, by the keyboard, numbers and simple commands for performing operations (addition and multiplication) and for printing the display content.

1 JTN

In this section we first describe the main features of the abstract design specification technique of [11]; then we describe how to target it to Java and give an overall presentation of the JTN notation.

1.1 Abstract Design Specifications

The specification technique of [11] distinguishes among the data-types, the passive and the active components of a system, because these components have a different conceptual nature and play a different role within the systems.

The *data-types* are static structures used by the other components, with no idea of an internal state changing over time. The *passive* components have an internal state that can be modified by the actions of other active components. The *active* components have an internal state, but they are also able to act on their own (possibly by interacting among them and with the passive components).

We can completely describe a data-type by giving its values and the operations over them, whereas we describe the passive components by giving their states and the actions that can be performed over them, which obviously can change such states. We want to remark this difference. Data-types define stateless elements (essentially, values) that are used by (active and passive) components. Instead, the passive components are actual components of the system, having an internal state that can be updated by the active components. Finally we describe the active components by giving the relevant intermediate states of their lives and their behaviours, which are the possible transitions leading from one state to another one. Every transition is decorated by a label abstractly describing the information exchanged with the external, w.r.t. to the component, world. Notice that many transitions can have the same source, and that allows to handle nondeterminism.

Let us consider, for simplicity, a system having a database inside. The database is a passive component whose internal state is modified by the operations it supplies outside; the data managed and exchanged by the database are data-types; and the processes using the database are the active components of the system.

The activity of a system results by describing how its components *cooperate*, i.e., to say which transitions of the active components and which operations over the passive components have to be performed together, and which is the exchange of information with the external (w.r.t. the whole system) world.

A system, in turn, can be seen as an active component of another system, and so we can specify systems with a multi-level architecture.

The underling formal model for an active component (and thus for a system) is a labelled transition system (LTS), that is triple consisting of a set S of elements (intermediate states), a set L of the labels (interactions with the external world) and a ternary transition relation, a subset of $S \times L \times S$ (transitions). The passive components and the data-types are modelled by first-order structures.

The graphic specifications of [11] follow the system structure; thus they consist of diagrams for the data-types, for the components (the behaviour of the active components is represented as a kind of finite automata) and for the cooperation among them. Formally, these diagrams correspond to an algebraic/logic specification having LTS's as models, see [11]. Note that the specification language of [11] has neither concepts nor mechanisms related to OO, nor features of some particular programming language, as handshake communications and asynchronous channels; instead it allows the specifier to directly define any feature of the system by describing the corresponding behaviours and cooperation.

1.2 Java-Targeting

We designed `JTN` by adapting the technique presented in Sect. 1.1 to the features and to the limitations of Java.

We want to keep distinct, at this more concrete level too, the concepts of data-types, passive and active components of a system. In our opinion, it is useful to have this distinction to avoid confusion and to make the specification more readable.

Then, we introduce the new OO concepts of class and instance and provide an explicit representation of the relevant relationships among them, as inheritance and use; but in `JTN` we have three kinds of classes, one for each kind of entities that we consider: data-types, passive and active components.

The data-types are described at an abstract level by giving their constructors and by defining the associated operations, without considering an OO perspective; however it is easy to transform them into Java classes.

The passive components are seen as objects, whose state is given by a set of typed fields and the operations to modify them are methods. The transformation of such object specifications into Java classes is immediate.

The active components are seen as processes, with a state, an independent activity and communication channels to interact with other active components and with the external world. In this case, the natural implementation is given by Java threads. We chose to use communication among processes via channels, rather than via method calls. In this way, a process does not offer methods outside and we do not have to manage method calls while the process is performing its activity (in Java, there is no built-in mechanism to disable method calls).

As Java objects and threads communicate by method calls and streams, the main typologies of cooperation are: between a process and an object, by means of a call to an object method, and among a set of processes by communication along asynchronous and synchronous channels. The asynchronous channels are rendered by streams, and the synchronous ones are implemented by particular additional objects.

Java supports only system architectures of at most two levels. The first level corresponds to multi-threaded Java programs, and the second corresponds to distributed Java applications consisting of programs possibly running on different machines. For lack of room, in this paper we do not consider the second level.

For the same reason, here we consider simple objects and processes, i.e., without sub-objects and sub-processes, so calls to methods of other objects cannot appear in a method body. Moreover, we do not consider the dynamic creation of process and objects: we assume to start with an initial configuration of the system where all the components have been already created in the initial state.

Using the `JTN` concepts we model the running example as follows. The calculator has four active components: the keyboard driver reading keys from the keyboard, the computing unit performing the computations, the display driver echoing inputs and results to the display, and the print driver printing the display content. All such processes use an object, which records the content of the display, and three different types of data: digits, lists of digits and commands.

The keyboard driver receives the keys by an asynchronous channel from the keyboard (the external world); the display driver and the print driver send their outputs to, respectively, the display and the printer by two other asynchronous channels. The communications and the synchronizations among the active components are realized by some synchronous channels.

1.3 Overall Structure of the `JTN` Specifications

We factorize a system specification into several diagrams showing different aspects or parts of the system. Thus the diagrams are not too large and complicated, and so really useful. For example, some diagrams focus on the behaviour of the components and other focus on the architectural structure of the system.

Class Diagram The *class diagram* captures the classes of the system components and their relationships. We consider three different kinds of classes: *data-types*, *object classes* (for passive components) and *process classes* (for active components), graphically represented by different icons.

All the information about a class is given by two complementary diagrams: *interface* and *body*. The first one describes which are the services (different for each kind of class) that the class offers outside. The latter defines such services and can be given apart from the class diagram. The forms of the two diagrams depend on the class kind and are described in deeper details in Sect. 2.

In an OO perspective, stand-alone classes are not so meaningful; most of them are related to accomplish more complex functionalities. Thus, we complete the class descriptions with the relevant relationships among them.

Inheritance between classes of the same kind, it states that a class is a specialization or an extension of another one.
Usage states that a class (of any kind) uses the data defined by a data-type.
Clientship states that a process class assumes the existence of an object class, as it can use its methods.

`JTN` defines the operations, the methods and the process behaviours inside the body diagrams by ordered lists of conditional rules, with a uniform presentation, just a general form of "guarded commands". The alternatives of a command are evaluated in order and the first one having a satisfied guard is chosen. The guards are partly realized by a boolean condition and partly by pattern matching (as in ML [6]) over the parameters of the operation, of the method call and over the state of the process respectively. The use of pattern matching is useful to make shorter and more readable the whole definitions. Let us remark that we avoid problems with overlapping patterns and conditions by explicitly ordering the guards.

Architecture Diagram The architecture diagram describes which are the components of the system, and how they interact (by which communication channels and by which method calls).

Sequence Diagram A sequence diagram is a particular form of message sequence chart (see [8]) that describes a sequence of actions occurring in a (possibly partial) execution of the system and involving some components. The represented actions are communications over channels and method calls. We introduced these diagrams because they are used in the most widely accepted specification techniques in the field of Software Engineering, such as UML.

The class diagram (with the possibly separated body diagrams) and the architecture diagram fully describe a system. The sequence diagrams are an additional way to present information on the system that is very intuitive and easy to be understood.

2 Class Diagram

There is one global class diagram for the whole system, representing the classes of all its constituents. It is a graph, where the nodes represent classes and the arcs class relationships. The icons for a data-type, an object class and a process class are, respectively, 🗋 ⧉ ▢.

For each class there are two diagrams, *interface* and *body*, both with a slot with the name of the class, i.e., the type of its elements. The interface diagrams are always in the class diagram, whereas the body ones can be given separately.

The contents of the interface and of the body diagrams vary with the kind of the class and in the following subsections we present them and the relationships among classes. In fig. 1 we report the class diagram for the calculator example.

Data-Type We chose to describe a data-type by giving its constructors and defining the associated operations by means of conditional rules with pattern matching a la ML (see [6]).

The interface diagram for a data-type contains the list of its visible constructors and operations, and the body diagram contains the the private constructors and the definition of the visible and private operations. The body diagram is divided into many slots, separated by dashed lines, each one containing the definition of an operation, by conditional rules.

The most common data-types, either basic (e.g., NAT) or parametric (e.g., LIST), are predefined and implicitly used by all the classes, so we do not report them in the class diagram. Moreover, data-types defined by combinations of predefined ones can be renamed and grouped together. In fig. 1 DIGIT is a renamed subrange of CHAR and KEY is the union of DIGIT and COMMAND.

The APPLY data-type, defined in fig. 1 by inheriting from the others, contains some operation definitions, one public, Apply, and two private, Code and Decode. It implicitly uses NAT. Decode is defined by using the pattern matching: given an actual parameter a, if a matches the pattern Empty (i.e., a= Empty, Empty is a constant constructor), then it returns 0; if a matches d::dl (i.e., $a = e :: l$, :: is the list constructor adding an element to a list), then it returns (Ord(e)-Ord('0')) + 10 * Decode(l).

Object Class An object is a passive component of the system, which has an internal state but it does not perform an independent activity. Objects cooperate with other components by offering services (i.e., methods) that the processes can call to complete complex functionalities.

Here, for lack of room, we do not present the complete version of the object classes with sub-objects and local methods.

Interface Diagram The interface diagram of an object class contains the list of the public methods with their names, the types of their parameters and of the returned values (if any). The DISPLAY class (fig. 1) has two methods, Write and Add, with one parameter of type DIGIT_LIST, and Read, with no parameter, that returns a DIGIT_LIST value.
Body Diagram The object body diagram is divided into two slots, containing:

- the *fields*; in `JTN` there are only private fields that can be accessed only by methods. For each field we give the name and the type plus its initial value (see field Cont in fig. 2);
- the definition of *public* and *private* methods.

Using `JTN`, we define the methods by conditional rules with pattern matching.

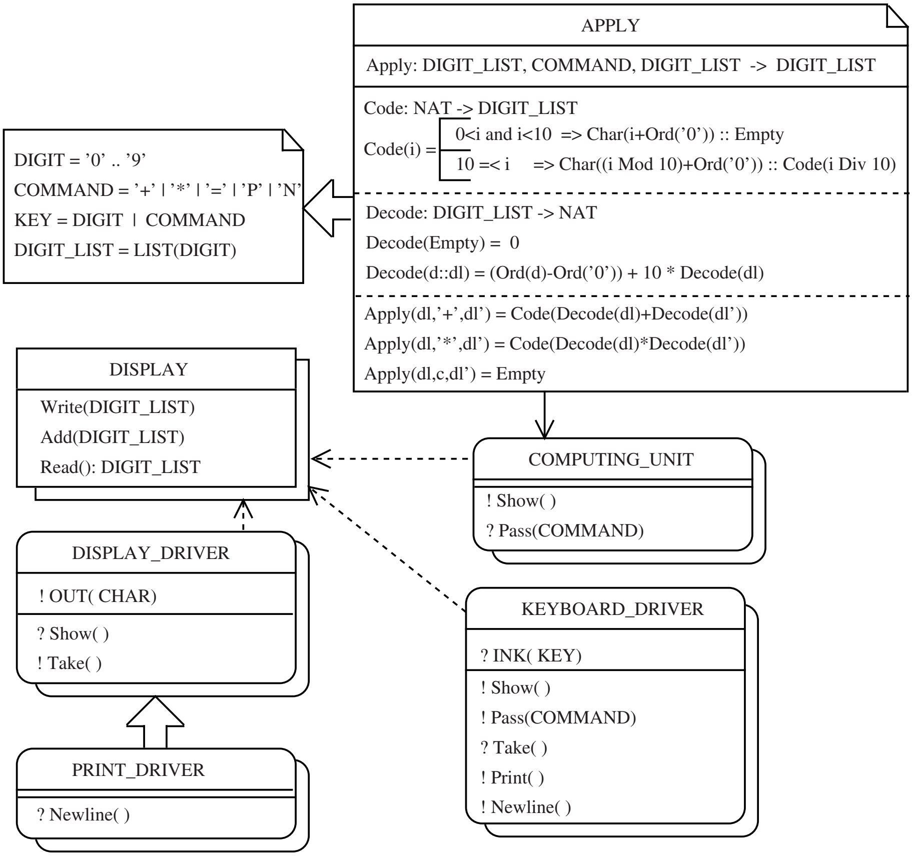


Fig. 1. Calculator: Class Diagram

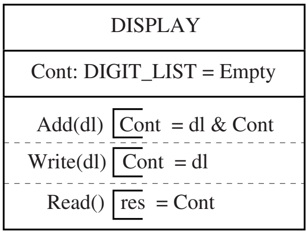


Fig. 2. Calculator: DISPLAY body diagram

A method $\mathrm{M}(\mathrm{IT}_1, \ldots, \mathrm{IT}_k)$: OT is defined by an ordered list of conditional rules whose form is

$$\mathrm{M}(i_1,...,i_k)\left[\begin{array}{l} \text{if cond}_1 \; \{ \text{ assignment\_list}_1 \, \} \\ \hline \cdots \\ \hline \text{if cond}_n \; \{ \text{ assignment\_list}_n \, \} \end{array}\right.$$

where: for each h ($1 \leq h \leq k$) i_h is a pattern, i.e., an expression of type IT_h built only by constructors and variables; for each j ($1 \leq j \leq n$) cond_j is a boolean expression over the variables in $i_1,..,i_k$ and the object fields; $\mathrm{assignment\_list}_j$ is a list of assignments of form either “f = e” or “res = e”, with e an expression over the variables in $i_1,..,i_k$ and the object fields, f a field name and res a special variable of type OT denoting the result returned by the method.

A method call is executed as follows. We find the first, w.r.t. the ordering, rule whose pattern matches with the parameters of the call and with a condition that holds. If none matches, then we have an error. Then, the corresponding assignments are performed. The value returned by the method (if any) is the final value of the variable res. Trivial examples of method definitions are in the body of the class DISPLAY in fig. 2 (& is the operation for appending lists).

Process Class In our approach, processes are different from objects and their description cannot be given in the same way. First note that processes are *active* components that behave independently and do not offer methods outside. Their behaviours are not sequential, instead they run concurrently and cooperate by message exchange with those of the other processes. So, the process interface diagram does not contain methods, but the communication channels, synchronous and asynchronous.

The body diagram describes the behaviour of the process, by presenting its interesting intermediate states, each one characterized by a name and typed parameters, and its transitions, precisely from every intermediate state, some conditional rules define all the states it can reach by interacting with the external world (i.e., the labelled transitions of [11]). In the general method of [11], there is no restriction on the form of the external interactions, which are just described by labels. In `JTN`, a process can communicate with the external world only by calling the object methods or by using the communication channels.

We give a graphic presentation of the behaviour that naturally depicts what a process does, by showing all the possible transitions starting from any state.

Interface diagram `JTN` processes use two kinds of channels: synchronous and asynchronous; both kinds of channels are distinguished into input and output ones. Thus the interface diagram for a process class has two slots containing the *asynchronous* and the *synchronous* channels, respectively, with their names, their directions (described by `!` and `?`) and the types of exchanged values (if any).

For example, the DISPLAY_DRIVER interface diagram in fig. 1 declares two synchronous channels Show and Take used only for a synchronization purpose (no value exchanged), and an asynchronous one, OUT, on which an instance of DISPLAY_DRIVER sends a char outside.

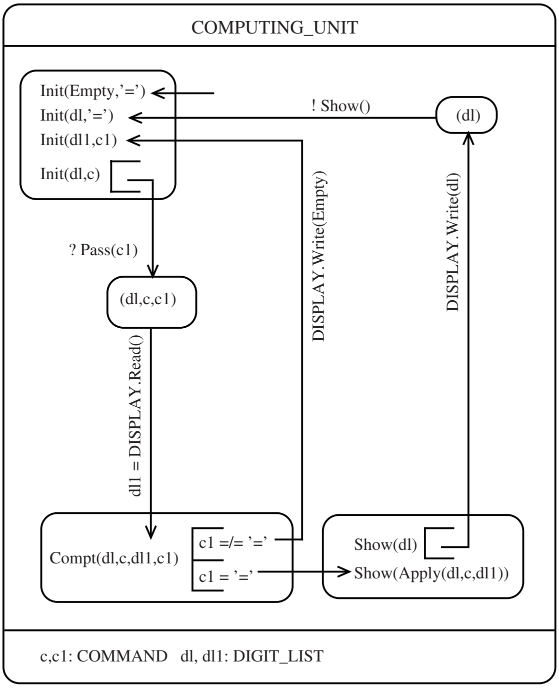


Fig. 3. Calculator: Computing Unit behaviour graph

Behaviour Graph [1] *(process body diagram)*
The *behaviour* of a process of a class is described by a graph, whose arcs represent "generic" labelled transitions, whose form is

$$S(pt_1,\ldots,pt_n)\ [\,cond(pt_1,\ldots,pt_n) \xrightarrow{l} S'(e_1,\ldots,e_k)$$

where S, S' are state constructors, pt_1, ...,pt_n are patterns for the state parameters, l is a pattern for the external interaction and e_1,..,e_k are expressions over the variables in l and $pt_1, \ldots, pt_n$.

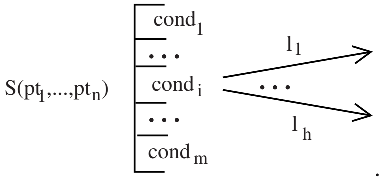
The label l can have six different forms, depending on the kind of interaction that is performed with the external world:

- ?ACH(x), !ACH(e): input from/output to an asynchronous channel;
- $r = oe.m(x_1, \ldots, x_n)$ (or $oe.m(x_1, \ldots, x_n)$ if m has no return type): object method invocation;
- $?sch(x_1, \ldots, x_k)$,$!sch(e_1, \ldots, e_k)$: input from/output to a synchronous channel;

[1] The behaviour graphs of JTN play the same role of the UML state diagrams.

- τ: internal activity, usually omitted.

All the sources and targets of transition representing state patterns of the same kind are grouped together into a node of the behaviour graph, avoiding repetitions. Thus each state pattern is written once, and the conditions are listed aside the corresponding state pattern to form an *alternative*, as below



The arrows leaving a condition are ordered. Thus, inside a node of the graph we have an ordered list of alternatives plus state patterns that are only targets of transitions.

The interpretation of the behaviour graph is as follows. When a process is in a state K(args), we consider in order all the alternatives inside the node for the K states, until we find one whose pattern matches args. Then, inside the chosen alternative, we look for the first true condition. Finally we consider the labels on the arrows leaving the condition, trying to determine the first one that can be executed (recall they are ordered). If no matching pattern with a true condition is found, then the process is definitively stopped.

Not all choices of labels $(l_1,..,l_h)$ are meaningful; the admissible cases are as follows, and for each of them we explain how to select the one to execute:

1. $h = 1$ and l_1 is a method call or an output on an asynchronous channel or an internal transition; the corresponding transition can be executed.
2. $h \geq 1$ and for each i $(1\leq i \leq h)$ $l_i = !sch_i(\dots)$ or $l_i = ?sch_i(\dots)$: all synchronous channels sch_i are checked in the order; if the communication on sch_i cannot be executed, then the following one is checked. If no communication can be executed, the process is suspended until the last communication completes.
3. $h \geq 1$ and for each i $(1 \leq i\leq h)$ $l_i = ?ACH_i(\dots)$: the asynchronous channels ACH_i are continuously tested in the order, until an available message is found.

In cases 2 and 3 we can add an arrow labelled with “else” with the meaning that whenever no other transition can be executed, such escape will be performed as an internal action, leading to another state from which the activity continues. An else label can be used in case 2 when no communication is immediately available, to return to the same state and start again the polling procedure. See e.g., the state Taking in fig. 4, where KEYBOARD_DRIVER can perform a synchronous communication on channel Take; otherwise KEYBOARD_DRIVER moves to another state (by else transition) in which it tries to read a character from the asynchronous input channel INK, and, if nothing is available, by another else transition, then it will come back to the state Taking.

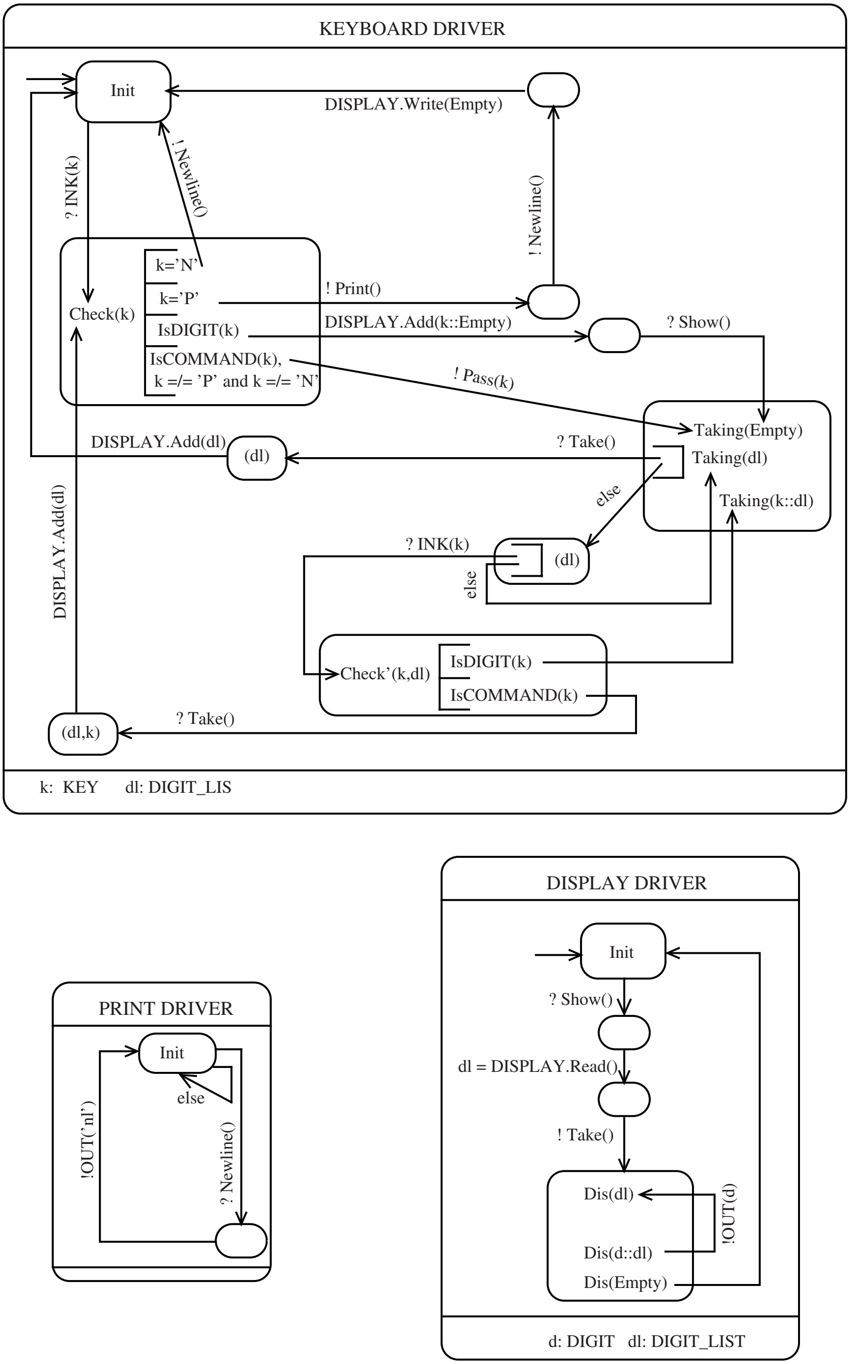


Fig. 4. Behaviour graphs

Instead of explicitly declaring in a behaviour graph each state constructor with the type of its parameters, we add a slot for declaring the types of the used variables; obviously each state constructor must be typed consistently.

In fig. 3 and 4 we omit the name of a state every time it is not relevant; in such a case, the icon is empty, or just contains the list of the arguments.

An arrow with neither starting state nor label enters in the initial state of the system (see the upper left arrow in fig. 3)

Class Relationships Here we briefly illustrate the relationships among classes that we can put in the class diagram.
Inheritance (⇦) states that a class extends another one. It is restricted to classes of the same kind. What really the word "extends" does mean, depends on the particular kind of class. With regard to data-types, inheritance is used to add new operations. An example is APPLY, that adds the operation Apply. When considering an object class, inheritance is a mechanism for adding new methods and fields; finally, when considering a process class, inheritance adds transitions (i.e., behaviour) and new communication channels.

In our example, the PRINT_DRIVER class inherits from DISPLAY_DRIVER: its interface diagram is the one of DISPLAY_DRIVER with a new synchronous channel Newline; the behaviour graph of class PRINT_DRIVER depicts only the new transitions (see fig. 4), implicitly assuming those described in the behaviour graph of DISPLAY_DRIVER. More precisely, the transitions starting from states of the same kind are merged together; the new alternatives for a given state, as well as new transitions associated with an existing condition, are added at the end of the list, as they represent alternatives to be considered after the existing ones (we are currently studying more suitable mechanisms to describe how to re-order these alternatives).

The three different inheritance relations define three hierarchies over, respectively, data, object and process types (i.e., classes).
Usage (⟵) states that a data-type is used by another class. It is represented by an arrow from the used data-type class to the using class. If all the system components use a data-type, then the usage relation is omitted.
Clientship (⇠) states that a class assume the existence of another one, because it calls its methods. In our example, the process of all classes call the methods of the DISPLAY class.

In this paper we do not consider structured objects, calling other object methods, so clientship relates only process classes with the classes of the objects whose methods they call.

3 Architecture Diagram

The architecture diagram describes the structure of the system showing its components and how they cooperate. The icons for the process and the object instances are slightly different from the corresponding ones for the classes; they are single boxes or single boxes with rounded corners: □ ▢.

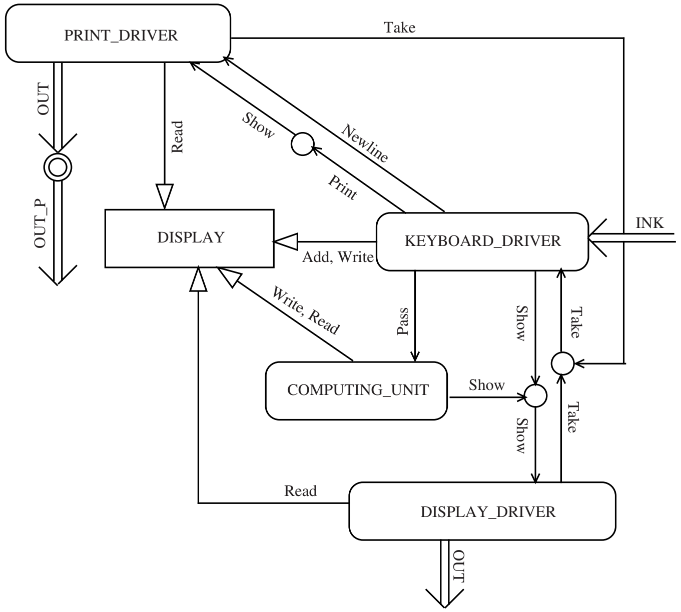


Fig. 5. Calculator: Architecture diagram

An instance icon contains only the instance identifier with the name of the corresponding class, separated by colons; the identifier is omitted if there is only one instance for such class, as in fig. 5.

The architecture diagram is a hyper-graph whose nodes are class instances that represent the components, and whose hyper-arcs represent how they cooperate. Let us remark that in this work we consider neither creation/deletion of components nor architectures having a generic number of components.

We can distinguish three kinds of hyper-arcs, representing:

method call: a process calls a method of an object; the icons of the two instances are linked by an arrow decorated with the method name; the arrow is oriented from the caller process to the called object.

asynchronous communication: fig. 6a) describes the connection of some asynchronous channels; $\mathrm{OAC_i}$ are output channels and $\mathrm{IAC_i}$ are input channels of the processes attached to the hyper-arc. The type of the exchanged message is the same for all the channels. Moreover, the channel types and versus must be in accord with the interfaces of the classes of the connected processes.

We can distinguish some cases. If $n = 1$,$m = 0$ or $n = 0$,$m = 1$, then the icon describes a channel for a process that communicates with the external world (e.g., the OUT channel associated with the DISPLAY_DRIVER in fig. 4). If $n > 0$,$m > 0$, a message sent on a generic $\mathrm{OAC_i}$ will be replied on all $\mathrm{IAC_1}$,..,$\mathrm{IAC_m}$.

If $n = 1, m = 1$ the channel connects two processes or it is used to rename a channel, as we can see in fig. 5, where channel OUT of PRINT_DRIVER is renamed as OUT_P to avoid name clash with the same channel of DISPLAY_DRIVER.

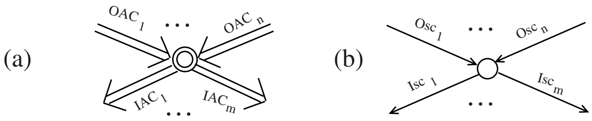


Fig. 6. Connectors for asynchronous and synchronous channels

synchronous communication: fig. 6b) describes the connection of some synchronous channels. We always have $n > 0, m > 0$, because synchronous channels cannot be used for communication with the external world. Again, the channel types and versus must be in accord with the interfaces of the classes of the connected processes.

The synchronous communication always involves two processes at a time: one process connected on a generic Osc_i acting as a sender, and one connected on a generic Isc_i acting as a receiver. Thus, the drawing can be interpreted as a shortcut for a set of channels connecting pairwise all sending to all receiving processes. An example is channel Show in fig. 5 that connects KEYBOARD_DRIVER, COMPUTING_UNIT (senders) and DISPLAY_DRIVER (receiver).

4 Sequence Diagram

A sequence[2] diagram is a kind of Message Sequence Chart [8] that gives a (possibly partial) description of a (possibly partial) execution of the system. Sequence diagrams are of particular interest because introduce in a specification formalism a technique that is used in the most widely accepted methods and notations in the Software Engineering field (such as UML).

A sequence diagram graphically represents some components taking part in a partial system execution and the ordered sequence of interactions among them and with the external world performed during such execution. The considered interactions are communications over channels and calls to object methods. The graphic presentation enlightens relevant aspects of the temporal ordering among interaction occurrences. Moreover, the diagram can be annotated with information about the state of the components, so it is possible to represent effects or conditions of action occurrences on single components.

The class and the architecture diagram supply complementary information about the system, whereas the sequence diagrams are just a different way to visualize information that has been already specified by the other diagrams. Several sequence diagrams may be presented for the same specification, to cover, e.g., the description of some interesting use cases of the system.

[2] In this case we use the same terminology of UML, since our sequences and the UML ones are rather similar; we can analogously define a form of collaboration diagram.

Sequence diagrams are not valuable for their information content (because it is already present in the other diagrams) but mainly from a methodological point of view and can be used for different purposes, for instance:

- to give a more natural and clear representation of the developed system to a client (e.g., to show how the calculator performs the addition);
- to show that the behaviour of the system specified by the class and the architecture diagrams is, in particular circumstances, the expected one. For example, by a sequence diagram we can show that if the user does not digit an "=" at the end of the operation, then the calculator does not return the result. From our experience, the construction of sequence diagrams, also if "by hand", helps to control the quality of the proposed design and allows to detect errors and omissions in the specification.

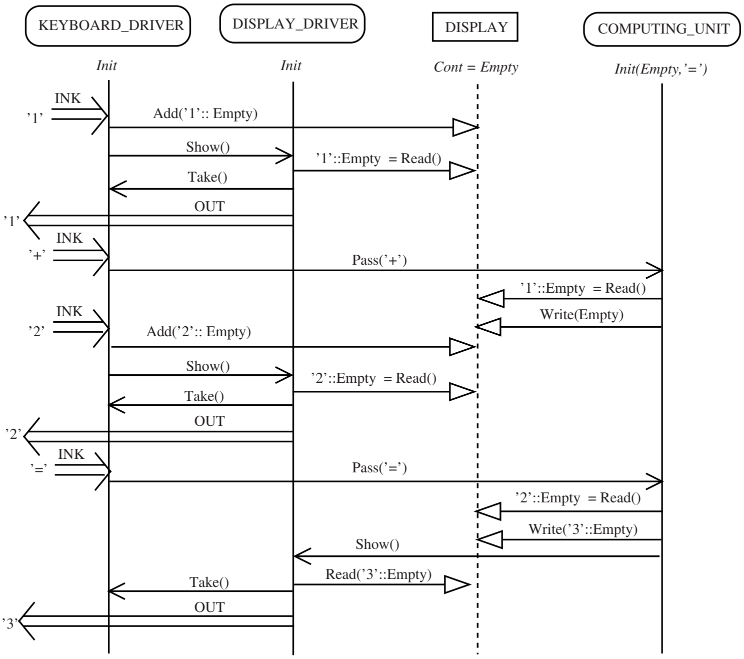


Fig. 7. Calculator: a sequence diagram for the computation of 1 + 2

Sequences diagrams are forms of message sequence charts, thus there are *vertical lines* representing the lives of the components involved in the execution. We use a dashed line to represent objects and a continuous line to represent processes. The *horizontal lines* describe the interactions occurring among the components or with the external world (i.e., communications on a channel, and method calls). Lines are put from top to bottom with respect to the temporal ordering of happening.

We use different icons for asynchronous communication (a double arrow), synchronous communication (a single arrow) and method call (a single arrow with outlined head) as we can see in fig. 7.

An asynchronous communication with the external world is just an incoming or outcoming double arrow, labelled by the channel name and by the exchanged data. An asynchronous communication between two processes is represented by two broken arrows. The part representing the start of the communication (send) is over the other one. They are separated by vertical dots and the data exchanged is annotated over both the two parts. Other actions may occur between the two phases of the asynchronous communications. A synchronous communication is decorated by the name of the channel and by the exchanged message. A method call is decorated by the name of the method and the parameters.

At any point of the vertical lines it is possible to put conditions on the value of the fields, for an object, and on the state and its arguments, for a process. The starting state of the execution may be described by such annotations, on the top of the corresponding vertical lines (see fig. 7).

Note that the elimination of a component and of its interactions returns another sequence diagram. If we drop DISPLAY_DRIVER in fig. 7, then we have a sequence diagram concerning only the updating of DISPLAY.

As sequence diagrams can erroneously depict executions that are not coherent with the rest of the specification, we must define when a sequence diagram is consistent with the information supplied by the class and the architecture diagrams.

Once we fixed the starting state of each instance, we can easily trace out how the system evolves. The object body diagrams describe how a method execution changes the state of an object. The behaviour graphs describe which communications or which method calls a process can perform from a given state and thus is the corresponding new state. The architecture diagram presents the topology of both the external as well as the inter-process communications. Thus, when we know which are the values arriving on the input asynchronous channels from the external world, we can find which actions the system components can perform and consequently which states the system can reach.

So, given a sequence diagram, we can determine the starting state of the system and then whether the depicted interactions can happen in the depicted order.

This consistency idea can be precisely defined, remembering that JTN is a formal specification language (the semantics of class plus architecture diagrams is an LTS) and that each sequence diagram corresponds to a formula in a branching time temporal logic saying that, from the starting state there exists a sequence of transitions where the depicted communications happened in the depicted order.

5 Implementation of Main Mechanisms

Here we briefly sketch out the implementation in Java of some of the JTN mechanisms. Obviously, the resulting Java program manages the classes and the instances shown in the diagrams, and also some auxiliary ones that are the standard implementation for synchronous and asynchronous channels and predefined data-types.

The predefined data-types are mostly obtained by combining Java primitive data (e.g., integer) and by extending some Java standard classes (e.g., Vector). The user-defined data-types have a standard translation: the constructors and their arguments are implemented as instance private fields; the component extractors operating on the data are trivial methods returning the value of the corresponding fields. The operations are translated into methods, whose code implements the guarded commands and the pattern matching used to define them.

The object classes are implemented as Java classes. The private fields implement the fields, initialized to the value represented in the body diagram; the methods are the direct encoding of the corresponding methods specified in the body diagram.

The process classes are implemented as Java thread classes and the intermediate states, described by constructors in the behaviour graph, are implemented by the fields of the class; the unique method is run, whose code is determined by the behaviour graph of the class.

```
class DisplayDriver extends Thread{
  private String state = "Init"; // state constructor implementation
  private List_Digit the_dl; // digit list;
  private Display the_d; // the display;

  private Synch_Sign Show; // synchronous communication channels
  private Synch_Sign Take;
  private FileOutputStream Out;  // asynchronous communication channel

  DisplayDriver(Synch_Sign s, Synch_Sign t, Display d){
   super();
   Show = s;
      ... // initialization part continues
  }
  public void run(){
   while(true){
    Show.get(); // receives a signal
    Take.put(); // sends a signal
    the_dl= the_d.Read(); // reads the Display content
    state = "Dis"; // changes its state
     while (!the_dl.isEmpty()){
      Out.println(the_dl.Head());
      the_dl = the_dl.Tail();
     }}}
}
```

Fig. 8. Java implementation of DISPLAY_DRIVER

The communication channels of a process are fields referencing particular objects. A synchronous channel is implemented by using a special object that act as a "synchronizer". When a process P1 tries to synchronize with P2, it accesses the synchronizer to check whether P2 is ready for the synchronization. If P2 is not ready, then P1 is suspended. When P2 is ready, P1 is resumed and reads or writes the exchanged data. To ensure that only one process at a time gains the access to a channel method, as well as to suspend-resume processes we use the `synchronized` and the `wait-notify` mechanisms of Java.

The asynchronous channels are trivially implemented by Java streams. In the particular case of asynchronous communication among process, we use the specialized stream classes for pipeline communication. Moreover, if the communication among processes involves m writers and n readers, a particular object implements the connector in fig. 6(a) that continuously reads a data from anyone among the input channels and replicate it on each one of the output ones.

6 Conclusions and Related Works

We think that JTN could help to design complex systems using Java, even if in this paper we have used it on a really toy example, for the following reasons:

- it is strongly visual; we have tried to visually render the process behaviours, the system architecture, the way the components cooperate, as well as the definition of data operations and of object methods;
- the complexity and the intricacies of the systems is mastered by keeping separated data-types, objects and processes, and allowing to design such entities at the most abstract level compatible with a direct implementation in Java; for example, data-types are not objects and the user can define her data with the constructors of her choice to represent them. For example, if we want to concatenate two lists L1 and L2, we do not have to create two objects realizing L1 and L2 respectively, and then call the concatenation method on L1 (or L2); instead, we just apply the concatenation operation to terms representing L1 and L2 respectively.
- there is a direct correct encoding of the specification into a Java program that it is possible to make automatic by the use of some tool.

JTN is not purely OO, as it only includes some OO concepts, precisely those that are useful to model the features of the considered systems. We use classes and instances, plus inheritance and other relationships, to model the three kinds of constituents of the systems (data-types, passive and active components). The interactions among processes via shared memory is modelled by objects and method calls; encapsulation allows to control how processes access the objects in the shared memory.

Although JTN is Java-targeted, it is not useful only to produce Java code; indeed it can also be fruitfully used to model and design systems implemented by using another programming language, as ADA.

Note that JTN is not addressed to real-time systems, because the abstract specification method of [11] and the features of Java do not adequately support real-time programming.

It is possible to produce a full set of software tools to support the use of JTN: from interactive graphic editors to a static checker including the consistency check of the sequence diagrams with respect to the other diagrams, to browsers enhancing the hyper-textual aspects of the diagrams composing the specifications, to the translator into Java. We are considering also a form of debugger obtained by using a variant of the translation into Java. The execution of the modified program produces an output that can be transformed in a sequence diagram and so we can have a graphic presentation of the execution. The underlined required technology for the realization of such tools is easily available. At the moment we are looking for human resources to realize them.

Our future work will consider how to complete JTN; we want to investigate the structuring of processes and objects by introducing sub-components, a mechanism for the packaging of classes when one global class diagram is too large, other communication mechanisms, the notation for the description of the distribution level of the architecture and so on. The notion of inheritance for the process class requires further investigations too, with the determination of an associated type hierarchy.

To fully take advantage of JTN we need to propose a method for passing from the abstract specifications of [11] to the more detailed JTN ones, that is guidelines and hints that help the user to perform this task.

We are not aware of other "Java targeted" specification languages/notations for systems in the literature, even if there exit tools for generating Java code from generic object-oriented specifications (e.g., ROSE for UML).

To relate our proposal to other approaches we must first recall that JTN is not an OO specification language, but it is intended for reactive/concurrent/distributed systems; this is the reason why it uses ingredients as processes strongly different from objects, system architecture and channels. However JTN encompasses a few OO concepts, for example, "object" as a way to encapsulate shared memory and "class" (for objects and processes) with inheritance as a way to modularly define "types" of objects and processes.

The JTN specifications are both graphic and formal, and in this respect JTN is similar to SDL [7] and Statecharts [5]; the differences with these two notations lay in the way the processes cooperate and in the paradigm followed for representing the process behaviour.

What said above shows also the differences/relationships with UML [9]: UML is OO, JTN is concurrency oriented; UML is a notation that can be used by many different development processes at different points, JTN is for Java targeted design of systems (companion formal/graphic notations for abstract design and requirement specifications have been developed, see [11,10]); UML is semi-formal (precise syntax including well-formed conditions, semantics by English text), JTN is fully formal (it has a complete formal semantics because it can be easily transformed into a graphic-formal specification of [11], see [3]).

The use of data-types with constructors and of pattern matching in guarded commands come from ML [6], because we think that in many case that could be a compact and clear way to represent the data-types and their operations.

References

1. E. Astesiano and G. Reggio. Formally-Driven Friendly Specifications of Concurrent Systems: A Two-Rail Approach. Technical Report DISI–TR–94–20, DISI – Università di Genova, Italy, 1994. Presented at ICSE'17-Workshop on Formal Methods, Seattle April 1995. 77
2. E. Astesiano and G. Reggio. A Dynamic Specification of the RPC-Memory Problem. In *Formal System Specification: The RPC-Memory Specification Case Study*, number 1169 in LNCS. Springer Verlag, 1996. 77
3. E.Coscia and G.Reggio. JTN: the Reference Manual. Technical report, DISI – Università di Genova, Italy, 1999. 78, 79, 96
4. Gosling, Joy, and Steele. *The Java Language Specification*. Addison Wesley, 1996. 77
5. D. Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8, 1987. 96
6. R. Harper, D. MacQueen, and R. Milner. Standard ML. Technical Report ECS-LFCS-86-2, LFCS-University of Edinburgh, 1986. 82, 83, 96
7. ITU. Z.100 ITU Specification and Description Language (SDL). Technical report, ITU, Geneva, 1993. 96
8. ITU. Z.120: Message Sequence Chart (MSC). Technical report, ITU, Geneva, 1993. 82, 91
9. RATIONAL. UML Notation Guide Version 1.1. Available at `http://www.rational.com/uml/html/notation/`, 1997. 96
10. G. Reggio. A Method to Capture Formal Requirements: the INVOICE Case Study. In *Int. Workshop Comparing Specification Techniques*. Universite de Nantes, 1998. 77, 96
11. G. Reggio and M. Larosa. A Graphic Notation for Formal Specifications of Dynamic Systems. In *Proc. FME 97*, number 1313 in LNCS. Springer Verlag, 1997. 77, 78, 79, 80, 85, 95, 96