A Software Architecture for Programming Robotic Systems based on the Discrete Event System Paradigm

Antonio C. Domínguez-Brito[1]*, Magnus Andersson[2]†
and Henrik I. Christensen[2]‡

CVAP244, Tech. Rep. ISRN KTH/NA/P–00/13–SE
Centre for Autonomous Systems, KTH (Royal Institute of Technology)
S-100 44 Stockholm, Sweden, September 2000

[1]University of Las Palmas de Gran Canaria, Spain
[2]Centre for Autonomous Systems, KTH, Sweden

Abstract

Transfer and reuse of software designed specifically to control robotic systems is difficult, and often apparently impossible due to the diversity of hardware and software typically involved. Software reuse is not only a problem of robotic systems. In other fields, as business software, "de facto" standard tools to define software components can be found, and a supplier component software industry even exists. On the contrary in the robotic field there are not any established standards to address this problem. The present work has been addressed to establish grounds to conceive what could be a feasible concept of software component for robotics systems.

Software for control robotic systems may be very heterogeneous, involving numerous devices and software, but, from a generic point of view, it can be considered as a network of weakly coupled parallel and/or concurrent active entities – processes or threads – interacting asynchronously among them in some way. In this document is presented a software model which identifies these active entities with software components and defines their interaction, modeling such entities as port automata, and their interaction through a small set of operators taken from process algebra. This software framework is presented along with a prototypical example, an obstacle avoidance behavior for a mobile robot, illustrating that transfer and reuse of code is possible using this software architecture.

*acdbrito@dis.ulpgc.es
†sungam@nada.kth.se
‡hic@nada.kth.se

1 Introduction

Transfer and reuse of software designed specifically to control robotic systems is difficult, and often apparently impossible due to the diversity of hardware and software typically involved. Development from scratch is not an uncommon situation in many systems, even at the same laboratory. Other times an important software integration effort must be done to reutilize software.

Software reuse is not only a problem of robotic systems. In other fields, as business software, "de facto" standard tools to define software components can be found, – e.g., ActiveX from Microsoft – and a supplier component software industry even exists. On the contrary in the robotic field there are not any established standards to address this problem.

During last years a boosting of hardware features along with a decrease of prices has created a big demand for day-to-day robotics applications, and has evidenced the lack of standard tools to facilitate the design and implementation of robotics systems, and also to define software components for robotic systems.

A software component should be something like an electronic component or chip in electronic industry. It is many years that off-the-shelf chips can be bought and deployed in other parts of the world. Each component has a clear functionality and a well established external interface. Furthermore, numerous standard tools exist to design electronic devices based on the composition, assembly and combination of these electronic components. A similar panorama would be desirable for the robotic industry.

The present work has been addressed to establish grounds to conceive what could be a feasible concept of software component for robotics systems, and once ideas were conceived, to try to put them into practice.

Previous work has been carried out on software architectures being able to grasp the inherent features of robotic systems, and in turn, to map systems designs into working implementations. A large research effort has been devoted to hybrid architectures for autonomous mobile robots – for example, **ISR** [1], **AuRA** [2] [3], **RAP** [4], **ATLANTIS** [5], **Saphira** [6], and **G^{en}oM** [7] –, which have usually three layers: the bottom layer or reactive layer, the intermediate layer or task control layer and the top layer or deliberative layer. The reactive layer is the closest to the hardware, so it deals directly with sensors and actuators, and tries to embody system behaviors. Typically, the behaviors correspond to software modules or a sort of combination of them. The second layer is a sequencer of behaviors in the lower layer. The task execution layer is in charge of initiating, combining, and monitoring behaviors to achieve tasks defined in terms of reactive layer behaviors. Last layer, the deliberative one, is usually responsible for long-term deliberative planning, where plans are defined in terms of task carried out by the second layer.

The mentioned hybrid architectures concern with layers interaction, behaviors integration, sequencers and planners. The present work must be considered in this context, since it has been motivated by one of these architectures – **ISR** [1] –, but it is intended to look for a concept of a generic software component for robotic systems, giving robotic developers the capacity to design and implement systems through combination, assembly and composition of them, and at the same time, being able to grasp the inherent particularities of robotic software, that is, ability to deal with different hardware and software. Such components would just be another level of abstraction between design and implementation, in developer hands, to implement robotic systems.

Typically, robotic systems software can be seen as a network of parallel and/or concurrent active entities, processes or threads, interacting asynchronously among them in some way. A software model which identifies these active entities with software components, and defines their interaction, is presented in this document. In this model, these active entities are weakly coupled, so the global system behavior is the result of the interaction among the entities, and also of the local behavior of these ones, therefore, once the functionality of each particular entity has been defined, the global control scheme resides on how these entities interact on each specific system, and thus, on the configuration of its network of parallel and/or concurrent active entities. The concept to model these active entities has been taken from the Discrete Event Systems – **DES** – framework [8]. Each entity is embodied as a thread, and modeled as a Port Automaton [9] [10] [11]. In this framework, these entities has been called simple DESs or just DESs, and their combination and interaction defines the behavior of the whole system. A small set of operators taken from process algebra [12] [13], has been used to formalize compositions of DESs, such assemblages define what has been called compound DESs, and can be used as DESs in other assemblages or combinations.

Another port automata based software architecture has been developed at CMU [14] [15]. It was mainly addressed to achieve reconfigurability and software reutilization for real time systems, concretely, a reconfigurable robotic arm. It defines a software component, the port object, relies on services of a specific real time operating system, Chimera [16] [17], for software assembly, and was designed for a particular hardware set: real time processing units and VME buses. The work presented in this paper pursuits has also defined a software component based on port automata theory, but there are no assumptions about a specific operating system with a particular set of services relying on a especial hardware set. The only assumption is that the operating system must support multithreading.

Next section, section 2, exposes briefly the motivations and goals. Section 3 provides the formal and theoretical grounds which conform the conceptual model of the software architecture, and at the same time, introduces

a prototypical example which will be used along the rest of the document. Section 4 illustrates the software architecture itself through its use with the example previously introduced. Finally, on section 5, conclusions are exposed, and current software framework limitations and probable future trends are indicated.

2 Motivations and Goals

Reutilization and deployment of software for robotics systems should be as easy as buying electronic components to make your own electronic designs. A similar panorama can be found in business computing, where a software components industry is rising.

The former paragraph resumes the motivations and goals of the work presented here, and as starting goals to understand the insights of the problem, the following objectives were established:

- Formalizing and devising a concept of software component, specific to robotic systems, which should be reusable and deployable, defining deployable as software which can be transferred and statically added to any project, meaning that the software can be transferred to another system at linking time without having to add new "glue" code [14] to interface the component to the rest of the system, obtaining the same functionality that was achieved when it was created and tested first.
- Formalizing and devising combinations or assemblages of these software components being possible that such combinations can be reutilized, deployed and combined in the same manner that simple components can be.

Thus, the aim is to design a software architecture which is able to embody in implementations the concepts expressed in the former two points, and also able to test such ideas in real robotic systems.

3 The Conceptual Model

As it was mentioned earlier in section 1, the introduction, the software typically involved in the control of robotic systems may be very heterogeneous, involving numerous hardware devices and software. Such a heterogeneity could be abstracted through a model of interaction among the different elements composing the system, thus, from this point of view, a robotic system might be considered as a network of weakly coupled parallel and/or concurrent active entities – processes or threads – interacting asynchronously among them in some way. This interaction among entities and the local behavior of each one define tasks to achieve by each specific robotic system.

This concept of active entity has been identified as the software component to be modeled.

To formalize this weakly coupled parallel/concurrent active entity, as a software component, the concept of port Automaton [9] has been used. Furthermore, to achieve tasks it is necessary to combine automata conforming an automata network. The concept of port automaton establishes a clear external interface, its ports, for external interaction among port automata, but, it is also needed a formal framework to express how these automata relate each other in run time. A small set of operators has been taken from process algebra [12] [13] to define a composition of port automata. These operators allow us to assert that the composition is also a port automaton, as its components.

Two key concepts have been devised to map port automata and their combinations to real software implementations: the concept of a simple DES – Discrete Event System – or DES which models a port automaton, and what will be defined as a Compound DES, which formalizes a composition of DESs and/or another Compound DESs.

These two key concepts will be presented in next subsections, and at the same time, to illustrate how this model can be used in a real system, an example will be presented. Concretely, the example will show how an obstacles avoidance behavior for a mobile robot might be implemented using this conceptual framework.

3.1 DESs

A Simple DES defines a parallel or concurrent active entity – a process or a thread – as a Port Automaton [9], i.e., a Finite State Automaton – FSA – that uses ports for all external communication. This FSA executes asynchronously transiting among states as a result of its own activity or upon reception of events/signals through its input ports.

Thus the port automaton concept establishes a clear distinction between the internal functionality of an active entity – its FSA – and its external interface – its ports –.

From [9] and [13], a port automaton P can be formally defined as a generator $G = (L, Q, \tau, \delta, \beta, X, Y, F)$, where:

- L is the set of ports.
- Q is the set of states.
- $\tau \subseteq Q$ is the set of initial states.
- $X = \{X_i : i \in L\}$, where X_i is the input set for port i.
- $Y = \{Y_i : i \in L\}$, where Y_i is the output set for port i.

- $\delta : Q \times \sqcup_{i \in L} X_i \to Q$ is the transition map, where $\sqcup_{i \in L} X_i = \{(x, i) : x \in X_i\}$ is the disjoint union of the X_i's.
- $\beta = \{\beta_i : i \in L\}$, where $\beta_i : Q \to Y_i$ is the output map for port i.
- $F \subseteq Q$ is the set of final states.

All subject to the axiom that for each $q \in Q : \{x \in X_i : \delta(q, (x, i)) \neq \emptyset\} = \emptyset$ or X_i assuring that, in any state $q \in Q$, for any port i, either all elements of the input set X_i will be capable of being accepted or none of them will.

In our framework, $L = L_i \cup L_o$ such that $L_i \cap L_o = \emptyset$, where L_i is the set of input ports, and L_o is the set of output ports. A port packet is defined as an information unit which can be received through an input port, and/or issued through an output port. So, elements of sets X_i's and Y_i's, corresponding to each input and output port respectively, will be all possible port packets received or issued through such ports. Port packets are also classified in types of ports packets, and only one type of port packet can be associated with each input or output port.

Figure 1 depicts a port automaton, a DES, from an external point of view, where the port automaton is represented by a circle, input ports by arrows oriented towards the circle, and output port by arrows oriented outwards. This figure depicts how the internals of a port automaton are isolated from outside by using the mechanism of ports. Figure 2 shows the internal view of a generic DES, the circles are the states of the automaton and the arrows, transitions among its states based on port packets received through its input ports. In the figure p_{ij} denotes that a port packet j of type i has been received through an input port. The figure does not illustrate the automaton functionality, only displays its states and transitions.

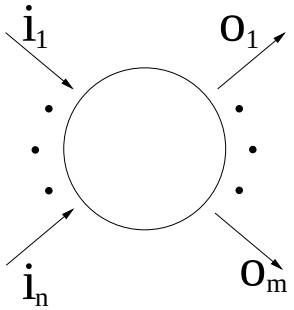

Figure 1: The external view of a DES (the circle): input and output ports (the oriented arrows).

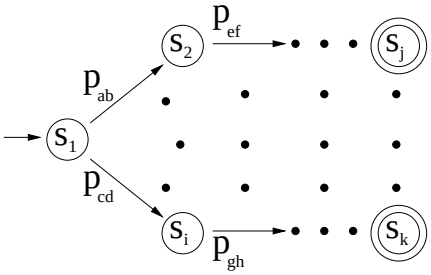

Figure 2: The internal generic view of a DES: states (circles) and transitions (oriented arrows) provoked by port packets. Double circles mean final states.

3.1.1 Default Ports, Default States and the Default Automaton

The model establishes two default ports and six default states for all DES. The default ports are: the control port, **c**, and the monitoring port **m**. The default states are: **idle**, **running**, **abort**, **success**, **fail** and **dead**.

Figure 3 depicts the default ports. The control port **c** is an input port used to force a state in a DES. The states which can only be forced externally are **running**, **abort** and **dead**. Control packets are the kind of port packets received in this port. The monitoring port **m** is an output port used by each DES to indicate its internal state changes. Each time a state change happens in a DES a monitoring port packet, indicating this change, is emitted through this port. Therefore, which these two ports, control and monitoring of each DES is possible.

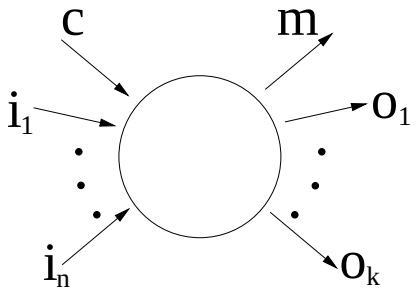

Figure 3: The default ports. The control port **c** and the monitoring port **m**.

In our framework each DES is a thread and is modeled through the same automaton structure, the **Default Automaton**, as shown in figure 4. The **idle** state is the starting state. There, the thread corresponding the DES has been launched and resources has been allocated for it, but is suspended waiting on its control port. The **running** state, in dashes in the figure, represents the not yet defined part of the automaton which is really in charge of giving functionality to each DES – later referred to as the **user automaton** –. This state just depicts the states and transitions that should be established by the developer/user. To go into **running** state a running control packet, $\mathbf{c}_r$, must force the state change. When a DES in **running** state is interrupted or aborted, the DES makes a transition to the **abort** state. An abort control packet, $\mathbf{c}_a$ can only force this change. A DES which finishes successfully its work, goes to **success** state, the transition to this state must be done by the DES itself. A **fail** state is reached when the DES fails its task, and as with the **success** state, the change must only be an own initiative of the DES itself. The two former states can not be forced using the control port. From **idle**, **abort**, **success** and **fail** states the DES is suspended waiting on its control port keeping thread resources. From

these states, a running control packet makes the DES get into **running** state, and a dead control packet, $\mathbf{c}_d$, brings it to **dead** state which means resources release and self destruction, and is the unique final state.

How it was established at the beginning of section 3.1, in our model of discrete event systems, events are modeled as port packets, so internal events, i.e. transitions originated by the DES itself, are also modeled by port packets, except that, in this case, the involved input and output ports are only used for internal communications. In particular, each modeled internal event will have associated an input port and an output port, thus, when the DES itself provokes the internal event, it sends out its corresponding port packet through its associated output port and will receive the same port packet through the associated input port, completing, in such a way, a transition. Examples of these kind of events are the transitions to **success** and **fail** states in figure 4. The ports associated with port packets corresponding to internal events are not shown in automaton figures along this document, but it is assumed that all internal events are modeled like this.

Thus, a DES is a generator G, where by default $\mathbf{c} \in L_i$, $\mathbf{m} \in L_o$, $\{\mathbf{idle}, \mathbf{running}, \mathbf{success}, \mathbf{abort}, \mathbf{fail}, \mathbf{dead}\} \subset Q$, $\tau = \{\mathbf{idle}\}$, $\{\mathbf{c_r}, \mathbf{c_a}, \mathbf{c_d}\} \subset X$, is the input set for $\mathbf{c}$, internal port packets **success** and **fail** are also included in X, the monitoring port packets are included in Y, and are the output set for $\mathbf{m}$, $\delta_{default} \subset \delta$ is defined according to figure 4, and $F = \{\mathbf{dead}\}$. The default automaton defines two new sets of states:

- $S = \{\mathbf{success}\} \subset Q$, is the set of functional successful states, which means that the automaton has finished its task successfully.
- $U = \{\mathbf{abort}, \mathbf{fail}\} \subset Q$, is the set of functional unsuccessful states, which means that the automaton has ended up its task with unsuccessful results.

In [13] the sets S and U are the sets F_s and F_u, respectively, such that $F = F_s \cup F_u$ where $F_s \cap F_u = \emptyset$. In this framework, $S \cap U = \emptyset$, but S and U are not included in F, because according to figure 4, the automaton does not finish when it reaches the states included in both sets, and it can be brought to **running** state again for a new task execution.

3.1.2 Input and Output Ports, States and Transitions

Non default input and output ports, $\mathbf{i}_i$**'s** and $\mathbf{o}_i$**'s** respectively in figure 3, are user defined, including all its types of port packets. Non default states and transitions among them are also user defined. Formally, for each DES, the following sets are user defined:

- $L_i - \{\mathbf{c}, \mathbf{success_i}, \mathbf{fail_i}\} = \{i_i\}$, $i \in \{1, \ldots, n\}$ is the set of non default input ports.

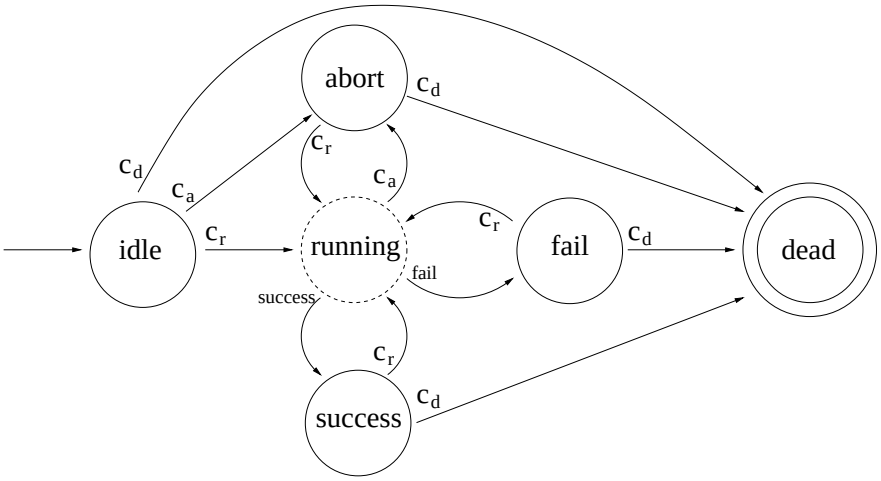

Figure 4: The Default Automaton.

- $L_o - \{\mathbf{m}, \mathbf{success_o}, \mathbf{fail_o}\} = \{o_i\},\ i \in \{1, \ldots, k\}$ is the set of non default output ports.
- $(X - \{\mathbf{c_r}, \mathbf{c_a}, \mathbf{c_d}, \mathbf{success}, \mathbf{fail}\}) \cup (Y - \{$ monitoring port packets $\})$ is the set of non default port packets.
- $Q - \{\mathbf{idle}, \mathbf{running}, \mathbf{success}, \mathbf{abort}, \mathbf{fail}, \mathbf{dead}\}$ is the set of non default states.
- $\delta - \delta_{default}$ is the transition map based on non default input ports, and not established in figure 4.
- and β, the output map.

where $\mathbf{success}_i$ and $\mathbf{success}_o$, and $\mathbf{fail}_i$ and $\mathbf{fail}_o$ are, respectively, the corresponding input and output ports for modeling internal events **success** and **fail**.

3.1.3 Input and Output Parameters

A running port packet, $\mathbf{c}_r$, can transport an **input parameter**. Input parameters are user defined data, and can be used to configure or initialize each DES at the beginning of each task execution. When a DES reaches its successful state, **success**, it sends a monitoring port packet through its port **m**, which can transport an **output parameter**. As input parameters, output parameters are also user defined data for each DES, and can be used as input parameters for other/s DES/s.

3.1.4 DES Examples: the Sensors, an Obstacles Detector and the Avoid DES

Several DESs have been devised to show how to implement an obstacle avoidance behavior for a mobile robot using this framework. This example involves three types of sensors: a belt of sonars, a laser range finder and a stereo robotic head, corresponding each one of them with a DES. Besides, there are another two additional DESs: an obstacle detector and a generator of avoidance trajectories, the avoid DES. All of them will be presented next.

The sensors will be modeled sharing the same DES structure, the **generic sensor**. Figure 5 shows the DES automaton for the generic sensor, the figure only displays user defined states and transitions based on user defined input ports. Notice that what is shown in the figure would be the **running** state in figure 4, the default automaton, which is hosting the DES, that is, the automaton in the figure must be consider into the context of the default automaton. The **running** pseudo state in the default automaton constitutes or represents the part of the automaton which must be completed by the user, which is a sensor in figure 5, and will be referred to as the **user automaton**. That means that from every state in the user automaton a transition to **abort** state in the default automaton is possible, just receiving an abort control packet, $\mathbf{c}_a$, and also, from all of them it is possible to transit to states **success** and **fail**, although in such cases it must be explicitly specified by the user. Additionally, the starting state in the user automaton is the entry state where the default automaton gets into when a running port packet, $\mathbf{c}_r$ is received. All following figures illustrating user automata for several DESs will not show these transitions to default automata states, except when transitions to **success** and **fail** occur, due to these last ones must be specified for each DES, the other ones are assumed by default.

Returning to the generic sensor in figures 5 and 6, there is only a user defined input port, **tick**, which could be a clock tick or an interruption coming from a hardware device which is the sensor. The automaton only has two user defined states: **inactive** and **readandsend**. The **inactive** state is the entry state. Normally resource allocation is localized in the entry state, so a fail during allocation usually provokes a transition to default automaton state **fail**. Once resource allocation is completed successfully, the automaton just wait for a port packet through its input port **tick** to transit to **readandsend**. In the **readandsend** state the automaton collects information from the associated sensor device, then, this information is packed in port packets and sent out through its output port **sense**. During this sensory data collection might happen a fail on the sensor which would cause a transition to the **fail** state in the default automaton. This DES never goes to the default automaton state **success** because it has a continuous operation without a specific goal, it only has to process sensory data, hence, to finish it, it must be aborted using an abort control packet $\mathbf{c}_a$. Figure 6

displays the external view of the DES **sensor** which embodies the generic sensor.

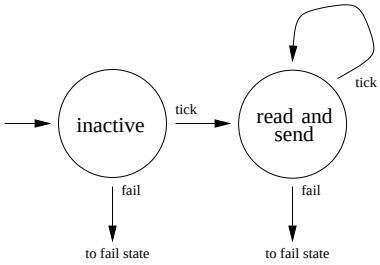

Figure 5: The DES automaton for the generic sensor. The default automaton and the control port are not shown.

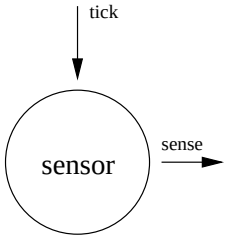

Figure 6: The generic DES **sensor** which implements the generic sensor. Control and monitoring ports are not shown.

All sensors involved in the avoidance behavior share the same DES structure that the generic sensor depicted in figures 5 and 6, and they are: the DES **sonarsensor** modeling the belt of sonars, the DES **lasersensor** modeling the laser range finder, and the DES **visionsensor** modeling the stereo robotic head cameras.

Figure 7 shows the DES automaton for an obstacles detector based on information which comes from sensors modeled as the generic sensor presented in previous paragraphs. The **inactive** state is homologous to the state with the same name in figure 5. It is also an entry state, and resource allocation is carried out when the automaton enters into this state first, so, a transition to default automaton state **fail** is possible. Once resource allocation has been done the state **inactive** is a doing-nothing state, just waiting for sensory information. The automaton also goes into this state when free space is detected, which is indicated by port packets on its input port **freespace**. When sensory information gets into through the input port **sense**, the automaton enters into its **detect** state, where obstacles are detected based on sensory information, issuing obstacle detections through its output port **obstacles**. If nothing is detected, a freespace port packet is issued through its output port **freespace**, which is normally connected to its synonymous input port **freespace**. This DES is also an automaton in continuous operation, so, it does not have any transition to the default automaton state **success**. Figure 8 shows an external view of the DES **detect** which embodies the mentioned automaton, only user defined input and output ports are shown. Notice that to combine this DES with sensors modeled like the **generic sensor**, is necessary that its input port **sense** transports the same type of port packet that the one emitted by these sensors through

their output port **sense**, see figure 6, which also implies that all sensors should produce the same kind of port packet on this port.

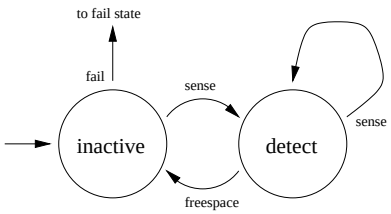

Figure 7: The DES automaton for an obstacle detector. The default automaton and the control port are not shown.

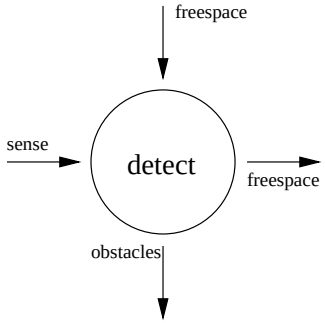

Figure 8: The generic DES **detect** which implements an obstacle detector. Control and monitoring ports are not shown.

Figure 9 shows the DES automaton for obstacle avoidance. It has two states, the **inactive** state which is the entry state, analogous to the state with the same name in figure 5. It also can get into the default automaton state **fail**, if resource allocation fails. Once a successful resource allocation is carried out the automaton waits for detected obstacles port packets, just to get into its second state, the **avoid** state, where avoidance velocities for the mobile robot motors are computed based on obstacle detections received through its input port **obstacles**, determining an obstacle avoidance trajectory for the robot. Then, these velocities are packed and sent out through its output port **velocities**. As previous DESs, the sensor DESs and the obstacle detector, this DES is also a continuous operation automaton without any transition to the default automaton state **success**. Figure 10 depicts the DES **avoid** only showing user defined input and output ports.

3.2 Compound DESs

Once a set of DESs have been defined, instances of these ones may be utilized to conform a network of port automata. A Compound DES, is a composition of instances of DESs and/or another compound DESs. Figure 12 explains graphically this concept, where the compound DES **c** is a composition of two DES instances, one of DES **a**, $\mathbf{a}_i$, and one of DES **b**, $\mathbf{b}_i$, which are shown in figure 11. Figure 13, depicts a compound DES **d** made of an instance of compound DES **c**, $\mathbf{c}_i$, and an instance of DES **b**, evidencing that instances of compound DESs are functionally equivalents to simple DESs in terms of composition and instantiation, so a compound DES is a port automaton

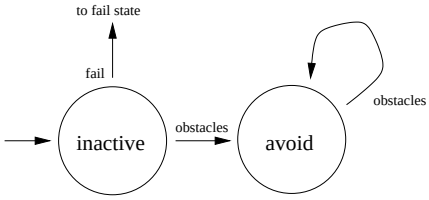

Figure 9: The DES automaton for obstacle avoidance. The default automaton and the control port are not shown.

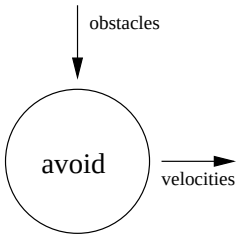

Figure 10: The DES **avoid** which implements the obstacle avoidance automaton. Control and monitoring ports are not shown.

which is a composition of port automata. Control and monitoring ports are not shown.

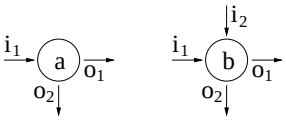

Figure 11: Two DES: **a** and **b**.

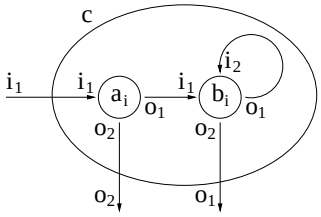

Figure 12: The compound DES **c**: a composition of **a** and **b**.

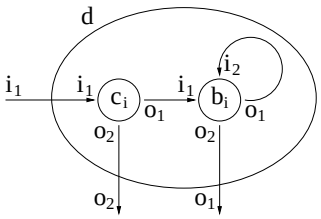

Figure 13: The compound DES **d**: a composition of **c** and **b**.

3.2.1 Execution Operators

A small set of operators has been taken from process algebra [12] [13] to define a compound DES as a composition of DESs and/or compound DESs

instances, these operators allow us to assert that the compound DES is also an automaton, as its components, and have been called **execution operators**. In the following definitions, when it is said that a DES instance finishes, it is in terms of task finalization, that is, the DES instance has reached a state included in the sets S or U. Also, a DES instance is said that is successful when it reaches the **success** state in figure 4, is aborted when it gets into **abort** state and fails when it goes to **fail** state in the same figure. In terms of composition, when a DES is used, it stands for a DES or for a compound DES, indistinctly.

- **Sequential Operator** What is known as **sequential composition**, and is represented by the symbol '**;**'. Let **a** and **b** be two DESs, then the compound DES **c=a;b** is such that an instance of **c**, $\mathbf{c}_i$, behaves like an instance of **a**, $\mathbf{a}_i$, until this one finishes, then behaves like an instance of **b**, $\mathbf{b}_i$. When $\mathbf{b}_i$ finishes, $\mathbf{c}_i$ finishes with the same state as $\mathbf{b}_i$. If $\mathbf{a}_i$ is aborted then $\mathbf{c}_i$ is also aborted.

- **Conditional Operator** What is known as **conditional composition**, and is represented by the symbol '**:**'. Let **a** and **b** be two DESs, then the compound DES **c=a<v>:b(v)** is such that an instance of **c**, $\mathbf{c}_i$, behaves like an instance of **a**, $\mathbf{a}_i$, until this one finishes successfully computing the output parameter **v**, then behaves like an instance of **b**, $\mathbf{b}_i$, which uses **v** as its input parameter. When $\mathbf{b}_i$ finishes, $\mathbf{c}_i$ finishes with the same state as $\mathbf{b}_i$. If $\mathbf{a}_i$ finishes unsuccessfully, i.e., it fails or is aborted, $\mathbf{c}_i$ finishes with the same state as $\mathbf{a}_i$.

- **Concurrent Operator** What is known as **parallel composition**, and is represented by the symbol '|'. Let **a** and **b** be two DESs, then the compound DES **c=a|b** is such that an instance of **c**, $\mathbf{c}_i$, behaves like an instance of **a**, $\mathbf{a}_i$, and an instance of **b**, $\mathbf{b}_i$ running in parallel – or concurrently –, and the state of the composition is a state pair which combines the states of both instances – see [12] for details –, $\mathbf{c}_i$ finishes with the same state as the last finished instance, either $\mathbf{a}_i$ or $\mathbf{b}_i$.

- **Disabling Operator** What is known as **disabling composition**, and is represented by the symbol '#'. Let **a** and **b** be two DESs, then the compound DES **c=a#b** is such that an instance of **c**, $\mathbf{c}_i$, behaves like an instance of **a**, $\mathbf{a}_i$, and an instance of **b**, $\mathbf{b}_i$ running in parallel – or concurrently –, and its state is the state pair conformed by the states of both instances, $\mathbf{c}_i$ finishes with the same state as the first finished instance, either $\mathbf{a}_i$ or $\mathbf{b}_i$, the not yet finished instance is aborted.

Thus, a composition of DESs can be established based on this four operators, so, for example, let **a**, **b**, **c**, **d** and **e** be five DESs, a compound DES **f**

can be defined as $\mathbf{f} = \mathbf{a}<\mathbf{v}> : ((\mathbf{b}\#\mathbf{c})(\mathbf{v}) ; (\mathbf{d}|\mathbf{e})(\mathbf{v}))$, note that **v** must be the input parameter for **b**, **c**, **d** and **e**.

3.2.2 The DES Executor

Once a compound DES has been defined as a composition of other DES and/or compound DES instances, when it is instantiated, an instance of a DES provided by the architecture, the **DES Executor** is in charge of control and monitoring the composition during execution. Figure 14 shows how a **DES Executor** instance, **exe**, use the control and monitoring ports of DES and compound DES instances, $\mathbf{d}_i$'s, inside the compound DES. It disposes of its $\mathbf{c}_i$'s output ports for controlling each DES or compound DES instances, and its $\mathbf{m}_{exe}$ input port for monitoring all of them. Additionally, its control and monitoring ports, **c** and **m**, constitute the control and monitoring ports of the compound DES instance, therefore, it is also in charge of tracking the state of the whole composition depending on how it has been defined in terms of execution operators. Any DES may have an input parameter and/or and output parameter, so a composition may have one or both of them too, thus, the **DES Executor** will inherit an input and/or an output parameter depending on how it has been defined in terms of execution operators and on which DES and/or compound DES instances conform the composition.

3.2.3 Inner Mapping and Outer Mapping

To define a compound DES, besides of indicating which DES and/or compound DES instances are involved, and how these instances are related through the execution operators, it is also necessary to specify how component DES' ports are connected internally in the compound DES, port mapping that will be referred to as the **inner mapping**, , and furthermore, what instances ports are visible to external DES or compound DES instances, the **outer mapping**. That is shown in figures 12 and 13, where control and monitoring ports are not shown.

Connections among ports are not restricted in number, so an input port can be connected to zero o more output ports, and an output port can be connected to zero or more input ports. The only restriction is that the ones involved in a connection should carry the same type of port packets, source ports should be output ports and destination ports should be input ports.

3.2.4 A Compound DES Example: the Avoidance Compound DES

Now the obstacles avoidance behavior for a mobile robot is synthesized through the composition of DESs presented previously in section 3.1.4 using the concept of compound DES.

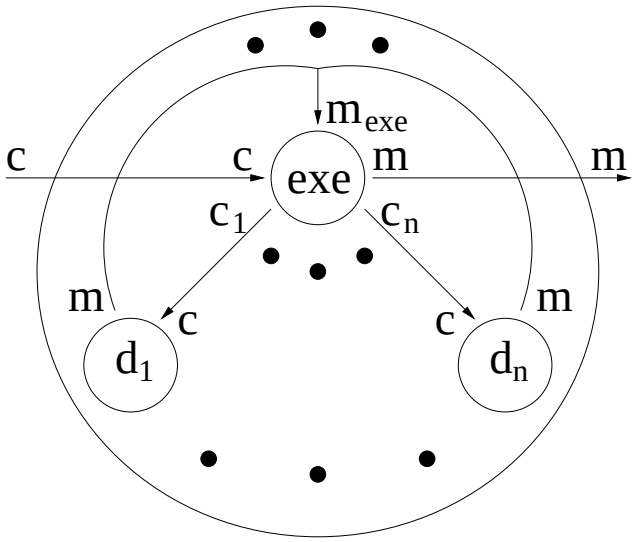

Figure 14: Control and monitoring port connections in a compound DES. The DES Executor instance, **exe**, controls and monitors the DES and compound DES instances, $\mathbf{d}_i$'s, inside a compound DES. It uses its $\mathbf{m}_{exe}$ port to monitor all of them, and its $\mathbf{c}_i$'s ports to control each one. Its control and monitoring ports, **c** and **m**, constitute the control and monitoring ports of the compound DES instance. User defined input and output ports are not displayed.

Figure 15 depicts the compound DES **avoidance** which performs obstacles avoidance using instances of the different DESs introduced in section 3.1.4: the sensory DESs – the **sonarsensor**, the **visionsensor** and the **lasersensor** –, the obstacles detector DES – **detect** – and the obstacles avoidance DES – **avoid** –. Thus, this compound DES implements an obstacles avoidance behavior based on sensory information coming from three types of sensors.

Once the compound DES **avoidance** and its different components have been implemented and tested, it may be used alone or as a component in another compound DES/s. As an example, in figure 16 is shown how **avoidance** might be utilized in an another compound DES, **gotowithavoidance**, where it is combined with another DESs or compound DESs. The compound DES **gotowithavoidance** is a behavior allowing a mobile robot to navigate to a specific place performing obstacles avoidance along a trajectory. As was said in section 3.1.4 the ticks input port packets for the sensors would be generated by timers or sensor device interruptions. It has been assumed in **gotowithavoidance** that the DES **servo** accesses directly to motors status, otherwise feedback between this DES and the DES **motors** in figure 16

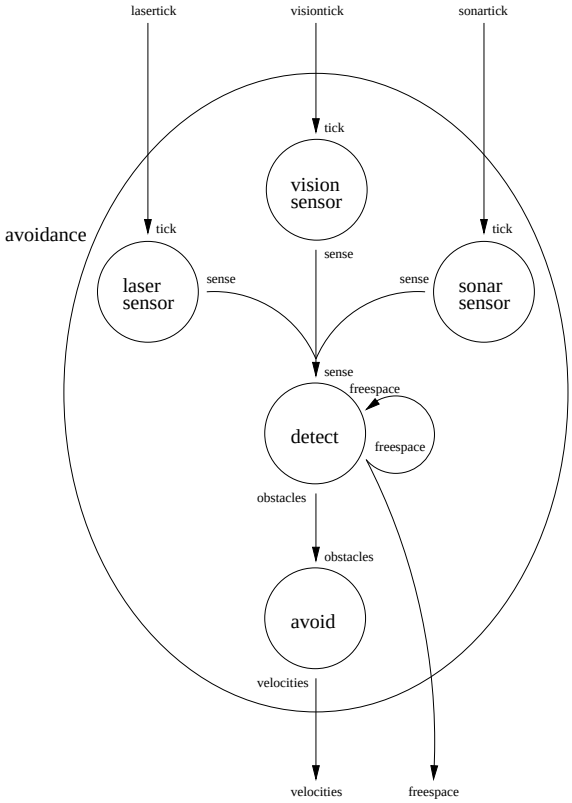

Figure 15: The compound DES **avoidance**. The DES Executor instance, and the control and monitoring ports are not shown.

should be necessary.

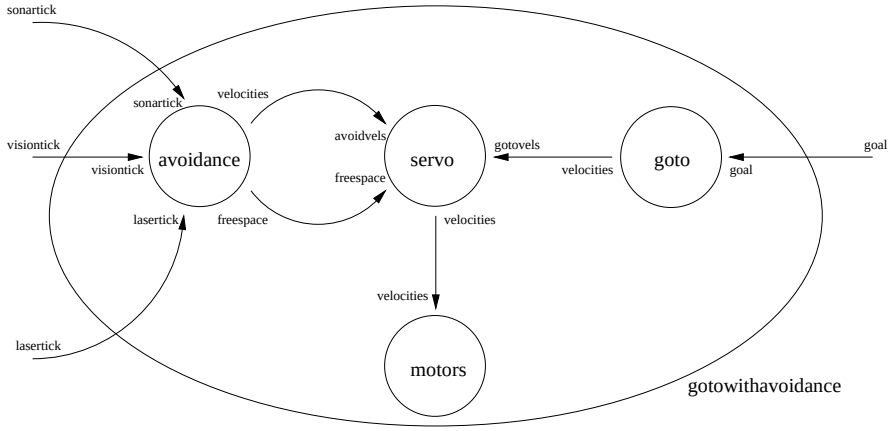

Figure 16: The compound DES **gotowithavoidance**. An example of using the compound DES **avoidance**.

4 The Software Framework

The conceptual model introduced in previous section 3, has been put into practice developing a software framework which allows developers/users to map DES and compound DES definitions to real implementations.

This software architecture provides two levels of abstraction:

- **A Compiler.** The **DES Compiler**, **desc**, generates code, consisting on Java subclasses defined in the context of a hierarchy of Java classes, the **DESpkg**, which implements DESs, compound DESs and associated data types – port packets and input and output parameters – based on specific description code for each particular robotic system defined through a description file – a .des file –.
- **A Hierarchy of Java Classes.** The software model provides a hierarchy of Java classes, the **DESpkg**, where the concepts of DES and compound DES, their default behavior and associated data have been implemented. This hierarchy of classes provides super classes to implement simple DESs, compound DESs, port packets and input and output parameters, according to definitions established in the preceding section 3.

The description code accepted by the compiler **desc** through .des files will be the higher level of abstraction, and the classes hierarchy **DESpkg** constitutes the lower level. Thus, in short words, as a first step, the developers/users create a .des file describing DESs and compound DESs to control a specific robotic system, then, apply the desc compiler to obtain a set of Java classes which will embody these DESs and compound DESs as subclasses inside the hierarchy of classes, **DESpkg**, that implement the default functionality for all of them. Finally, the developers/users will have to finalize the implementation of such subclasses completing with Java code their non default functionality.

Along the next sections, these two abstraction levels will be presented in more detail using the same example, the obstacles avoidance behavior for a mobile robot, already introduced.

4.1 The DES Compiler

The **DES Compiler**, **desc**, allows developers/users to define the software skeleton to control a robotic system. The compiler accepts a description code, a .des file, to define such a skeleton in terms of DESs and compound DESs, then, generates a set of Java subclasses, immersed in a hierarchy of Java classes – the **DESpkg** –, mapping that description code in Java shell classes that, then, must be completed by developers/users in order to achieve an operative system.

4.1.1 The desc Code

The description code accepted by the compiler, the desc code, will be shown next through the example already introduced along section 3, the obstacles avoidance behavior for a mobile robot.

As it was said in section 3.1.4, all sensors involved in the obstacles avoidance behavior share the same DES structure that the generic sensor depicted in figures 5 and 6. Figure 17, shows the DES description code for one of the sensors, the belt of sonars. Figures 18 and 19 display the desc code for the other involved sensors, the laser range finder and the cameras on the robotic head. Notice that the desc code is the same for all of them, except the DES name, which is **sonarsensor** for the sonars belt, **lasersensor** for the laser range finder and **visionsensor** for the cameras, due to they have in common the same DES structure. Also observe how all of them use the same type of port packet on their different ports, **CTick** on the input port **tick**, and **CMap** on the output port **sense**.

```
des sonarsensor(none,none) /* no input parameter,
                              no output parameter */
{
   input ports
   {
      tick: circular, CTick, 2;
   };

   output ports
   {
      sense: CMap;
   };

   entry state inactive
   {
      transition in tick;
   };

   state readandsend
   {
      transition in tick;
   };
};
```

Figure 17: The desc description for the sonar sensor.

Figures 17, 18 and 19 show us how desc code to define a typical DES looks like. The code describes the DES, in terms of input and output ports, and states. And for each state, establishes which input port is activated and when state transitions are possible. The framework implements input ports in three ways: a circular buffer of port packets, a FIFO – a queue – of port packets, and a growing FIFO of port packets. For each input port, its type, length and port packet must be specified. Notice that one of the states should be the **entry** state, where the default automaton enters when goes to **running** state, shown in figure 4. Input and output parameters can

```
des lasersensor(none,none) /* no input parameter,
                              no output parameter */
{
   input ports
   {
      tick: circular, CTick, 2;
   };

   output ports
   {
      sense: CMap;
   };

   entry state inactive
   {
      transition in tick;
   };

   state readandsend
   {
      transition in tick;
   };
};
```

Figure 18: The desc description for the laser sensor.

```
des visionsensor(none,none) /* no input parameter,
                               no output parameter */
{
   input ports
   {
      tick: circular, CTick, 2;
   };

   output ports
   {
      sense: CMap;
   };

   entry state inactive
   {
      transition in tick;
   };

   state readandsend
   {
      transition in tick;
   };
};
```

Figure 19: The desc description for the vision sensor.

also be indicated, if any, DESs shown in the mentioned figures do not use them. Comments may be added using C standard notation for comments.

Observe that nothing related to the default automaton, its states and transitions, or related to the control and monitoring ports, **c** and **m**, appears in the description code. All this is transparently added by the compiler to the Java subclasses which will be generated. Nothing is also declared

about what happens inside each state, that is, its inner functionality, mainly, when a state change happens – a state transition –, and when output port packets must be emitted through output ports. All this is part of the specific functionality of each DES, and must be completed by the developer/user after compilation in the Java subclasses generated by the compiler.

The desc code for the DES **detect** corresponding to figures 7 and 8, is shown in figure 20. It does not have input and output parameters either. Note that its input port **sense** accepts the same type of port packet, **CMap**, which is issued by the sensors – the sonarsensor, the lasersensor, and the visionsensor – on their respective **sense** output ports.

```
des detect(none,none) /* no input parameter,
                         no output parameter*/
{
   input ports
   {
      sense: circular, CMap, 4;
      freespace: circular, CFreeSpace, 2;
   };

   output ports
   {
      obstacles: CObstacles;
      freespace: CFreeSpace;
   };

   entry state inactive
   {
      transition in sense;
   };

   state detect
   {
      transition in sense;
      transition in freespace;
   };
};
```

Figure 20: The desc description for the obstacles detector.

The **avoid** DES, figures 9 and 10, has the desc code displayed in figure 21. As the **detect** DES, it does not have input and output parameters either.

Figure 22 shows the desc code for the **avoidance** compound DES, shown in figure 15. In this description, DES and/or compound DES instances must be specified. Furthermore, the inner mapping – local compound DES connections among inputs and outputs ports – and the outer mapping – input and output ports of the whole composition – may be indicated. Finally, the combination among DES instances should be explicitly established. Concretely, a concurrent composition among the three types of sensors has been specified, and this concurrent composition is combined in a disabling way with the detector DES and the obstacles avoidance DES, see section 3.1.4.

```
des avoid(none,none) /* no input parameter,
                        no output parameter*/
{
   input ports
   {
      obstacles: circular, CObstacles, 2;
   };

   output ports
   {
      velocities: CVelocities;
   };

   entry state inactive
   {
      transition in obstacles;
   };

   state detect
   {
      transition in obstacles;
   };
};
```

Figure 21: The desc description for the **avoid** DES.

In this way, the avoidance behavior works in the worst case, when only one of the sensors is operative, and in the best case when all of them are. If the obstacles detector DES, or the avoid DES finishes – either aborted, or with success, or unsuccessfully –, the behavior will be finished too. Observe that nothing related with control and monitoring ports, and the DES Executor is indicated in this code, because it is part of the default behavior for each compound DES, and as with DESs, it is also transparently added by the compiler to the Java subclasses that are generated.

4.1.2 Compiler Verifications

During compilation the DES compiler **desc** performs a set of verifications on the code, and when any of them is not fulfilled, the violation is notified and the compilation is aborted. The following summary resumes these verifications.

On each DES, it verifies that:

- There is not a reuse of names for input and output ports and states, i.e., if any input port, output port or state has been redefined.
- References to input ports are correct in state statements, that is, if there is any input port reference which has not been defined.
- All DESs must have one and only one entry state.
- There is not an idle state, that is, a state without a cycle and without transitions. A cycle in a state allows functionality in this state when

```
compound des avoidance
{
   instances
   {
      sonarsensor sonarsensor1;
      visionsensor visionsensor1;
      lasersensor lasersensor1;
      detect detect1;
      avoid avoid1;
   };

   inner mapping
   {
     from sonarsensor1.sense to detect1.sense;
     from visionsensor1.sense to detect1.sense;
     from lasersensor1.sense to detect1.sense;
     from detect1.freespace to detect1.freespace;
     from detect1.obstacles to avoid1.obstacles;
   };

   outer mapping
   {
      input sonartick: sonarsensor1.tick;
      input visiontick: visionsensor1.tick;
      input lasertick: lasersensor1.tick;

      output velocities: avoid1.velocities;
      output freespace: detect1.freespace;
   };

   execute as  [ ( sonarsensor1 |
                   visionsensor1 |
                   lasersensor1 ) # detect1 # avoid1 ];
};
```

Figure 22: The desc description for the **avoidance** compound DES.

no input port packets are received. When a cycle is defined in a state the developer/user will have a cycle function to fill in for this state in the Java code generated by the DES Compiler, more precisely, in the Java class which will embody the DES containing this state. The cycle feature is not shown in this document.

On each compound DES, it verifies that:

- There is not a reuse of names for instances, and input and output ports, that is, if any instance, or input port or output port has been redefined.
- Instances definitions should be only referred to other defined DESs or compound DESs.
- There is no any kind of recursive definitions of instances, i.e., it is not possible to define a compound DES containing an instance or instances of itself, or containing an instance or instances of compound

DESs including, in turn, direct or indirectly, an instance or instances of this compound DESs.

- References in inner and outer mappings are consistent with instance definitions specified in the compound DES, i.e., if the referred instances, input and output ports have been defined.
- Inner mapping connections are established among compatible input and output ports , i.e., transporting the same type of port packets.
- The execute statement is referred to instances defined in the compound DES, and each instance should be in the execute statement once and only once.
- Output and input parameters match among instances in conditional operators in the execute statement.

Thus, once, a set of developed DESs and compound DESs is available, new assemblages and combinations are easily verified through compilation.

4.2 The Hierarchy of Java Classes DESpkg: The Software Backbone

The Java classes hierarchy **DESpkg** is really the software backbone which implements the concepts established in section 3. For each robotic system described through a .des file, the **desc** compiler generates Java subclasses in the context of this hierarchy of classes, which will constitute its software skeleton, and will have to be completed by the developer/user.

To illustrate how the developer/user should complete the Java code generated by the compiler, and, at the same time, to outline the **DESpkg** set of classes, we will have a look to part of the code generated by the **desc** compiler for the example which has already been introduced along previous sections, the obstacles avoidance behavior for a mobile robot, concretely, the skeleton classes generated for DES **detect** and the compound DES **avoidance**.

4.2.1 The DES detect

For each defined DES in a .des file, the **desc** compiler generates a Java class. As a sample, appendix A shows the Java class generated by the compiler for DES **detect**, corresponding to the desc code depicted in figure 20.

First of all, having a look to the code, observe that, there are a lot of pairs of marks as comments, which may have one out of these three forms:

- either //<->section<->
  and //<->/section<->,

- or //<->section<->state,port<->
  and //<->/section<->state,port<->,
- or //<->section<->state<->
  and //<->/section<->state<->.

Each pair of these marks delimits portions of code which could be modified by the compiler in future compilations, so there, the developer/user does not have to add any code, otherwise it will be lost in the next compilation, if any. On the contrary, all code added by the developer/user situated outside of any of these pairs of marks will be preserved among desc compilations. Thus, if the description code for a DES is modified and compiled, the previous code already added by the developer is not missed, but preserved, if it has been added outside of any of these pair of marks. In general, as it will be seen later, the compiler adds these kinds of marks to all code that it creates, not only to the one corresponding to DESs descriptions, and, the rule is the same, code to be preserved among compilations should be added outside of these pairs of marks. Furthermore, all these marks should be preserved as they have been generated by the compiler, because any mark alone without its partner will cause a compilation error. The compiler does not protect all it generates with these marks, it only protects in this way things which might change among compilations. The compiler also generates code which remains invariant among compilations which is not protected with marks, and which is preserved along successive compilations, but only generated the first time, so it should not be modified by the developer/user, because the compiler does not verify in consecutive compilations if this code has been modified or not. To conclude, as rules which are a must for developers/users when they are completing any class generated by the compiler, the next two rules must be observed:

- Any code generated by the compiler must strictly be preserved without changes, even any comment, and specially, the mentioned pairs of marks.
- Any code added by developers/users must be situated outside of the portions of code delimited by the pairs of marks generated by the compiler.

According to desc code, in figure 20, corresponding to DES **detect**, the compiler has generated the Java class **CDESdetect** which is a subclass of the **DESpkg** class **CDES**; see appendix A, where **CDESdetect** appears as it was generated first. **CDESdetect** constitutes the skeleton to implement the DES **detect**.

In this framework, the functionality of each DES will be coded on the transitions among the automaton states, including the transitions corresponding to the default automaton in figure 4. The desc code for a DES

only specifies for each one of its states what input ports could be listened to, thus, meaning that, only a part of the infrastructure of the DES is constructed by the compiler, its skeleton, the rest must be completed by the developer/user. Specifically, the compiler generates on each state a function to fill in, corresponding to each input port that can be listened to in that state. Figure 23 displays a sample from the code generated for DES **detect**, appendix A, showing the function which should be completed for its state **inactive** corresponding to an input port packet on its input port **sense**.

In the figure the function has already been filled in. Notice how the transition to state **detect** must be specified explicitly. Obviously, the added code has been included outside the marks which define the body of the function generated by the compiler, as it was said in previous paragraphs, just to preserve the code from future desc code modifications.

Figures 24 and 25 show the functions created by the compiler for the state **detect** with each one of the input ports activated in this state according to desc code in figure 20. In these figures and figure 23, the functions used to send an output port packet through an output port and to transit to other state are implemented on the DES super class **CDES**, but other several functions, also used in the code appearing in the figures, are supposed to be implemented in some other place by the developer/user. For example, the function member **ProcessMap(CMap ppCMap, CObstacles ppObstacles)** which performs object detection from sensory data contained on the port packet **ppCMap** and returns the result already packed on **CObstacles**, which is a port packet that may be sent directly through the output port **obstacles**. There are no restrictions to implement data and function members in the skeleton class to complete the expected functionality for a specific DES, obviously, always that, compiler marks are preserved. Observe that in terms of Java, these skeleton classes are also Java classes, so, may be linked without any restrictions with whatever other Java code, for example, a driver for a sensor, a math library, etc.

Functions corresponding to transitions among default automaton states are already implemented in the super class **CDES**, but they are implemented as idle functions – they do nothing –. These functions can be overridden in the subclass, if necessary. To illustrate this extend, the member function _Started() displayed in figure 26 has been overridden in class CDES-detect. Particularly, this function is called when the automaton gets into the entry state of the automaton once a running port packet – $\mathbf{c}_r$ – has been received and the transition to the entry state have been completed, see figure 4. This function was intended as the typical place to write code to allocate the necessary resources to execute conveniently the user automaton, and a fail in this allocation will provoke a transition to the fail state of the default automaton as shown in the figure, otherwise the automaton remains in the entry state continuing execution. In any case, keep in mind that the decision to override a default automaton transition, which typically has an

```
//<->begintransition<->inactive,sense<->
  // Funtion pointer definition for transition sense in state inactive.
  private static class Cinactivesense implements IFunctionPointer
  {
    public void Function(Object oParam)
    {
      CDESdetect thisCDESdetect=(CDESdetect) oParam;
      CMap ppCMap=(CMap) thisCDESdetect._ppCurrentPacket;
      // State inactive: here starts your code
//<->/begintransition<->inactive,sense<->

      // BEGIN: Added code

      // get output port packet CObstacles
      CObstacles ppObstacles=thisCDESdetect._obOutputBox.GetPortPacket(iOP_obstacles);

      // process sensory data
      ProcessMap(ppCMap,ppObstacles);

      if(ppObstacles.IsAnyObstacle()) // Is there any obstacle?
      {
        // send out CObstacles output port packet
        thisCDESdetect._obOutputBox.SendPortPacket(iOP_obstacles);
      }
      else // No obstacles
      {
        // send a CFreeSpace output port packet
        thisCDESdetect._obOutputBox.SendPortPacket(iOP_freespace);
      }

      // transit to detect state
      thisCDESdetect._SetState(iS_detect);

      // END: Added code


//<->endtransition<->inactive,sense<->
      // State inactive: here ends your code
    }
  }
//<->/endtransition<->inactive,sense<->
```

Figure 23: State **inactive** with a port packet on its input port **sense**.

idle default implementation, is up to the developer/user depending on the expected automaton functionality, and all its corresponding code must be explicitly added by him/her to the classes generated by the compiler.

Figure 26 also shows how internal events have been implemented. There, the internal event fail is just the boolean return value of function AllocateResources(). If it were implemented literally – see section 3.1–, the automaton would have to have an extra input port, an extra output port, and an extra type of port packet definition, and besides, each time the event occurs, the automaton should emit a port packet to signal the event occurrence to itself.

Skeleton classes for port packets are also created by the compiler, appendices C.1, C.2 and C.3, respectively, show the classes generated for port packets **CMap**, **CFreeSpace** and **CObstacles**, which are used in DES **de-**

```
//<->begintransition<->detect,sense<->
  // Funtion pointer definition for transition sense in state detect.
  private static class Cdetectsense implements IFunctionPointer
  {
    public void Function(Object oParam)
    {
      CDESdetect thisCDESdetect=(CDESdetect) oParam;
      CMap ppCMap=(CMap) thisCDESdetect._ppCurrentPacket;
      // State detect: here starts your code
//<->/begintransition<->detect,sense<->

      // BEGIN: Added code

      // get output port packet CObstacles
      CObstacles ppObstacles=thisCDESdetect._obOutputBox.GetPortPacket(iOP_obstacles);

      // process sensory data
      ProcessMap(ppCMap,ppObstacles);

      if(ppObstacles.IsAnyObstacle()) // Is there any obstacle?
      {
        // send out the obstacles port packet
        thisCDESdetect._obOutputBox.SendPortPacket(iOP_obstacles);
      }
      else // No obstacles
      {
        // send a CFreeSpace port packet
        thisCDESdetect._obOutputBox.SendPortPacket(iOP_freespace);
      }

      // END: Added code

//<->endtransition<->detect,sense<->
      // State detect: here ends your code
    }
  }
//<->/endtransition<->detect,sense<->
```

Figure 24: State **detect** with a port packet on its input port **sense**.

tect, as they were created first by the compiler, and note how all of them are subclasses of the **DESpkg** class **CPortPacket**. They must implement, at least, a copy method, as these appendices show. But besides, could also implement its own functionality – data and function members –, as the function member **IsAnyObstacle()** for port packet **CObstacles** which is called on functions displayed in figures 23 and 24.

In short, using this software framework, the structure for an automaton defining a simple DES must be completed at this level of filling-in the skeleton, because it is only at this level where primitives to perform states transitions, to make decisions based on the port packets, to send output port packets and to code internal events, are available.

Additionally the compiler will create skeleton classes too, for input and output parameters, if any. All input and output parameters, as other skeleton classes already introduced, are also subclasses of another class, **ISel-**

```
//<->begintransition<->detect,freespace<->
  // Funtion pointer definition for transition freespace in state detect.
  private static class Cdetectfreespace implements IFunctionPointer
  {
    public void Function(Object oParam)
    {
      CDESdetect thisCDESdetect=(CDESdetect) oParam;
      CFreeSpace ppCFreeSpace=(CFreeSpace) thisCDESdetect._ppCurrentPacket;
      // State detect: here starts your code
//<->/begintransition<->detect,freespace<->

      // BEGIN: Added code

      // transit to inactive state
      thisCDESdetect._SetState(iS_inactive);

      // END: Added code

//<->endtransition<->detect,freespace<->
      // State detect: here ends your code
    }
  }
//<->/endtransition<->detect,freespace<->
```

Figure 25: State **detect** with a port packet on its input port **freespace**.

```
protected void _Started()
{
  if( !AllocateResources() ) // Is there anything wrong?
  {
    // Transit to fail state in the default automaton
    _SetFailState();
  }
}
```

Figure 26: CDES class overridden function.

fReplication, which is a Java interface provided by **DESpkg**. Note that this feature has not been used in the example shown in this document.

4.2.2 The Compound DES avoidance

Appendix B shows the Java class **CCoDESavoidance** generated first by the compiler corresponding to the compound DES **avoidance**, figure 15, from its description code in figure 22. **DESpkg** implements compound DESs through derivation of class **CCompoundDES**, as it can be seen in appendix B. Classes generated by the compiler corresponding to compound DESs are not skeleton classes, they do not have to be filled in, they can be used directly as they have been created first by the compiler, but, in any case, the compiler also interleaves pairs of marks indicating where developer/user code can not be added, although, these classes usually do not need to be completed.

5 Conclusions

With respect to the goals expressed in section 2, a software architecture has been devised and built, which establishes a formal definition for a software component for robotic systems, and a formal definition for combinations of these software components in a way that eases its reusability and deployment.

The software framework was put into practice and tested through non real examples, and during evaluation various limitations and ideas arose that were not though initially, and are commented next.

5.1 Current Limitations and Future Work

Future work is very related to current limitations, and the following points express some of these limitations and, at the same time, possible trends to follow in future improvements.

- **Automata Mapping.** The current architecture maps port automata into implementations using two levels of abstraction: the **desc** compiler and the hierarchy of Java classes **DESpkg**. The automaton can only be completed at the last level, **DESpkg**, because primitives to perform state transitions, to make decisions based on the port packets, and to implement internal events are only available at this level. Complete automata mapping at first level, the compiler, might be possible if such primitives were also available at compiler level. Automata mapping at compiler level could allow verifying automaton stability, automaton isolated states, etc, at compilation.

- **Software Deployment.** Normally to complete the skeleton of a DES, it is necessary to link the skeleton class generated by the compiler with some other library or libraries provided by the developer/user, this information is not present anywhere in the architecture at this moment. Software deployment would only be possible if the software architecture disposed of such a information, for example, adding information about linking and libraries in the description code for each DES, for each port packet, for each input and output parameter, and even, for each, compound DES.

- **A Combination Language.** Once a set of DESs and compound DESs has been developed and tested enough, and can be used to construct a complex robotic system, the fact of having a quite static and rigid way of combining DESs and compound DESs arises. It looks necessary to devise a more dynamic and complex way – a combination language – to combine DESs and compound DESs which makes possible to program a robotic system in terms of components, that is,

in terms of DESs and compound DESs, and such a language needs to keep the formality of process algebra, because in this way it is possible to argue that such combinations are also port automata. Besides, just to enumerate, several points might deserve future considerations in this language:

  - **fail recovering:** there should be mechanisms for fail detection and recovering. At this moment there is no mechanism for fail recovering, a fail implies a transition to fail state, and then it is possible to restart execution or just kill the automaton, see figure 4.
  - **timing information:** information about worst working periods for a DES , its priorities, its working latencies, watch-dog timers, etc, and means to module such information dynamically.

- **Porting to C++.** The current architecture implementation in Java classes is quite slow for low level components. Porting it to C++ makes sense as we want to apply it on real robotic systems.

- **Distributed Framework.** At this moment the architecture is not distributed. Robotic systems usually involve multiple computers, future versions should be distributed

- **Tools.** Different tools could be quite usable and valuable, for example, graphical tools as a graphical DES designer and a graphical DES composer, debugging tools as a DES debugger, etc.

6 Acknowledgements

This work was performed during a seven-month stay of one of the authors, Mr. Antonio C. Domínguez-Brito, at the Centre for Autonomous Systems, Royal Institute of Technology, Stockholm, Sweden, from October 1999 to April 2000. The authors would like to thank the institutions which made this stay possible: the **Margit and Folke Perhzon Foundation**, the **Centre for Autonomous Systems** and the **University of Las Palmas de Gran Canaria**, because without their support this work would never have been performed.

References

[1] M. Andersson, A. Orebäck, M. Lindström, and H. I. Christensen. ISR: an Intelligent Service Robot. Lecture Notes in Artificial Intelligence, Heidelberg, Springer Verlag, 1999. Intelligent Sensor Based Robotics, ch. To appear.

[2] R. C. Arkin. Integrating Behavioral, Perceptual and World Knowledge in Reactive Navigation. *Robotics and Autonomous Systems*, 6:105–122, 1990.

[3] R. C. Arkin and T. Balch. AuRA: Principles and Practice in Review. College of Computing, Georgia Institute of Technology, Mobile Robot Laboratory, Atlanta, Georgia 30332, 1997. Report.

[4] R. J. Firby. Task Networks for Controlling Continuous Processes. Second International Conference on AI Planning Systems, pp. 49-54, 1994.

[5] E. Gat. Integrating Planning and Reacting in a Heterogeneous Asynchronous Architecture for Controlling Real-World Mobile Robots. Proceedings of the IAAA Conference, 1992.

[6] K. Konolige, K. Myers, A. Saffiotti, and E. Ruspini. The Saphira Architecture: a Design for Autonomy. *Journal of Experimental and Theoretical Artificial Intelligence*, 9:215–235, 1997.

[7] S. Fleury, M. Herrb, and R. Chatila. G^{en}oM: a Tool for the Specification and the Implementation of Operating Modules in a Distributed Robot Architecture. IROS 97, Grenoble, France. LAAS Report 97244, 1997.

[8] P. J. Ramadge and W. M. Wonham. The Control of Discrete Event Systems. *Proceedings of the IEEE*, 77(1):81–97, 1989.

[9] M. Steenstrup, M. A. Arbib, and E. G. Manes. Port Automata and the Algebra of Concurrent Processes. *Journal of Computer and System Sciences*, 27:29–50, 1983.

[10] D. M. Lyons and M. A. Arbib. A Formal Model of Computation for Sensory-Based Robotics. *IEEE Transactions on Robotics and Automation*, 5(3):280–293, June 1989.

[11] D. M. Lyons. A Process-Based Approach to Task Representation. *IEEE Proceedings Robotics and Automation*, pages 2142–2147, 1990.

[12] J. Košecká. *Supervisory Control Theory of Autonomous Mobile Agents.* PhD thesis, University of Pennsylvania, GRASP Laboratory, February 1996.

[13] J. Košecká, H. I. Christensen, and R. Bajcsy. Experiments in Behavior Composition. *Robotics and Autonomous Systems*, 19:287–298, March 1997.

[14] D. B. Stewart. *Real-Time Software Design and Analysis of Reconfigurable Multi-Sensor Based Systems.* PhD thesis, Carnegie Mellon University, Dept. Electrical and Computing Engineering, Pittsburgh, 1994.

[15] D. B. Stewart, R. A. Volpe, and P. K. Khosla. Design of Dynamically Reconfigurable Real-Time Software Using Port-Based Objects. *IEEE Transactions on Software Engineering*, 23(12):759–776, December 1997.

[16] D.B. Stewart and P. Khosla. Chimera 3.1: the Real-Time Operating System for Reconfigurable Sensor-Based Control Systems. Advanced Manipulators Laboratory, The Robotics Institute and Department of Electrical and Computer Engineering, Carnegie Mellon University, January 1993.

[17] D.B. Stewart and P. Khosla. The Chimera Methodology: Designing Dynamically Reconfigurable and Reusable Real-Time Software using Port-Based Objects. *International Journal of Software Engineering and Knowledge Engineering*, 6(2):249–277, June 1996.

A The DES detect: the CDESdetect class

```
//<->header<->
  /*
   * File: CDESdetect.java
   * Compiled by: DES Compiler v0.1 (desc)
   * Date: Tue 16 May 2000 13:35:53
   */
//<->/header<->

package DESpkg;

//<->definition<->
public class CDESdetect extends CDES {
  private static CInstancesNaming _inanaming=new CInstancesNaming("CDESdetect");
//<->/definition<->

//<->inputportsids<->
  // User defined input ports
  public static final int iIP_sense=1;
  public static final int iIP_freespace=2;
//<->/inputportsids<->

//<->inputportsconfigs<->
  // Input ports configuration data
  public static final CInputBox.CInputPortConfiguration[] aipcIP_PORTS=
  {
    // iIP_CONTROL
    ipcIP_CONTROL,
    // iIP_sense
    new CInputBox.CInputPortConfiguration("DESpkg.CCircularPort",
                                          "DESpkg.CMap",
                                          4),
    // iIP_freespace
    new CInputBox.CInputPortConfiguration("DESpkg.CCircularPort",
                                          "DESpkg.CFreeSpace",
                                          2)
  };
//<->/inputportsconfigs<->

//<->outputportsids<->
```

```
  // User defined output ports
  public static final int iOP_obstacles=1;
  public static final int iOP_freespace=2;
//<->/outputportsids<->

//<->outputportsconfigs<->
  // Output ports configuration data
  public static final String[] asOP_PORTS=
  {
    sOP_MONITORING, // iOP_MONITORING
    "DESpkg.CObstacles", // iOP_obstacles
    "DESpkg.CFreeSpace"  // iOP_freespace
  };
//<->/outputportsconfigs<->

//<->statesids<->
  // User defined states Ids
  public static final int iS_inactive=5;
  public static final int iS_detect=6;
//<->/statesids<->

//<->statesnames<->
  // States names
  private static final String[] _asstateNames=
  {
    asStateNames[iS_IDLE],
    asStateNames[iS_RUNNING],
    asStateNames[iS_SUCCESS],
    asStateNames[iS_ABORT],
    asStateNames[iS_FAIL],
    "inactive",
    "detect"
  };
//<->/statesnames<->

//<->statesmasks<->
  // States masks
  private static final boolean[][] _aabostateMasks=
  {
    // iS_IDLE
    {
      true, // iIP_CONTROL
      false, // iIP_sense
      false  // iIP_freespace
    },
    // iS_RUNNING (not necessary, but it must be a position for this state)
    {
      true, // iIP_CONTROL
      false, // iIP_sense
      false  // iIP_freespace
    },
    // iS_SUCCESS
    {
      true, // iIP_CONTROL
      false, // iIP_sense
      false  // iIP_freespace
    },
    // iS_ABORT
    {
      true, // iIP_CONTROL
      false, // iIP_sense
      false  // iIP_freespace
```

```
    },
    // iS_FAIL
    {
      true, // iIP_CONTROL
      false, // iIP_sense
      false  // iIP_freespace
    },
    // iS_inactive
    {
      true, // iIP_CONTROL
      true, // iIP_sense
      false  // iIP_freespace
    },
    // iS_detect
    {
      true, // iIP_CONTROL
      true, // iIP_sense
      true  // iIP_freespace
    }
  };
//<->/statesmasks<->

//<->statestransitionsdefs<->
//<->begintransition<->inactive,sense<->
  // Funtion pointer definition for transition sense in state inactive.
  private static class Cinactivesense implements IFunctionPointer
  {
    public void Function(Object oParam)
    {
      CDESdetect thisCDESdetect=(CDESdetect) oParam;
      CMap ppCMap=(CMap) thisCDESdetect._ppCurrentPacket;
      // State inactive: here starts your code
//<->/begintransition<->inactive,sense<->


//<->endtransition<->inactive,sense<->
      // State inactive: here ends your code
    }
  }
//<->/endtransition<->inactive,sense<->
//<->begintransition<->detect,sense<->
  // Funtion pointer definition for transition sense in state detect.
  private static class Cdetectsense implements IFunctionPointer
  {
    public void Function(Object oParam)
    {
      CDESdetect thisCDESdetect=(CDESdetect) oParam;
      CMap ppCMap=(CMap) thisCDESdetect._ppCurrentPacket;
      // State detect: here starts your code
//<->/begintransition<->detect,sense<->


//<->endtransition<->detect,sense<->
      // State detect: here ends your code
    }
  }
//<->/endtransition<->detect,sense<->
//<->begintransition<->detect,freespace<->
  // Funtion pointer definition for transition freespace in state detect.
  private static class Cdetectfreespace implements IFunctionPointer
  {
    public void Function(Object oParam)
```

```
    {
      CDESdetect thisCDESdetect=(CDESdetect) oParam;
      CFreeSpace ppCFreeSpace=(CFreeSpace) thisCDESdetect._ppCurrentPacket;
      // State detect: here starts your code
//<->/begintransition<->detect,freespace<->


//<->endtransition<->detect,freespace<->
      // State detect: here ends your code
    }
  }
//<->/endtransition<->detect,freespace<->
//<->/statestransitionsdefs<->

//<->statestransitionsmatrix<->
  // Matrix of transitions (function pointers)  for each state
  private final IFunctionPointer[][] _aafpstateCallbacks=
  {
    // iS_IDLE
    {
      new CDES.CIdleControlPacket(), // iIP_CONTROL
      null, // iIP_sense
      null  // iIP_freespace
    },
    // iS_RUNNING (not necessary, but it must be a position for this state
    {
      new CDES.CEntryControlPacket(), // iIP_CONTROL
      null, // iIP_sense
      null  // iIP_freespace
    },
    // iS_SUCCESS
    {
      new CDES.CSuccessAbortFailControlPacket(), // iIP_CONTROL
      null, // iIP_sense
      null  // iIP_freespace
    },
    // iS_ABORT
    {
      new CDES.CSuccessAbortFailControlPacket(), // iIP_CONTROL
      null, // iIP_sense
      null  // iIP_freespace
    },
    // iS_FAIL
    {
      new CDES.CSuccessAbortFailControlPacket(), // iIP_CONTROL
      null, // iIP_sense
      null  // iIP_freespace
    },
    // iS_inactive
    {
      new CDES.CEntryControlPacket(), // iIP_CONTROL
      new Cinactivesense(), // iIP_sense
      null  // iIP_freespace
    },
    // iS_detect
    {
      new CDES.CEntryControlPacket(), // iIP_CONTROL
      new Cdetectsense(), // iIP_sense
      new Cdetectfreespace()  // iIP_freespace
    }
  };
//<->/statestransitionsmatrix<->
```

```
//<->statescyclesdefs<->
  // None
//<->/statescyclesdefs<->

//<->statescyclesvector<->
  // Nothing (no cycles)
//<->/statescyclesvector<->


// BEGIN: Local variable space ( advisable a private modifier for them )


// END: Local variable space

//<->constructor<->
  public CDESdetect()
//<->/constructor<->
  {
    _sInstanceName=_inanaming.NewName();
    _sName=_sInstanceName;

    _ibInputBox=new CInputBox(aipcIP_PORTS);
    _obOutputBox=new COutputBox(asOP_PORTS);

//<->entrystate<->
    _iEntryState=iS_inactive;
//<->/entrystate<->
  }

  public boolean IsInvalid() { return false; }

  public String GetStateName(int iState)
  {
    if(iState==iS_DEAD) return "Dead";
    if( (iState<0) || (iState>=_asstateNames.length) ) return null;
    return _asstateNames[iState];
  }

  public COutputBox.CInputPortRef GetInputPortRef(int iInputPort)
  {
    if( (iInputPort<0) || (iInputPort>=aipcIP_PORTS.length) ) return null;
    return new COutputBox.CInputPortRef(_ibInputBox,iInputPort);
  }

  public Object Connect(int iPort, COutputBox.CInputPortRef iprInputPortRef)
               throws CPortPacketMismatchException
  { return _obOutputBox.Connect(iPort,iprInputPortRef); }

  public boolean Disconnect(int iPort,Object oInputDescriptor)
  { return _obOutputBox.Disconnect(iPort,oInputDescriptor); }

  public void DisconnectAll() { _obOutputBox.DisconnectAll(); }

  public void run()
  {
    if(IsInvalid()) return;

    _Launched();
    _SetState(iS_IDLE);
    _ppCurrentPacket=null;
    _iCurrentInputPort=-1;
```

```
    while(_iCurrentState!=iS_DEAD)
    {
//<->runningkernel<->
      _waitAndTransition();
//<->/runningkernel<->

      // NOTE: Just release the cpu (necessary with green threads)
      try { Thread.sleep(0); } // Release the cpu (Green threads)
      catch(InterruptedException ieException) { /* Nothing */ }
    }
  }

  private void _waitAndTransition()
  {
    try
    {
      _iCurrentInputPort=_ibInputBox.WaitForSomething();
      _ppCurrentPacket=_ibInputBox.GetPortPacket(_iCurrentInputPort);
      if(_aafpstateCallbacks[_iCurrentState][_iCurrentInputPort]!=null)
        _aafpstateCallbacks[_iCurrentState][_iCurrentInputPort].Function(this);
    }
    catch(InterruptedException ieException) { _SetDeadState(); }
  }

  protected boolean[] _GetStateMask(int iState) { return _aabostateMasks[iState]; }

//<->inputparam<->
  //No input param
//<->/inputparam<->
}
```

B The Compound DES avoidance

```
//<->header<->
  /*
   * File: CCoDESavoidance.java
   * Compiled by: DES Compiler v0.1 (desc)
   * Date: Tue 16 May 2000 13:35:53
   */
//<->/header<->

package DESpkg;

//<->definition<->
public class CCoDESavoidance extends CCompoundDES {
  private static CInstancesNaming _inanaming=new CInstancesNaming("CCoDESavoidance");
//<->/definition<->

//<->instancesids<->
  // Ids for DES instances
  public static final int iINSTANCES=5;
  public static final int iINS_sonarsensor1=0;
  public static final int iINS_visionsensor1=1;
  public static final int iINS_lasersensor1=2;
  public static final int iINS_detect1=3;
  public static final int iINS_avoid1=4;
//<->/instancesids<->

//<->innermappinglength<->
```

```
  // Inner mapping
  public static final int iINNER_MAPPING_REFS=5;
//<->/innermappinglength<->

//<->inputportsids<->
  // Outer mapping: input ports
  public static final int iINPUT_PORTS=3;
  public static final int iIP_sonartick=1;
  public static final int iIP_visiontick=2;
  public static final int iIP_lasertick=3;
//<->/inputportsids<->

//<->outputportsids<->
  // Outer mapping: output ports
  public static final int iOUTPUT_PORTS=2;
  public static final int iOP_velocities=1;
  public static final int iOP_freespace=2;
//<->/outputportsids<->

//<->constructor<->
  public CCoDESavoidance()
//<->/constructor<->
  {
    _sInstanceName=_inanaming.NewName();

//<->instancescreation<->
    //DES instances creation
    _adesInstances=new CDES[iINSTANCES];
    _adesInstances[iINS_sonarsensor1]=new CDESsonarsensor();
    _adesInstances[iINS_visionsensor1]=new CDESvisionsensor();
    _adesInstances[iINS_lasersensor1]=new CDESlasersensor();
    _adesInstances[iINS_detect1]=new CDESdetect();
    _adesInstances[iINS_avoid1]=new CDESavoid();
//<->/instancescreation<->

//<->innermappingcreation<->
    // Inner mapping creation
    _amrSourcePorts=new CMappingRef[iINNER_MAPPING_REFS];
    _amrDestinationPorts=new CMappingRef[iINNER_MAPPING_REFS];
    _amrSourcePorts[0]=new CMappingRef(iINS_sonarsensor1,CDESsonarsensor.iOP_sense);
    _amrDestinationPorts[0]=new CMappingRef(iINS_detect1,CDESdetect.iIP_sense);
    _amrSourcePorts[1]=new CMappingRef(iINS_visionsensor1,CDESvisionsensor.iOP_sense);
    _amrDestinationPorts[1]=new CMappingRef(iINS_detect1,CDESdetect.iIP_sense);
    _amrSourcePorts[2]=new CMappingRef(iINS_lasersensor1,CDESlasersensor.iOP_sense);
    _amrDestinationPorts[2]=new CMappingRef(iINS_detect1,CDESdetect.iIP_sense);
    _amrSourcePorts[3]=new CMappingRef(iINS_detect1,CDESdetect.iOP_freespace);
    _amrDestinationPorts[3]=new CMappingRef(iINS_detect1,CDESdetect.iIP_freespace);
    _amrSourcePorts[4]=new CMappingRef(iINS_detect1,CDESdetect.iOP_obstacles);
    _amrDestinationPorts[4]=new CMappingRef(iINS_avoid1,CDESavoid.iIP_obstacles);
//<->/innermappingcreation<->

//<->inputportscreation<->
    // Outer mapping creation: input ports
    _amrInputPorts=new CMappingRef[iINPUT_PORTS];
    _amrInputPorts[iIP_sonartick-1]=
                        new CMappingRef(iINS_sonarsensor1,CDESsonarsensor.iIP_tick);
    _amrInputPorts[iIP_visiontick-1]=
                        new CMappingRef(iINS_visionsensor1,CDESvisionsensor.iIP_tick);
    _amrInputPorts[iIP_lasertick-1]=
                        new CMappingRef(iINS_lasersensor1,CDESlasersensor.iIP_tick);
//<->/inputportscreation<->
```

```
//<->outputportscreation<->
    // Outer mapping creation: output ports
    _amrOutputPorts=new CMappingRef[iOUTPUT_PORTS];
    _amrOutputPorts[iOP_velocities-1]=new CMappingRef(iINS_avoid1,CDESavoid.iOP_velocities);
    _amrOutputPorts[iOP_freespace-1]=new CMappingRef(iINS_detect1,CDESdetect.iOP_freespace);
//<->/outputportscreation<->

//<->executiontreecreation<->
    // Execution tree creation
    CExeTree.CNode nTree0=CExeTree.CreateLeafNode(iINS_sonarsensor1);
    CExeTree.CNode nTree1=CExeTree.CreateLeafNode(iINS_visionsensor1);
    CExeTree.CNode nTree2=CExeTree.CreateOpNode(CExeTree.iCONCURRENT,nTree0,nTree1);
    CExeTree.CNode nTree3=CExeTree.CreateLeafNode(iINS_lasersensor1);
    CExeTree.CNode nTree4=CExeTree.CreateOpNode(CExeTree.iCONCURRENT,nTree2,nTree3);
    CExeTree.CNode nTree5=CExeTree.CreateLeafNode(iINS_detect1);
    CExeTree.CNode nTree6=CExeTree.CreateOpNode(CExeTree.iDISABLING,nTree4,nTree5);
    CExeTree.CNode nTree7=CExeTree.CreateLeafNode(iINS_avoid1);
    CExeTree.CNode nTree8=CExeTree.CreateOpNode(CExeTree.iDISABLING,nTree6,nTree7);
    _nExeTree=nTree8;
//<->/executiontreecreation<->
  }
}
```

C The Port Packets

C.1 The CMap class

```
//<->header<->
  /*
   * File: CMap.java
   * Compiled by: DES Compiler v0.1 (desc)
   * Date: Tue 16 May 2000 13:35:53
   */
//<->/header<->

package DESpkg;

//<->definition<->
public class CMap extends CPortPacket
//<->/definition<->
{

  // The Copy method is mandatory, you must implement it. The rest it is up to you.
  public boolean Copy(CPortPacket ppPacket)
  {
    if(ppPacket==null) return false;
//<->functioncast<->
    CMap ppCMap=(CMap) ppPacket;
//<->/functioncast<->
    // Your copy code starts here

    // Your copy code ends here
    return true;
  }

}
```

C.2 The CFreeSpace class

```
//<->header<->
  /*
   * File: CFreeSpace.java
   * Compiled by: DES Compiler v0.1 (desc)
   * Date: Tue 16 May 2000 13:35:53
   */
//<->/header<->

package DESpkg;

//<->definition<->
public class CFreeSpace extends CPortPacket
//<->/definition<->
{

  // The Copy method is mandatory, you must implement it. The rest it is up to you.
  public boolean Copy(CPortPacket ppPacket)
  {
    if(ppPacket==null) return false;
//<->functioncast<->
    CFreeSpace ppCFreeSpace=(CFreeSpace) ppPacket;
//<->/functioncast<->
    // Your copy code starts here

    // Your copy code ends here
    return true;
  }

}
```

C.3 The CObstacles class

```
//<->header<->
  /*
   * File: CObstacles.java
   * Compiled by: DES Compiler v0.1 (desc)
   * Date: Tue 16 May 2000 13:35:53
   */
//<->/header<->

package DESpkg;

//<->definition<->
public class CObstacles extends CPortPacket
//<->/definition<->
{

  // The Copy method is mandatory, you must implement it. The rest it is up to you.
  public boolean Copy(CPortPacket ppPacket)
  {
    if(ppPacket==null) return false;
//<->functioncast<->
    CObstacles ppCObstacles=(CObstacles) ppPacket;
//<->/functioncast<->
    // Your copy code starts here

    // Your copy code ends here
    return true;
  }
```

```
}
```