Automatic generation of discrete handlers of real-time continuous control tasks

Soufyane Aboubekr, Gwenaël Delaval, Roger Pissard-Gibollet, Eric Rutten, Daniel Simon
INRIA Grenoble Rhône-Alpes, France
{firstname.lastname}@inria.fr

***Abstract*—We present a novel technique for designing discrete, logical control loops, on top of continuous control tasks, ensuring logical safety properties of the tasks sequencings and mode changes. We define this new handler on top of the real-time executives built with the Orccad design environment for control systems, which is applied, e.g. to robotics and real-time networked control. It features structures of control tasks, each equipped with a local automaton, used for the reactive, event-based management of its activity and modes. The additional discrete handler manages the interactions between tasks, concerning, e.g., mutual exclusions, forbidden or imposed sequences. We use a new reactive programming language, with constructs for finite-state machines and data-flow nodes, and a mechanism of behavioural contracts, which involves discrete controller synthesis. The result is a discrete control loop, on top of the continuous control loops, all integrated in a coherent real-time architecture. Our approach is illustrated and validated experimentally with the case study of a robot arm.**

keywords: real-time control, adaptive systems, reactive programming, discrete controller synthesis

I. Motivation: RTOS and reactive control

Control systems and their programming: A control system is a heterogeneous collection of physical devices, in continuous time, and information sub-systems, with discrete time scales. The physical devices, e.g. mechanical, electrical or chemical devices, are governed by the laws of physics and mechanics. Their input/output transfer characteristics exhibit a complex dynamic behaviour (e.g. due to inertia) described by differential equations where time is a continuous variable. For their control, their state is measured or estimated using various sensors. Control theory provides a large set of methods and algorithms to govern their behaviour through closed-loop control, ensuring the respect of required performance and crucial properties like stability.

Control systems are often implemented as a set of tasks running on top of a real-time operating system (RTOS). Closed-loop digital control systems use computers to cyclically sample sensors, compute a control law and send control signals to the actuators of the physical process. The performance of a control loop, e.g. measured by the tracking error, and even more importantly its stability, strongly relies on the values of the sampling rates and sensor-to-actuator latencies [2]. A quite general rule states that smaller are the periods and latencies, better is the control performance. Thus it is essential that the implementation of the controller respects a specified timing behaviour to meet the expected performance, i.e. the actual sampling periods and latencies must be fit in ranges which are consistent with the digital controller specification. Orccad is a design environment dedicated to such control systems [5], as briefly recalled in Section II.

Discrete, reactive controllers: Another level of control systems is more related to events and states, which define execution modes of the control system, typically with changes of control law. Reactive languages based on finite state automata, like StateCharts [11], or StateFlow in Matlab/Simulink [15], are widely used for these aspects. Their underlying fundamental model, transition systems, is the basic formalism for discrete control theory, which studies closed-loop control of discrete-event and logical aspects of control systems.

Different reactive languages exist, like StateCharts mentioned before, and the languages of the synchronous approach [4]: Lustre, Esterel or Lucid Synchrone. They are used industrially in avionics and safety-critical embedded applications design [16]. They offer a coherent framework for specification languages, their compilers, with functionalities for distributed code generation, test generation and verification.

In the framework of discrete control theory, a basic technique used for the design of control loops is supervisory control, with Discrete Controller Synthesis (DCS) algorithms [14], [6]. It consists in, from a controllable system, and a behavioural property, computing a constraint on this system so that the composition of the system and this constraint satisfies the property. There also is a tool able of automated DCS [12], which is concretely connected to reactive languages and has been applied to the modelling of automatic generation of task handlers [13].

More recently the BZR language has been defined with a contract mechanism, which is a language-level integration of DCS [1], [9]: the user specifies possible behaviours of a component, as well as safety constraints, and the compiler synthesises the necessary control to enforce them. The programmer does not need to design it explicitly, neither to know about the formal technicalities of DCS, which is used in a completely encapsulated way. It is briefly explained in Section III.

Contributions of this paper: We design discrete controllers, ensuring safety properties on the interactions of underlying continuous control tasks, by applying DCS. Concretely:

1) We concretely integrate the automatically generated task handlers in the Orccad real-time executives; we make the DCS formal method usable by non-experts, as it is encapsulated in a programming language and compiler.
2) We treat the case study of a realistic application: a robot arm controller.

The compilation performance is subject to the natural complexity of the exponential algorithms, but we claim that it automatically generates an executable control solution, which is to be compared with manual programming, verification and debugging, which is even more costly. The execution cost of the controller is very small (see Section IV-D).

Outline of the paper: The next sections make brief recalls, on the programming of control systems and the Orccad approach in Section II, and in Section III on reactive programming with the BZR programming language involving DCS. Section IV describes our contribution integrating the Orccad real-time executive and the BZR programming language. Section V then illustrates the technique on the case study of a robot arm, and its different control tasks which have to be sequenced according to a reconfiguration strategy.

II. Programming control systems in Orccad

Orccad is an integrated design and programming environment dedicated to robotic systems. Robots of any type interact with their physical environment. Although this environment can be sensed by exteroceptive sensors like cameras or sonars, it is only partially known and can evolve because of robot actions or external causes. Thus a robot will face different situations during the course of a mission and must react to perceived events by changing its behaviour according to corrective actions. These abrupt changes in the system's behaviour are relevant of the theory of Discrete Events Systems. Besides the logical correctness of computations the efficiency and reliability of the system relies on many temporal constraints. The performance of control laws strongly depend on the respect of sampling rates and computing latencies. Their execution must cope with strong resource constraints.

Therefore robotic systems belong to the class of hybrid reactive and real-time systems in which different features require different programming and control methods. The Orccad environment is aimed to provide users with a set of coherent structures and tools to develop, validate and encode robotic applications.

A. Real-time tasks for continuous control

Orccad provides a bottom-up approach in which a robot controller design begins with the design and implementation of specific control laws. Most feedback control systems are essentially periodic, where the inputs (reading on sensors) and the outputs (posting on actuators) of the controller are sampled at a fixed rate. While basic digital control theory deals with systems sampled at a single rate, it has been shown, e.g. [7], that the control performance of a closed-loop digital control system can be improved using a multi-rate and multi-tasks controller : some parts of the control algorithm, e.g. updating parameters or controlling slow modes, can be executed at a slower pace. Examples are hybrid position/force control of a robot arm, visual servoing of a mobile robot following a wall or constant altitude survey of the sea floor by an underwater vehicle.

Reaching efficient control requires an adequate setting of periods, latencies and gains according to the available computing resource, e.g. as done through control/scheduling co-design approaches [3]. To this end Orccad provides a set of design, programming and code generation tools allowing the control designer to arbitrarily assign priorities and synchronisations to the set of control modules. Such a system can be analysed through algebraic techniques and can be implemented using the basic features of an off-the-shelf RTOS.

Once control laws have been designed and tuned, they are encapsulated in a so-called Robot-Task object (RT) as depicted in Figure 1. Different computation modules are defined, that take care of the drivers of the sensors and actuators, of the various numerical computations calculating the control values (which can have multiple rates, or be suspended and resumed in certain phases), of the observers which can produce diagnostic events (e.g., thresholds, or the `UnStableCam` event in the example); all the modules are assembled in a data-flow fashion, orthogonally to the logical behaviour, which is managed via discrete events, as we describe next.

B. Automata for task management

In Orccad, logical behaviour appears at two levels: locally to RTs, and at a higher level in missions.

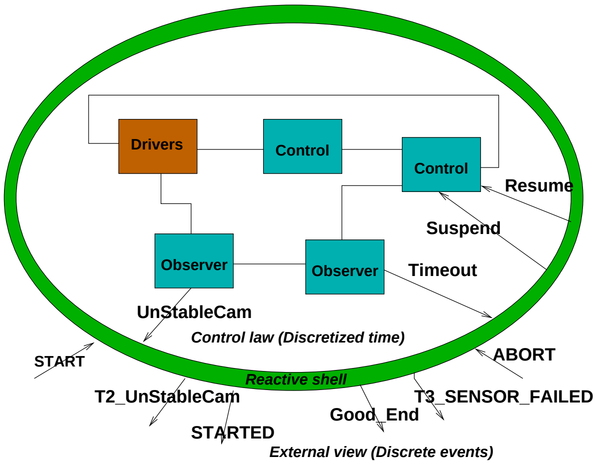


Figure 1. Encapsulation of the control law in a reactive shell

1) Generic control of RTs: It involves these events:

- preconditions, associated with e.g., measurements, sensors and watchdogs;
- events and exceptions of four types :
  - synchronisations between RTs, e.g. w.r.t. state (e.g., in Figure 1, event `STARTED`);
  - type 1 exceptions, processed locally to the RT, e.g. by tuning a parameter of the control law;
  - type 2 exceptions, ending the current RT, passing control to the upper level mission (e.g., event `T2_UnStableCam`);
  - type 3 exceptions, fatal, stopping the whole system (e.g., event `T3_SENSOR_FAILED`);
- postconditions, emitted upon RT successful termination (e.g., event `Good_End`).

2) Missions design: The RT automaton gives an abstract view which facilities their composition into more complex actions: the **Robot-Procedures** (RPs). The RP paradigm is used to logically and hierarchically compose RTs and RPs, designed to fulfil a basic goal through several possible modes, e.g, a mobile robot can follow a wall using predefined motion planning, visual servoing, or acoustic servoing according to sensory data availability. RPs design is hierarchical so that common structures and programming tools can be used from basic actions up to a full mission specification.

3) Specification and validation: The original Orccad framework uses Esterel [4] for each RT and RP logical behaviour design, verification and code generation. The global behaviour is defined by the parallel composition of the automata. The synchronous technology enables the use of formal techniques for automatic verification of the behaviour, for liveness and safety properties. For example, a safety property specifically related with control systems states that every physical actuator must be always under control, by one and only one control law. More specific properties can also be defined and validated for various case studies.

4) Execution machine for the automata: Besides the user-defined signals (pre and post-conditions, exceptions), Orccad also defines many signals used at run time to spawn and manage all the real-time threads necessary for the execution of the tasks and procedures. The current ORCCAD ESTEREL automata are compiled into a transition function in C. Input and output functions are associated to received and emitted signals, which are used to interface the synchronous reactive program with the asynchronous execution environment, i.e. the RTOS. Numerical computations can be called in linked libraries. The execution machine is in charge of feeding the automaton with signals synthesised from collected input events, running the automaton transition and exporting the output actions to the system. The automaton and execution machine are further compiled into a real-time task and event queue glued with the rest of the system, as depicted in section IV-D.

5) Position of the contribution in this paper: Until now, in ORCCAD, the discrete events control code is designed as a computer programming work, written manually, then formally verified. One drawback is the difficulty for control engineers users of specifying the discrete control without a methodology related to control theory, and the intrication of verification techniques. Another is that static manual programming of all cases fails to encompass adaptive behaviour, with regulation w.r.t. the system's state and available resources. This papers addresses these issues by considering discrete control loops on top of the continuous control loops.

III. PROGRAMMING REACTIVE SYSTEMS IN BZR

In this section we briefly introduce first the basics of the Heptagon language, to program data-flow nodes and hierarchical parallel automata [8]. As for the reactive languages introduced in Section I, the basic execution scheme is that at each reaction a step function is called, taking input flows as parameters, computing the transition to be taken, updating the state, triggering the appropriate actions, and emitting the output flows. We then define the BZR language which extends Heptagon with a new contract construct [1], [9].

A. Data-flow nodes and mode automata

Figure 2 shows a simple example of a Heptagon node, for the control of a task that can be activated by a request `r`, and according to a control flow `c`,

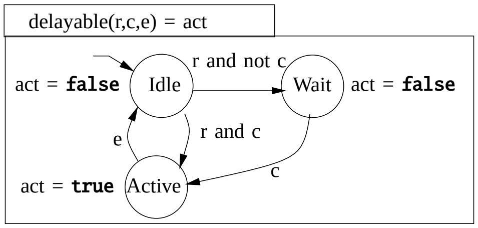


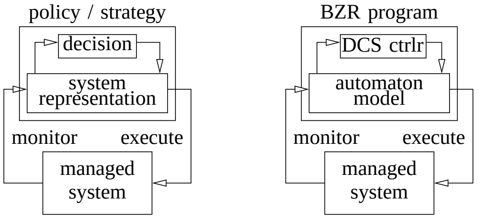
Figure 2. Example of a node in graphical syntax.

put in a waiting state; input `e` signals the end of the task. Its signature is defined first, with a name, a list of input flows (here, simple events which can be seen as Boolean flows), and outputs (here: the Boolean act), which is true when the task is active. In the body of this node we have a mode automaton : upon occurrence of inputs, each step consists of a transition according to their values; when no transition condition is satisfied, the state remains the same. In the example, `Idle` is the initial state. From there transitions can be taken towards further states, upon the condition given by the expression on inputs in the label. Here: when `r` and `c` are true then the control goes to state `Active`, until `e` becomes true, upon which it goes back to `Idle`; if `c` is false it goes towards state `Wait`, until `c` becomes true. This is a mode automaton [8] in the sense that to each state we associate equations to define the output flows. In the example, the output `act` is defined by different equation in each of the states.

We can build hierarchical and parallel automata, as will be seen in the case study e.g., in Figure 13 In the parallel automaton, the global behaviour is defined from the local ones: a global step is performed synchronously, by having each automaton making a local step, within the same logical instant. In the case of hierarchy, the sub-automata define the behaviour of the node as long as the upper-level automaton remains in its state.

B. Contracts in the BZR language

1) Motivation: With this new construct, the management of dynamical adaptivity can be considered as a control loop, on continuous or discrete criteria. It is illustrated in Figure 3(a): on the basis of monitor information and of an internal representation of the system, a control component enforces the adaptation policy or strategy, by taking decisions w.r.t. the adaptation or reconfiguration actions to be executed, forming a closed control loop. The design of control loops with known behaviour and properties is the classical object of control theory. Applications of continuous control theory to computing systems have been explored quite broadly. In contrast, qualitative or logical aspects, as addressed by discrete control theory, have been considered only recently for adaptive computing systems [17]. In our new approach, DCS is encapsulated in the compilation of BZR [1], [9]. Models of the possible behaviours of the managed system are specified in terms of mode automata, and adaptation policies are specified in terms of contracts, on invariance properties to be enforced. Compiling BZR yields a correct-by-construction controller, produced by DCS, as illustrated in Figure 3(b), in a user-friendly way: the programmer does not need to know technicalities of DCS.



(a) Adaptive system.

(b) BZR controller.

Figure 3. BZR programming of adaptation control.

2) Contract construct: As illustrated in Figure 4, we associate a *contract* to a node. It is itself a program, with its internal state, e.g., automata, observing traces, and defining states (for example an error state where e_G is false, to be kept outside an invariant subspace). It has two outputs: e_A, *assumption* on the node environment, and e_G, to be guaranteed or *enforced* by the node. A set $C = \{c_1, \ldots, c_q\}$ of local controllable variables will be used for ensuring this objective. This contract means that the node will be controlled, i.e., that values will be given to $c_1, \ldots, c_q$ such that, given any input trace yielding e_A, the output trace will yield e_G. This will be obtained automatically, at compilation, using DCS.

Without giving details [9] out of the scope of this case study, we compile such a BZR contract node into a DCS problem as in Figure 5. The body and the contract are each encoded into a state machine with transition function (resp. $Trans$ and TrC), state (resp. $State$ and

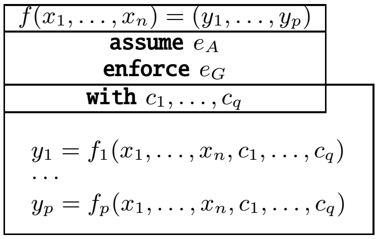

$f(x_1, \ldots, x_n) = (y_1, \ldots, y_p)$
assume e_A
enforce e_G
with $c_1, \ldots, c_q$
$y_1 = f_1(x_1, \ldots, x_n, c_1, \ldots, c_q)$
$\cdots$
$y_p = f_p(x_1, \ldots, x_n, c_1, \ldots, c_q)$

Figure 4. BZR contract node graphical syntax

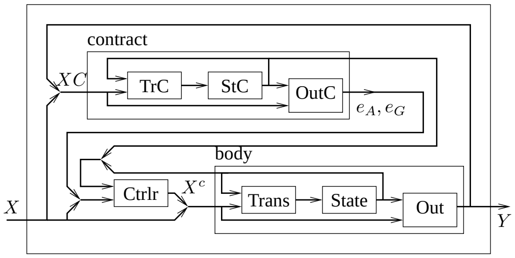


Figure 5. BZR contract node as DCS problem

StC) and output function (resp. Out and $OutC$). The contract inputs XC come from the node's input X and the body's outputs Y, and it outputs e_A, e_C. Assuming e_A produced by the contract program, DCS will obtain a controller $Ctrlr$ for the objective of enforcing e_G (i.e., making invariant the sub-set of states where $e_A \Rightarrow e_G$ is true), with controllable variables $c_1, ...c_q$. The controller then takes the states of the body and the contract, the node inputs X and the contract outputs e_A, e_G, and it computes the controllables X_c such that the resulting behaviour satisfies the objective.

IV. Discrete control handlers of continuous control tasks

A. Integration in a development process

As announced in Section I, our first contribution is the integration of BZR reactive controllers, using DCS, into the Orccad runtimes. The general scheme for using BZR consists of a treatment of the control part, using our target-independent language and compiler, in derivation of the main system development process. In its instantiation for the case of ORCCAD, illustrated in Figure 6, one can see phases of:

- extraction of control part from the adaptive system, in the form of a BZR program;
- BZR compilation: synchronous compilation to:
  - a Boolean equations form, with contracts compiled into DCS objectives; given to DCS to produce the constraint on controllables;
  - a sequential C code for the automata;

  both are then assembled into an executable involving a resolution of the synthesised constraint;
- re-linking of the latter into the global executive.

B. General architecture

1) Discrete and continuous layers: The contribution of this paper is a novel method for designing discrete, logical control handlers, on top of continuous control tasks. The goal is to ensure, by a discrete control loop, logical safety properties of the tasks sequencings and

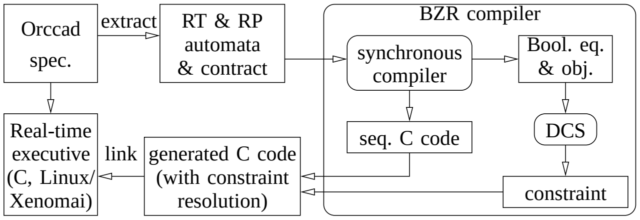

Orccad spec.
extract
RT & RP automata & contract
BZR compiler
synchronous compiler
Bool. eq. & obj.
Real-time executive (C, Linux/ Xenomai)
link
generated C code (with constraint resolution)
seq. C code
DCS
constraint

Figure 6. Development process for BZR with Orccad.

mode changes. We contribute this new layer on top of the real-time executives built with the Orccad design environment for control systems, by establishing the connection with the BZR language and compiler, which is relying upon discrete controller synthesis techniques.

This is illustrated in Figure 7 where, elaborating on the general Figure 3(b), we show how the physical system (a robot, with sensors giving values, and actuators taking commands) is in a closed loop with the continuous control layer of the computing system. The latter is implemented on a RTOS, in the form of real-time tasks in the Orccad approach .

These tasks are provided with local controllers in terms of reactive automata, that are interacting with the real-time tasks typically through events corresponding to activation of tasks, or their stopping, or exceptions to be handled. We will consider also application automata, which are describing the sequencings of tasks, in reaction to internal events like task ends, or also to external events from the controlled system. The application automaton interacts with the local automata typically through emitting starting events towards them, and receiving end or exception events. On the basis of these automata, we build another layer of closed-loop control, in the computing system, this time on discrete aspects modelled in these transition systems. We will use DCS to produce a controller that will enforce logical

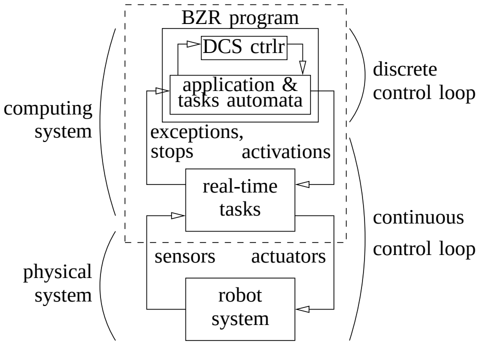


Figure 7. Discrete control handlers of continuous control tasks.

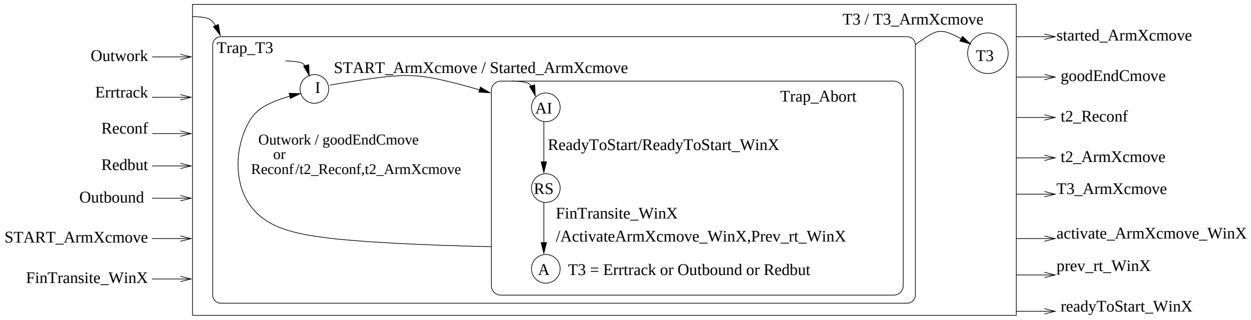


Figure 8. BZR/Heptagon programming of the generic task control automaton, in the case of ArmXcmove.

objectives on the allowed sequencings of tasks.

2) Design and development process: Figure 6 shows that the particularities are in the interface between Orccad and BZR, at the two levels of: language, to have the RT and RP automata of Orccad in BZR; and executive, where the code generated by BZR is linked into the real-time executive generated by Orccad.

C. Language-level integration

1) RT automata: Figure 8 illustrates the BZR/-Heptagon programming of the generic automaton node associated to each task, in the case of ArmXcmove. Input and output signals are exchanged with three main components of the architecture:

- the real-time tasks managed by the RTOS: typically to activate them, abort them, ...
- the controlled system, through sensors and monitors, as e.g., the Outbound input corresponding to the target being outside of the robot work area; signals with names featuring WinX interact with the robot (2D simulator, see Section V-A2);
- the application-level RP automaton, typically by the start signal, or T2 and T3 exceptions.

For the two first classes, the automaton is interfaced with the real-time platform as described in Section IV-D.

The hierarchical automaton is read as follows:

- The task is initially in the higher-level state called Trap_T3. This state is exited upon occurrence of the condition T3, which is defined inside the underlying mode as a disjunction of three input signals: Outbound , Errtrack, Redbut. This transition goes to the end state T3, with emission of T3_ArmXcmove towards the RP level.
- at the lower level, inside state Trap_T3, the sub-automaton is initially in state I. Upon input signal start_ArmXcmove from the application, it goes into state Trap_Abort, where another sub-automaton is executed, until the outgoing transition takes the control back to I; this happens upon the disjunction of two possible conditions: upon input reconf, then t2_reconf and t2_ArmXcmove are emitted for the RP, or upon input outwork, then goodEndCmove is emitted towards the RP, meaning that the task ended with success.

This automaton constitutes the BZR/Heptagon encoding of the behaviour described previously in Section II.

2) RP automaton: The RP behaviour could of course be programmed in automata as in classical ORCCAD. Using the special feature of BZR involves a change in specification style, because of the mixture between imperative behaviours and declarative control objectives.

Automaton of tasks sequencing: It describes possible behaviours, with alternatives leading to different sequencings of the tasks upon incoming events. The choice points are associated with free Boolean variables; the intention is to use the latter as controllable variables in the DCS. The automata can also involve models of parts of the environment, occupation of resources, or observers of intended or forbidden sequences of events. It interacts with RT automata typically by sending them requests to start, and reacting from their end or exception signals. This automaton is naturally application specific; Figure 13 illustrates one on the case study.

Contracts and control objectives: The properties to be considered for controlling the tasks are coded as BZR contracts. For a given set of tasks of a system to be controlled, and application automaton, the contract specifies what properties must be invariantly enforced, e.g. those mentioned in Section II-B3. The controller obtained by DCS will enforce these, by restricting the system to required behaviours, using the controllable variables for which the values are chosen in order to satisfy the properties. Figure 13 gives an example of such a RP, equipped with a contract.

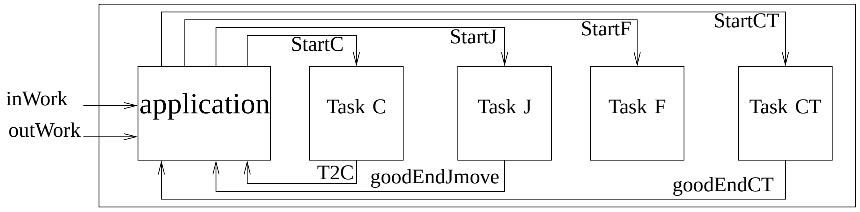


Figure 9. Complete BZR program (simplified).

3) Complete automaton: The global automaton, representing the complete control part of the system, in terms of Figure 6, is then obtained by the composition of the tasks automata, and of the application automaton. Figure 9 illustrates this for the case study.

D. Executive-level integration

At this level, we have to interface the code generated by the BZR compiler, as shown in Figure 6, with the Orccad-generated real-time executive mentioned in Section II-B4. It implements the transition step function, to be called at the appropriate pace, with appropriate input parameters, and handling of outputs. The implementation of this execution machine (i.e. of the dotted box in Figure 7) is sketched as shown in Figure 10.

A main task sets up the whole system. It spawns all the real-time tasks and associated communication and synchronisation objects. In particular it generates the needed clocks used to trigger the cyclic calculation modules. Real-time threads are made cyclic by blocking their first input port on a semaphore which is released by clock ticks. Otherwise they can be triggered by any other event, such as a data production from another thread or a signal sent by a driver.

The automaton is the highest priority task : it is awakened by the occurrence of input signals related to the execution of the controllers, e.g. pre-conditions, exceptions, and post-conditions issued by the feedback controllers. All events are serialised and received on a FIFO input events queue. In reaction, the automaton tells the RTOS what action must be taken by releasing the corresponding semaphore. Thanks to the use of a model based approach all the glue code is automatically generated, while using only basic features of operating systems make easier porting the tools for different targets (current targets are Linux/Posix threads and Xenomai).

Although this automaton is crucial for a safe and successful behaviour of the application, it spends most of time doing nothing, just waiting for input events during the cyclic execution of the control algorithms managed by the RTOS. Moreover its transitions take very short times (typically some μsecs) so that the overhead due to discrete events control is negligible.

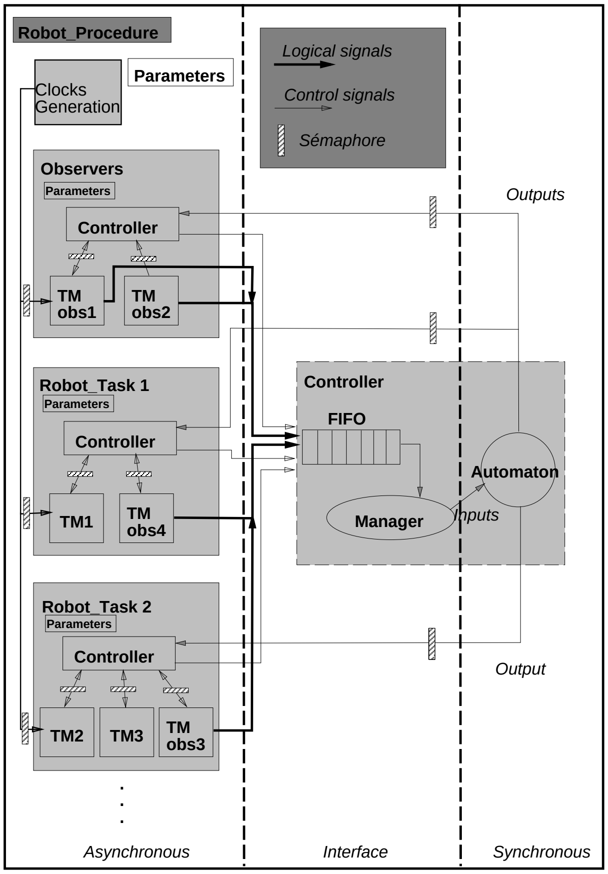


Figure 10. Implementation of the execution machine.

V. Case study of a robot arm

A. Description of the case study

1) The ArmX robot arm model: We define a robot arm, called `ArmX`, which is a two-link manipulator with rotational joints (q1,q2) shown on Figure 11. Each link i ([1,2]) has a point masses Mi ([1,2]) at the end of links. The dynamic model of the manipulator can be written in the form: $\tau = M(q)\ddot{q} + V(q,\dot{q}) + G(q)$ where $M(q)$

is the 2×2 mass matrix of the manipulator, $V(q, \dot{q})$ is an 2×1 vector of centrifugal an Coriolis terms, $G(q)$ is an 2×1 vector of gravity terms and τ the input joint torque. For this simple manipulator all details of calculation can be found in [10].

`ArmX` is equipped with a robotic tool changer which allows the robot to switch end effector. There are two tools manipulated by the arm, one is used when the target is inside the robot workspace (for example a gripper) and the second is used outside of this space (for example a proximity sensor to point the target).

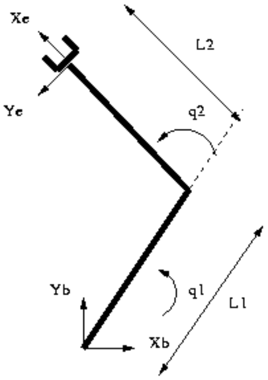


Figure 11. The ArmX model.

2) The Orccad Robot-Tasks: In this application, we identify four control-laws, embedded in four RTs:

the joint space control task: `ArmXjmove` controls the move in the joint space of the manipulator i.e., in terms of values of angles at the joints;

the Cartesian space control task: `ArmXcmove` controls the move in the Cartesian space of the manipulator, in terms of 3d coordinates; it is appropriate for aiming at targets *inside* the workspace.

the target aiming task: `ArmXfmove` controls the pointing towards a point by trajectory following; it is appropriate for aiming at targets *outside* the workspace.

the tool change task: `CT` first brings the robot to its initial position $(q1 = 0, q2 = 0)$, in order to then switch the end effector tool.

The Simulation environment: As our case study is made in simulation, we need to simulate the dynamics on the two-link manipulator `ArmX` modelled previously. We use its inverse dynamic model to compute joint accelerations: $\ddot{q} = M^{-1}(q)(\tau - V(q, \dot{q}) - G(q))$ and we obtain the current q and $\dot{q}$ by a double Euler integration.

The simulation is animated through a X11 window like in Figure 12. This window is interactive and the user can use keyboard to give information to the robot or move a target (a white square) with the mouse.

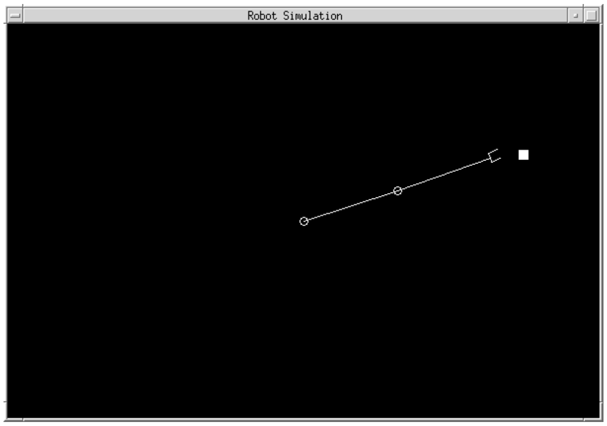


Figure 12. The ArmX 2D simulation

So, from ORCCAD or another application, this simulator is perceived like a real robot; we have functions to initialise it, to put torque, to get joint position, etc.

3) The application: The application designed is a target following task. When the target is inside the robot workspace, the effector follows the target. When it is outside of the robot workspace the manipulator point towards this target. This application must be safe and so it is performed taking into account exceptions like the tracking error is too high, joints limit are reached, or reconfiguration arm is required.

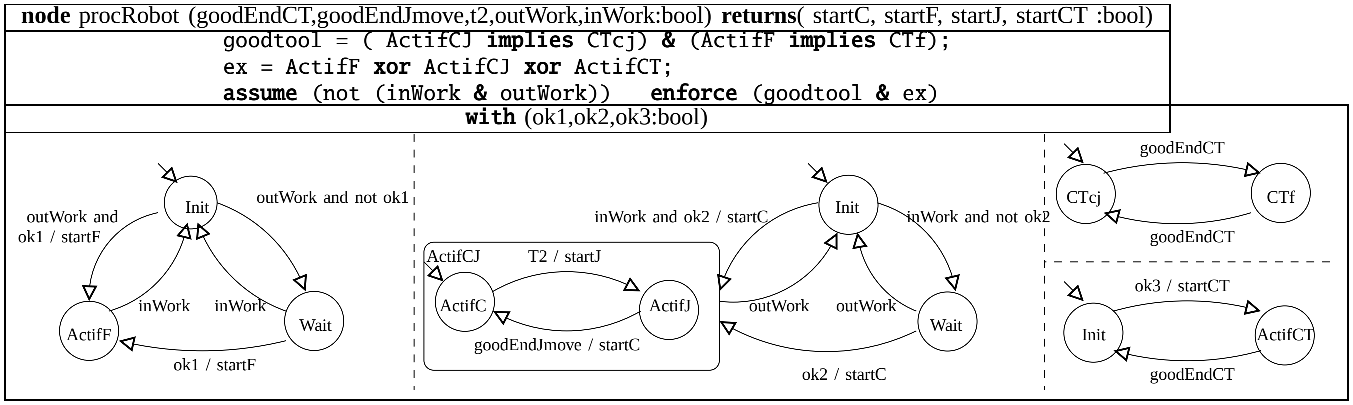
The objective is that the arm automatically changes to the appropriate tool, according to the target being inside or outside the workspace. The fact that the tool change task is inserted automatically in function of the current situation makes it an adaptive system.

B. The tasks and their local RT control

To each task corresponds an instance of the generic task control automaton; for the case of the `ArmXcmove` task the automaton is shown in Figure 8. Each of the three other tasks is associated with a similar one. All are featured in the global controller as shown in Figure 9.

C. The application RP and its global control

1) Specification as a BZR contract: We apply the BZR programming methodology: first describe possible behaviours, then specify control objectives in the contract. The application must launch robot tasks corresponding to the current state of the target (inside or outside the workspace) and change the tool arm to get the right tool for each task. So the control objective is first to ensure we have the right tool, and second,



```
node procRobot (goodEndCT,goodEndJmove,t2,outWork,inWork:bool) returns( startC, startF, startJ, startCT :bool)
                    goodtool = ( ActifCJ implies CTcj) & (ActifF implies CTf);
                    ex = ActifF xor ActifCJ xor ActifCT;
                    assume (not (inWork & outWork))   enforce (goodtool & ex)
                                        with (ok1,ok2,ok3:bool)
```

Figure 13. Global BZR node, with contract.

to check the smooth running of the application, i.e., allowing at most task to be active at a time, and also at least one, as mentioned in Section II-B3. A set $C = \{ok1, ok2, ok3\}$ of local controllable variable will be used for ensuring this objective. The contract specifies that the node will be controlled, i.e., that values will be given to ok_1, ok_2, ok_3 such that, given any uncontrollable input trace, the output trace will satisfy the two objectives.

2) The BZR node for the application: It is named `procRobot`, and illustrated in Figure 13.

PR automaton: It is composed of 4 parallel automata, described from left to right:

- the automaton for the F task: it can start the `ArmXfmove` task, by emitting `startF`, when it receives the signal `outWork` and obtains the permission of the controller by the flow `ok1`; if `ok1` is false, then it goes to state `Wait`, until `ok1` becomes true. It models the choice to delay the starting of F, and corresponds to the delayable tasks pattern illustrated in Figure 2.
- the automaton for the C and J tasks: it is hierarchical with two levels. The upper level is also an instance of the delayable task pattern; the Boolean `ok2` is used to mark the choice point.
  The sub-automaton is in the ActifCJ state manages the alternation between C and J tasks. Upon occurrence of an exception of type T2 in task C, it gives control to the task J. This is a way of handling singularities, which are points that can't be reached by using the control laws of task C: in this case control is given to task J, by sending `startJ`, to reposition the arm to reach this point. At its end a signal `goodEndJmove` is received from the RT, then task C is started again.
- the automaton observing the current tool state (top) is used to memorise the current tool of the arm. It has two states corresponding to two tools manipulated by the arm, the first one is used in the workspace accessible by the arm, and the other in outside. Every change of tool this automaton receives a `goodEndCT` signal from the RT automaton to indicate that the task ended well.
- the automaton for the CT task (bottom) is modelling the fact that it can be triggered by the controller that will be synthesised. Using controllable variable `ok3`, the controller can force the tool change by sending `startCT`.

This parallel automaton describes the possible sequencings of the tasks. It can be noted that it does not explicitly care for their exclusion, or for managing the appropriateness of the tool. This is shown next in the declarative contract, and compiled with DCS.

Contract: It can be seen in the upper part of Figure 13: it is itself a program, with its own equations. Three controllable variables, defined in the `with` part, will be used for ensuring two objectives:

- the right tool for the right task: a Boolean variable `goodtool` is defined, as the conjunction of two implications: they state that when a task is active (`ActifCJ`, respectively `ActifF`), it implies that the arm carries the right tool (`CTcj`, respectively `CTf`).
- Mutual exclusion and default control: an equation defines `ex`, which is the exclusive disjunction of active states for the tasks. it means actually two things: that there is at most one active task, and also at least one, so that the arm is always controlled, as mentioned in Section II-B3.

The contract is that, assuming that the target can not be inside and outside of the workspace at the same time, control enforces that the two Boolean are true.

D. Simulation and typical scenario

Here is a typical scenario showing the intervention of the controller on the system, so that control objectives are preserved. At some point the task `CJmove` is active, and the target inside the workspace, and the tool carried by the arm corresponds to state `CTcj`. Then, the user clicks outside of the workspace, so the application receives the `outWork` input. This causes the automaton for CJ to move by a transition to its initial state.

It also causes the automaton for task F to quit its initial state; here, we have a choice point conditioned by `ok1`. Due to the first contract property, `goodtool` must be kept true, so given that the current tool state is `CTcj` the controller can not allow the transition to `ActifF`, and must give the value `false` to `ok1`. Hence task F goes into `Wait` state. Due to the other contract property, `ex` must be kept true, which forces the controller to maintain at least one active state. Therefore it launches the task `CT` using the controllable variable `ok3`, which will change the tool. At the end of the task `CT`, the `goodEndCT` event allows the automaton observing the current tool to pass in the state `Ctf`. Thus we have the right tool for task F, and the controller can release `F` from `Wait` to `ActifF`, by giving value `true` to controllable variable `ok1`. This shows how mutual exclusion, and insertion of reconfiguration tasks can be obtained declaratively.

VI. Conclusion and perspectives

We propose a novel technique to design discrete control loops on top of continuous control tasks, ensuring logical safety properties of the tasks sequencings and mode changes. Its implementation integrates ORCCAD, a real-time control executives design environment, and the BZR reactive language, encapsulating in a user-friendly way the formal DCS technique in its compilation. A case of a robot arm is studied. It constitutes a concrete approach to implementing hybrid systems. Further work includes consolidating the integration of ORCCAD and BZR beyond this case study, enriching the models with more quantitative aspects [13], defining libraries of control models and contracts, and considering the more involving example of a Mars rover.

References

[1] S. Aboubekr, G. Delaval, and E. Rutten. A programming language for adaptation control: Case study. In *Proc. of the 2nd Workshop on Adaptive and Reconfigurable Embedded Systems, APRES'09*, 2009.

[2] K. J. Åström and B. Wittenmark. *Computer-Controlled Systems*. Information and System Sciences Series. Prentice Hall, third edition, 1997.

[3] C. Aubrun, D. Simon, and Y.-Q. Song, editors. *Co-design approaches for dependable networked control systems*. ISTE-Wiley, 2010. to appear.

[4] A. Benveniste, P. Caspi, S. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone. The synchronous languages twelve years later. *Proc. of the IEEE*, 91(1), January 2003.

[5] J.-J. Borrelly, E. Coste-Manière, B. Espiau, K. Kapellos, R. Pissard-Gibollet, D. Simon, and N. Turro. The Orccad architecture. *Int. J. of Robotics Research*, 17(4), 1998.

[6] C. Cassandras and S. Lafortune. *Introduction to Discrete Event Systems*. Kluwer Acad. Publ., 1999.

[7] A. Cervin, J. Eker, B. Bernhardsson, and K.-E. Årzén. Feedback-feedforward scheduling of control tasks. *Real-Time Systems*, 23(1–2):25–53, July 2002.

[8] J.-L. Colaço, B. Pagano, and M. Pouzet. A Conservative Extension of Synchronous Data-flow with State Machines. In *ACM Int. Conf. on Embedded Software (EMSOFT'05)*, September 2005.

[9] G. Delaval, H. Marchand, and E. Rutten. Contracts for modular discrete controller synthesis. In *Proc. of the ACM Conf. on Languages, Compilers and Tools for Embedded Systems, LCTES 2010*, 2010.

[10] J. Graig. *Introduction to Robotics, mechanics and Control*. Addison-Wesley Publishing Company, 1989.

[11] D. Harel and A. Naamad. The statemate semantics of statecharts. *ACM Trans. Softw. Eng. Methodol.*, 5(4):293–333, 1996.

[12] H. Marchand, P. Bournai, M. Le Borgne, and P. Le Guernic. Synthesis of discrete-event controllers based on the Signal environment. *Discrete Event Dynamic System: Theory and Applications*, 10(4), October 2000.

[13] H. Marchand and E. Rutten. Managing multi-mode tasks with time cost and quality levels using optimal discrete control synthesis. In *Proc. of the 14th Euromicro Conf. on Real-Time Systems, ECRTS'02*, 2002.

[14] P.J. Ramadge and W.M. Wonham. Supervisory control of a class of discrete event processes. *SIAM J. on Control and Optimization*, 25(1):206–230, January 1987.

[15] N. Scaife, C. Sofronis, P. Caspi, S. Tripakis, and F. Maraninchi. Defining and translating a "safe" subset of simulink/stateflow into Lustre. In *EMSOFT '04: 4th ACM Int. Conf. on Embedded software*, 2004.

[16] Esterel technologies. Scade: model-based development environment dedicated to safety-critical embedded software, 2010. `http://www.esterel-technologies.com/`.

[17] Y. Wang, T. Kelly, and S. Lafortune. Discrete control for safe execution of it automation workflows. In *Proc. of the 2007 EuroSys Conf.*, 2007.