Integrating Discrete Controller Synthesis in a Programming Language

Gwenaël Delaval*, Eric Rutten*, Hervé Marchand**

* LIG (UJF, INRIA) Grenoble – ** INRIA Rennes

November 17, 2011

Motivation

- mixed imperative/declarative programming language
  separation of concerns between:
  - functionality of sub-systems
  - properties of their assembly in a system

  declarative contracts are enforced upon
  imperatively described behaviors (automata)

Motivation

- mixed imperative/declarative programming language
  separation of concerns between:
  - functionality of sub-systems
  - properties of their assembly in a system

  declarative contracts are enforced upon
  imperatively described behaviors (automata)
- Discrete Controller Synthesis (DCS)
  formal technique from supervisory control theory of DES
  events, states, control modes $\leftrightarrow$ automata (e.g., StateFlow)

Motivation

- mixed imperative/declarative programming language
  separation of concerns between:
  - functionality of sub-systems
  - properties of their assembly in a system

  declarative contracts are enforced upon
  imperatively described behaviors (automata)
- Discrete Controller Synthesis (DCS)
  formal technique from supervisory control theory of DES
  events, states, control modes $\leftrightarrow$ automata (e.g., StateFlow)
- application to adaptive and reconfigurable computing systems
  closed-loop adaptation controllers : flexible execution of functionalities w.r.t. changing resource and environment

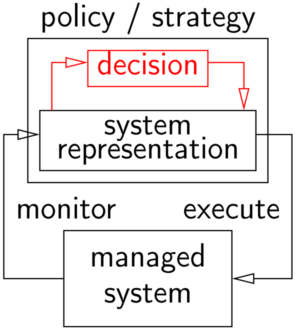
Motivation

- mixed imperative/declarative programming language
  separation of concerns between:
  - functionality of sub-systems
  - properties of their assembly in a system

  declarative contracts are enforced upon
  imperatively described behaviors (automata)
- Discrete Controller Synthesis (DCS)
  formal technique from supervisory control theory of DES
  events, states, control modes $\leftrightarrow$ automata (e.g., StateFlow)
- application to adaptive and reconfigurable computing systems
  closed-loop adaptation controllers : flexible execution of functionalities w.r.t. changing resource and environment
- Contributions
  1. BZR programming language: semantics and compilation
  2. case study : robot arm controller

Control Techniques for Adaptive Computing

- Control of computation adaptation as a closed control loop

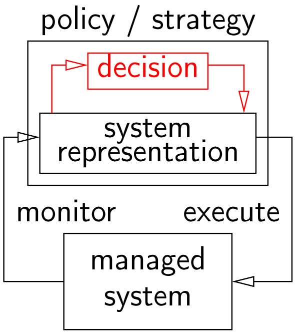


Discrete Control Techniques for Adaptive Computing

Use of Discrete Event Systems and supervisory control:

Petri nets, language theory (R&W), automata (synchronous)

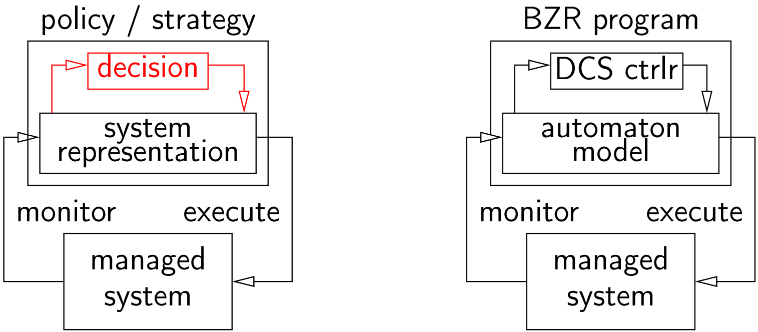
- Control of computation adaptation as a closed control loop



Discrete Control Techniques for Adaptive Computing

Use of Discrete Event Systems and supervisory control:
Petri nets, language theory (R&W), automata (synchronous)

- Control of computation adaptation as a closed control loop
- BZR programming language, and Discrete Controller Synthesis to compute the decision component (controller)



Examples of discrete computing modes

state ↔ configuration

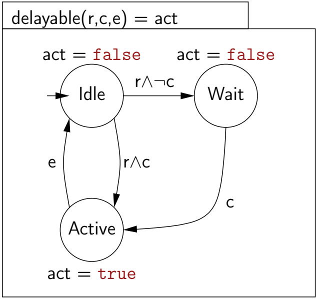
resource access, level of consumption/quality, ...

Examples of discrete computing modes

state ↔ configuration

resource access, level of consumption/quality, ...

- computation task control (example of Heptagon node)

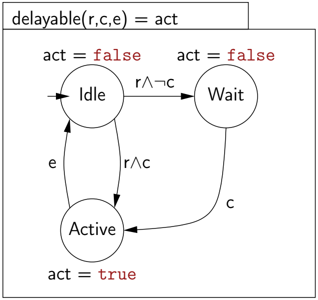


Examples of discrete computing modes

state ↔ configuration

resource access, level of consumption/quality, ...

- computation task control (example of Heptagon node)
- modes: algorithm variants for a functionality (resource, QoS)

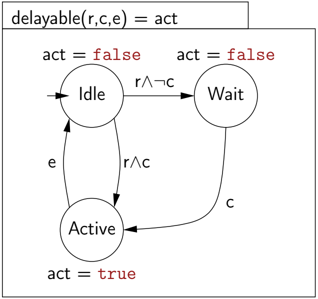


Examples of discrete computing modes

state ↔ configuration

resource access, level of consumption/quality, ...

- computation task control (example of Heptagon node)
- modes: algorithm variants for a functionality (resource, QoS)
- placement and migration : task T_i on processor/core P_j

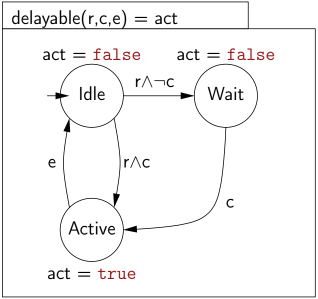


Examples of discrete computing modes

state ↔ configuration

resource access, level of consumption/quality, ...

- computation task control (example of Heptagon node)
- modes: algorithm variants for a functionality (resource, QoS)
- placement and migration : task T_i on processor/core P_j
- resource budgeting: proc./core taken for other application

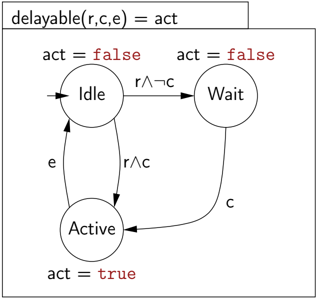


Examples of discrete computing modes

state ↔ configuration

resource access, level of consumption/quality, ...

- computation task control (example of Heptagon node)
- modes: algorithm variants for a functionality (resource, QoS)
- placement and migration : task T_i on processor/core P_j
- resource budgeting: proc./core taken for other application



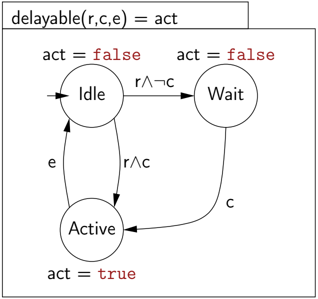
- fault tolerance : migration/rollback upon processor failure

Examples of discrete computing modes

state ↔ configuration

resource access, level of consumption/quality, ...

- computation task control (example of Heptagon node)
- modes: algorithm variants for a functionality (resource, QoS)
- placement and migration : task T_i on processor/core P_j
- resource budgeting: proc./core taken for other application



- fault tolerance : migration/rollback upon processor failure
- architecture control: frequency, DVS, stand-by in MPSoC

Discrete controller synthesis: principle

Goal

Enforcing a temporal property Φ on a system (on which Φ does not a priori hold)

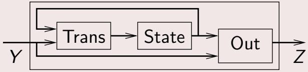
Discrete controller synthesis: principle

Goal

Enforcing a temporal property Φ on a system (on which Φ does not a priori hold)

Principle (on implicit equational representation)

State	memory
Trans	transition function
Out	output function



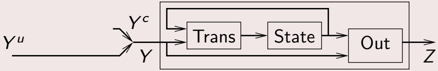
Discrete controller synthesis: principle

Goal

Enforcing a temporal property Φ on a system (on which Φ does not a priori hold)

Principle (on implicit equational representation)

State	memory
Trans	transition function
Out	output function



- Partition of inputs into controllable (Y^c) and uncontrollable (Y^u) inputs

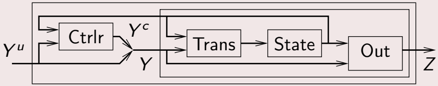
Discrete controller synthesis: principle

Goal

Enforcing a temporal property Φ on a system (on which Φ does not a priori hold)

Principle (on implicit equational representation)

State	memory
Trans	transition function
Out	output function



- Partition of inputs into controllable (Y^c) and uncontrollable (Y^u) inputs
- Computation of a controller, *maximally permissive*, such as the controlled system satisfies Φ
- tool: `sigali` (H. Marchand, INRIA Rennes)

BZR: contracts and DCS

$$f(x_1, \dots, x_n) = (y_1, \dots, y_p)$$

$$e_A \Longrightarrow e_G$$

$$\texttt{with } c_1, \dots, c_q$$

$$y_1 = f_1(x_1, \dots, x_n, c_1, \dots, c_q)$$

$$\dots$$

$$y_p = f_p(x_1, \dots, x_n, c_1, \dots, c_q)$$

- built on top of `heptagon` synchronous nodes (M. Pouzet e.a.)

BZR: contracts and DCS

$$f(x_1, \ldots, x_n) = (y_1, \ldots, y_p)$$

$$e_A \Longrightarrow e_G$$

with $c_1, \ldots, c_q$

$$y_1 = f_1(x_1, \ldots, x_n, c_1, \ldots, c_q)$$
$$\ldots$$
$$y_p = f_p(x_1, \ldots, x_n, c_1, \ldots, c_q)$$

- built on top of `heptagon` synchronous nodes (M. Pouzet e.a.)
- contract construct :
  - assuming e_A (on the environment), enforce objective e_G
  - by constraining the additional controllable variables $c_1, \ldots, c_q$ local to the component (`with`)

BZR: contracts and DCS

$f(x_1, \ldots, x_n) = (y_1, \ldots, y_p)$

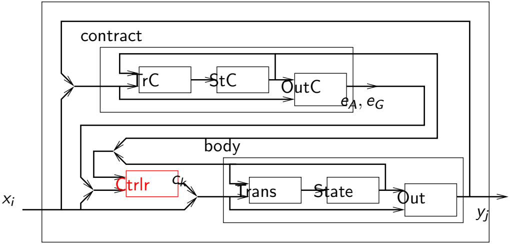
$e_A \Longrightarrow e_G$

with $c_1, \ldots, c_q$

$y_1 = f_1(x_1, \ldots, x_n, c_1, \ldots, c_q)$

$\ldots$

$y_p = f_p(x_1, \ldots, x_n, c_1, \ldots, c_q)$



- built on top of heptagon synchronous nodes (M. Pouzet e.a.)
- contract construct :
  - assuming e_A (on the environment), enforce objective e_G
  - by constraining the additional controllable variables $c_1, \ldots, c_q$ local to the component (with)
- encoded as a DCS problem (invariance) computes a local controller for each component

BZR: contracts and DCS

$f(x_1, \ldots, x_n) = (y_1, \ldots, y_p)$

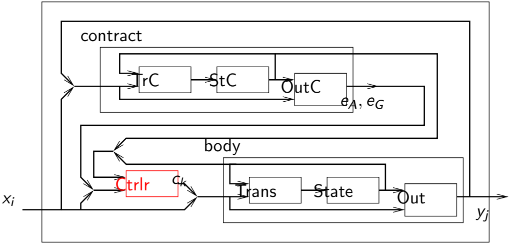
$e_A \Longrightarrow e_G$

with $c_1, \ldots, c_q$

$y_1 = f_1(x_1, \ldots, x_n, c_1, \ldots, c_q)$

$\ldots$

$y_p = f_p(x_1, \ldots, x_n, c_1, \ldots, c_q)$



- built on top of heptagon synchronous nodes (M. Pouzet e.a.)
- contract construct :
  - assuming e_A (on the environment), enforce objective e_G
  - by constraining the additional controllable variables $c_1, \ldots, c_q$ local to the component (with)
- encoded as a DCS problem (invariance)
  computes a local controller for each component

[ACM LCTES'10]

BZR modularity

BZR composite contract node:

re-use contracts of sub-nodes for the controller of the composite

assuming e_A, as well as $(e_{A1} \Rightarrow e_{G1})$ and $(e_{A2} \Rightarrow e_{G2})$

enforce e_G, as well as e_{A1} et e_{A2}

$f(x_1, \ldots, x_n) = y_1, \ldots, y_p$

assume e_A

enforce e_G

with $c_1, \ldots, c_q$

$f_1(x_{11}, \ldots, x_{1n}, c_1, \ldots, c_q) = y_{11}, \ldots, y_{1p}$

assume e_{A1}

enforce e_{G1}

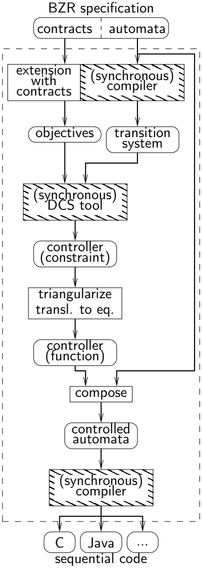
...

$f_p(x_{p1}, \ldots, x_{pn}, c_1, \ldots, c_q) = y_{p1}, \ldots, y_{pp}$

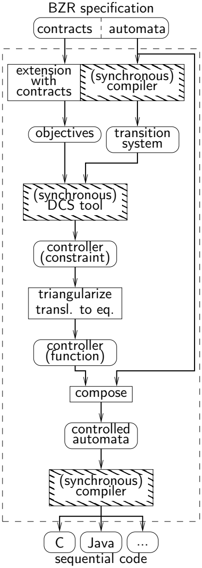
assume e_{Ap}

enforce e_{Gp}

Compilation & implementation



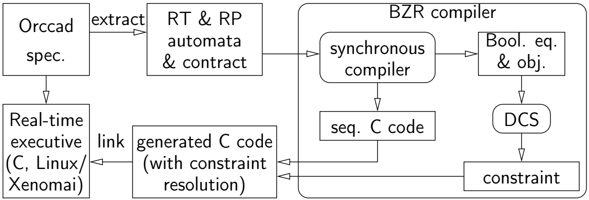
Compilation & implementation



Development process:

integration in computing system

(here: Orccad):



Programming methodology

classical programming: write the solution, then verify
here: specify the problem, then the solution is derived

- write nodes describing the possible behaviors
  in the absence of control
  identify possible choice and control points

Programming methodology

classical programming: write the solution, then verify
here: specify the problem, then the solution is derived

- write nodes describing the possible behaviors
  in the absence of control
  identify possible choice and control points
- write contracts for the control objectives
  different objectives can be possible
  controllability for a specific objective is not always given

Programming methodology

classical programming: write the solution, then verify
here: specify the problem, then the solution is derived

- write nodes describing the possible behaviors
  in the absence of control
  identify possible choice and control points
- write contracts for the control objectives
  different objectives can be possible
  controllability for a specific objective is not always given
- compile the program to obtain the controller using DCS
  can be seen as completion of partially specified program

Case study: robot arm

- robot arm (inspired from [IFAC11])
  articulations define mechanically reachable workspace
  always under control of a control law (at least one)
  exclusion between control laws (at most one)

Case study: robot arm

- robot arm (inspired from [IFAC11])
  articulations define mechanically reachable workspace
  always under control of a control law (at least one)
  exclusion between control laws (at most one)
- 6 real-time control task:
  - *CJ*: grouping 2 tasks *C*: moving *inside workspace* (*Cartesian* coord.) and *J*: moving around singularities (*Joint* coord.)
  - *F*: trajectory following, to point to a target *outside workspace*
  - *B*: same, for a target at the *border of workspace*
  - *CT*: *tool change*, with move towards tool rack, and change two arm-held tools: gripper, camera
  - *M*: *background* task, maintaining current position.

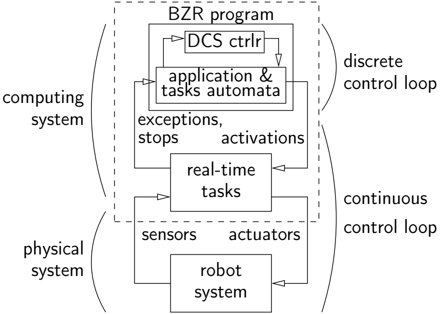
Case study: robot arm

- robot arm (inspired from [IFAC11])
  articulations define mechanically reachable workspace
  always under control of a control law (at least one)
  exclusion between control laws (at most one)
- 6 real-time control task:
  - *CJ*: grouping 2 tasks *C*: moving *inside workspace* (*Cartesian* coord.) and *J*: moving around singularities (*Joint* coord.)
  - *F*: trajectory following, to point to a target *outside workspace*
  - *B*: same, for a target at the *border of workspace*
  - *CT*: *tool change*, with move towards tool rack, and change two arm-held tools: gripper, camera
  - *M*: *background* task, maintaining current position.
- application:
  when target is *inside* workspace: grip it
  when *at border*: go to center, and point with camera
  when *outside*: point towards it, with camera

Discrete control handlers of continuous control tasks

Discrete control of tasks sequencings and mode changes

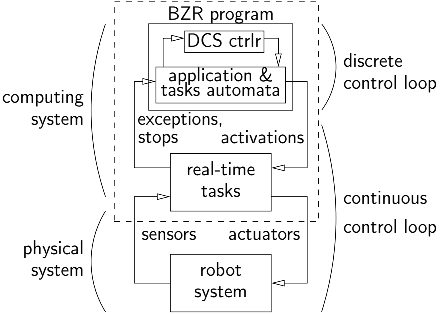
- Discrete and continuous layers



Discrete control handlers of continuous control tasks

Discrete control of tasks sequencings and mode changes

- Discrete and continuous layers

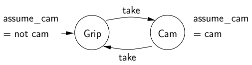


- Local task automata, coordinated by application automata with discrete supervisor, enforcing logical objective

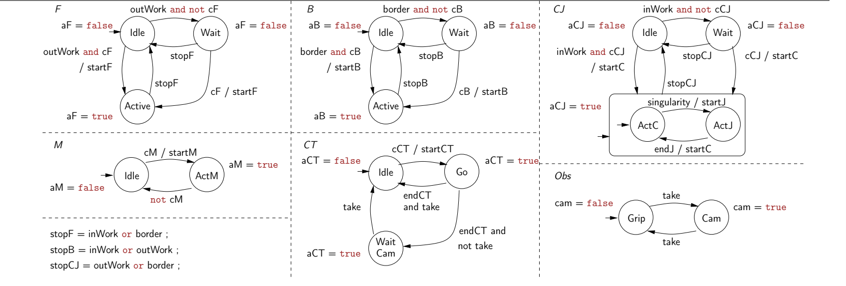
Modeling behaviors and specifying contract

behaviors of control tasks, contract featuring an observer

```
rob ( endCT, endJ, singularity, inWork, outWork, border , take ) returns startC, startF, startB, startM, startJ, startCT, cam
```




```
righttool = ( aCJ implies (not cam) )
        and ( (aF or aB) implies cam );
ex = aF xor aCJ xor aB xor aCT xor aM;
morethanone = (inWork and outWork)
           or (outWork and border)
           or (inWork and border)
assume ( assume_cam and take implies aCT and not morethanone )
enforce ( righttool and ex )
with cF, cB, cCJ, cCT, cM
```



Simulation and modularity

- typical scenario
  - `CJ` is Active, target `inWork`, tool `not cam`.

Simulation and modularity

- typical scenario
  - `CJ` is Active, target `inWork`, tool `not cam`.
  - the user clicks outside of the workspace → input `outWork` true
    transition: CJ to its initial state; F quits initial, choice cF
    contract `righttool`: $\Rightarrow cF =$ `false`: F to Wait;
    contract `ex`: $\Rightarrow cCT =$ `true`: CT to Active

Simulation and modularity

- typical scenario
  - `CJ` is Active, target `inWork`, tool `not cam`.
  - the user clicks outside of the workspace → input `outWork` true
    transition: CJ to its initial state; F quits initial, choice cF
    contract `righttool`: $\Rightarrow cF =$ `false`: F to Wait;
    contract `ex`: $\Rightarrow cCT =$ `true`: CT to Active
  - CT ends → input `EndCT`
    transition: `take` true, CT to Init; tool observer to `Cam`
    contracts `ex` and `goodtool`: $\Rightarrow cF =$ `true`: F to Active

Simulation and modularity

- typical scenario
  - CJ is Active, target inWork, tool not cam.
  - the user clicks outside of the workspace → input outWork true
    transition: CJ to its initial state; F quits initial, choice cF
    contract righttool: $\Rightarrow cF =$ false: F to Wait;
    contract ex: $\Rightarrow cCT =$ true: CT to Active
  - CT ends → input EndCT
    transition: take true, CT to Init; tool observer to Cam
    contracts ex and goodtool: $\Rightarrow cF =$ true: F to Active

- modular contracts
  two robots sharing an exclusive camera, with each its gripper

```
tworobs ( endCT1, ... border1, endCT2, ... border2 )
returns  startC1, ... startCT1, startC2, ... startCT2
enforce ( not ( cam1 and cam2 ))
with take1, take2
  startC1, ... startCT1, cam1 = rob (endCT1, ... border1, take1);
  startC2, ... startCT2, cam2 = rob (endCT2, ... border2, take2);
```

Conclusion & Perspectives

- Conclusions

  get BZR free! : `bzr.inria.fr`

  - **Discrete control integrated in programming language**
    Integration of DCS tool in compiler
  - **Application of DCS to computing system**
    here: task management
  - **Case study** robot arm, specification & simulation

Conclusion & Perspectives

- Conclusions

  get BZR free! : `bzr.inria.fr`

  - **Discrete control integrated in programming language**
    Integration of DCS tool in compiler
  - **Application of DCS to computing system**
    here: task management
  - **Case study** robot arm, specification & simulation

- Perspectives
  - **more DCS** : efficiency, expressivity
    reachability, dynamical controllers, costs on paths [WODES10]
  - **more elaborate models** of adaptive systems
    finer grain, e.g. fault tolerance [FMSD09]
  - **more integration** in existing frameworks
    e.g. component-based Fractal [EMSOFT11]
  - **more adaptive computing systems**
    reconfigurable FPGA architectures (ANR Famous)
    administration loops in data-centers (ANR CtrlGreen)