Integrating Discrete Controller Synthesis into a Reactive Programming Language Compiler

Gwenaël Delaval, Éric Rutten, Hervé Marchand

Discrete Event Dynamic Systems 2013
DOI 10.1007/s10626-013-0163-5

Integrating Discrete Controller Synthesis into a Reactive Programming Language Compiler

Gwenaël Delaval · **Eric Rutten** · **Hervé Marchand**

Received: 6 April 2012 / Accepted: 20 March 2013

Abstract We define a mixed imperative/declarative programming language: declarative contracts are enforced upon imperatively described behaviors. This paper describes the semantics of the language, making use of the notion of Discrete Controller Synthesis (DCS). We target the application domain of adaptive and reconfigurable systems: our language can serve programming closed-loop adaptation controllers, enabling flexible execution of functionalities w.r.t. changing resource and environment conditions. DCS is integrated into a1 programming language compiler, which facilitates its use by users and programmers, performing executable code generation. The tool is concretely built upon the basis of a reactive programming language compiler, where the nodes describe behaviors that can be modeled in terms of transition systems. Our compiler integrates this with a DCS tool, making it a new environment for formal methods. We define the trace semantics of our contracts language, describe its compilation and establish its correctness, and discuss implementation and examples.

Keywords Reactive systems · Synchronous programming · Discrete control synthesis · Compilation · Behavioral contracts · Adaptive / reconfigurable systems

This work was partly supported by the Minalogic project MIND and the ANR project Ctrl-Green.

Gwenaël Delaval
Univ. Grenoble, LIG, France
gwenael.delaval@inria.fr

Eric Rutten
INRIA/ LIG, Grenoble, France
eric.rutten@inria.fr

Hervé Marchand
INRIA Rennes, France
herve.marchand@inria.fr

1 Introduction

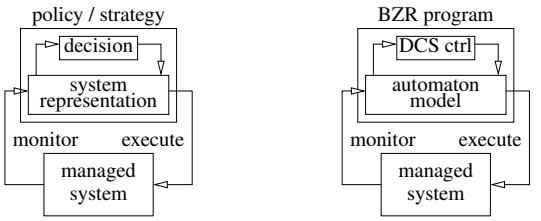
1.1 Motivation w.r.t. programming languages

We define a mixed imperative/declarative programming language, in which declarative contracts, stating dynamical temporal properties, are enforced at compilation-time upon imperatively described behaviors. We propose in this way a programming language design and compilation involving the concrete exploitation of the formal model of the *dynamical behavior* of the program, as represented by the state space of its control flow. Classically compilation takes into account statical properties holding for all states (type consistency checking, optimizations and code generation). In contrast, we want to consider properties on the dynamical control flow of the program under compilation. We consider here safety properties on sequences, for which we use synchronous observers in order to reduce them to state-based properties (we do not consider properties that evolve at run-time).

One example could be to use model-checking operations and test for the reachability of states in order to determine whether code associated to such states is dead code or not. In our approach we want to go further than this, by considering the formal technique of *Discrete Controller Synthesis (DCS)*. We integrate it in the compilation to produce (part of) the control logic implementing the program. We consider the family of reactive languages like StateCharts [19] or synchronous languages [5], which lend themselves naturally to our approach. They rely on finite state machine models, for specification at front-end, and at back-end as a target representation for code generation, model checking or DCS.

1.2 Motivation w.r.t. DCS

DCS is stemming from control theory: it ensures by construction some required dynamical and qualitative properties on a transition system, by coupling it in a closed-loop to a controller that determines the set of actions which may be taken without compromising the properties [8,37]. Application of Discrete Control Theory to computing systems is relatively recent, e.g., DCS on Petri nets can be used to automatically derive controllers avoiding dead-lock configurations in a multi-thread program [40]. We model the transition system by Symbolic Transition Systems [33], an implicit Boolean representation of the dynamic behavior (implemented by means of BDD to avoid the enumeration of the state space), and focus on the synthesis of controllers for safety properties. We integrate DCS into a compiler, and thereby improve its usability by programmers, and provides them with support for executable code generation. From a description in a high-level programming language of both the system and the expected properties to be fulfilled, the controlled system is automatically produced, in the same high-level language, from which executable code is generated.



(a) Adaptive system. (b) BZR programming.

Fig. 1 BZR programming of adaptation control.

1.3 Motivation w.r.t. adaptive and reconfigurable systems

We target the application domain of reconfigurable computing systems, which are also called adaptive systems, in the sense that they adapt, by reconfiguring themselves, to changes in their environment or execution platform concerning, e.g., power supply, communication bandwidth, quality of service, surface used in a FPGA, computation load, or dependability and fault tolerance for a safe execution. The adaptive systems that we consider can switch between known modes, and we want to control these switches (but we do not consider adaptive control). The run-time management of this *adaptivity* is the object of research on the design of adaptation strategies. A global approach is referred to as autonomic computing, where functionalities are defined for sensing the state of a system, deciding upon reconfiguration actions, and performing and executing them. These functionalities are assembled into a closed-loop as illustrated in Figure 1(a). When *safe design* is an important issue, there is a contradiction with dynamical operating system features. Obtaining *static predictability* is the goal of model-based methods for specification, validation and verification techniques of embedded systems. In a context of increasing complexity of systems, coupled with more and more integration (independent functional tasks sharing common resources), handwriting correct controllers remains difficult and error-prone. We want to combine these two different requirements for adaptive systems, i.e., to be *adaptive and predictable*. We consider the controller of such an adaptive system as a reactive system, and the design of a correct controller as a Discrete Control problem. Such a system has running configurations, represented by states, and it can perform reconfigurations, represented by labelled transitions.

We want a well-defined language that separates concerns, that is to say that supports separate specification of, on the one hand, the possible behaviors of the components, their different execution modes, the way they can switch between them, and their controllability, in the form of an automaton model; and on the other hand, in a contract, the adaptation policy to be followed, the control objectives for the components assembly, from which DCS can generate a control decision component.

1.4 Typical examples

We consider a computing system featuring a set of tasks, considered at the level of their activity behavior, with states characterized by their consuming resources: processing or memory, power, or other. We want to coordinate such tasks while enforcing the constraints around resources. These tasks are represented by their *behavior* in terms of activation state, initially idle. They can be started into an active state. Some tasks can be controlled into an intermediary state, before being activated, where they may be waiting until a required resource is free; there, they do not consume any resource. Others can be controlled, from the active state, into a suspended state, where they will consume no computing resource, but will hold their memory resource. Functionalities can also have different modes, characterized by the use of different resources, and by different levels of quality of service for the offered functionality.

Managing this multi-task system involves enforcing some *coordination properties* on the interactions between the tasks, around the resources, e.g., simple mutual exclusion between active states of two tasks, or bounded number of tasks in the active mode at the same time, in order to limit access to some bounded resource. Another, more dynamical management property concerns enforcing that between activations of two given tasks, another third task must always have been activated (e.g., re-initializing or cleaning up some accessed resources).

For example an adaptive communication system may have a behavior with modes accessing the cellular phone network, and other modes accessing the WiFi: they may involve different protocols, different prices (which constitutes another resource). An adaptation policy must define what mode to choose, in terms of properties, separately from the possible behaviors. As soon as the programmed system comprises several such concurrent tasks, with several policies to be enforced, the controller enforcing these policies can be very intricate to program manually. Therefore, automated controller generation can here be helpful.

Such simple examples illustrate the separation of concerns enabled by our language: it is the compilation, involving DCS, that computes automatically the correct relation between, on the one hand, the controllability of the components, and on the other hand, the adaptation policy. As in Figure 1(b), the reactive component, written in our reactive programming language, will be receiving input flows of task activation requests and of task termination signals; it will produce flows of task starting signals to be executed by the system platform. It decides what signals to send out, w.r.t. the automaton model of the set of tasks, while enforcing the management policy, implemented into a controller automatically generated by DCS.

1.5 Contribution and overview

Our contribution in this paper is particularly in the programming language level integration of discrete control objectives, concretized by the use of DCS within the compilation, for which we do not know, to the best of our knowledge, other closely related work. From the point of view of programming languages, it is uncommon

that a model of the dynamics of the program is taken into account by the compilation, and even rarer that it is exploited for synthesizing the resulting behavior : our approach is therefore novel. From the point of view of Discrete Control, it allows to consider novel application areas, and shows the relevance of theoretical approaches to modularity and abstraction.

A companion paper describes how compilation works with modular DCS computations [12], whereas this paper defines the programming language semantics in a denotational way. Previous work, preceding these papers, involved some separate and partial aspects of the problem, testing the idea in the framework of a more modest specialized language and elaborating on the articulation between reactive programs and DCS [33, 2].

We proceed by defining the BZR programming language and its compilation as an extension of the Heptagon reactive language presented in Section 2.1. In the semantic framework of transitions systems, the operation of DCS can be applied, as we recall in Section 2.2. On these bases, our contribution is the definition of a construct for nodes with contracts: they have an assumption part, given which they will satisfy the enforce part, relying upon local control variables. This is presented first informally in Section 3, with a simple example. The trace semantics of the language defines the behavior of the programs, as presented in Section 4.

As detailed in Section 5, the compilation of these programs involves:

- the extraction of the control part from the body and contract, and its compilation into an uncontrolled transition system;
- the extraction of the control objectives from the contracts;
- the application of DCS upon the previous two elements;
- the transformation of the obtained maximally permissive constraint into a deterministic controller function, by triangularization;
- the composition of the obtained controller with the uncontrolled program, producing the correct controlled automaton.
- the resulting composition is compiled towards target code, e.g., C or Java, and consists of a step function, to be called at each reaction of the reactive system, and a reset function for (re)initialization purposes. They then have to be embedded as a control component in the adaptive system under design [13].

In this compilation process, we re-use existing tools, for synchronous compilation and for DCS, and build our contribution on top of them.

Section 6 describes an example, illustrating how the programming language works. Section 7 gives an overview of related work, and Section 8 concludes.

2 Reactive systems and their supervisory control

This section introduces the classical bases, upon which we will build our contribution in the next sections. We first rely on the corpus of reactive languages, and more particularly synchronous languages and Mode Automata, with notations inspired from LUCID SYNCHRONE [11]. We further present the notion of discrete event systems, their supervisory control, and more particularly the automated technique of DCS.

2.1 Reactive programming and synchronous languages

2.1.1 Nodes

For scalability and abstraction purpose, we consider synchronous programs structured in *nodes*, consisting in a name, input and output variables representing flows of values, and equations defining outputs as functions of inputs. The basic behavior is that at each reaction step, values in the input flows are used in order to compute the values in the output flows for that step. Inside a node, this is expressed as a set of declarations D, which takes the form of equations defining, for each output and local, the values that the flow takes, in terms of an expression on other flows, possibly using local flows and values computed in preceding steps (also known as state values). The complete syntax of our language is given in Section 3.2. A simple equation node is illustrated in Figure 2, where for input flows a, b, c and d, all Boolean, a Boolean output flow m is **true** when more than two out of the four inputs are **true**.

```
node morethantwo(a,b,c,d:bool) = (m:bool)
  let
    m = (a and b and (c or d)) or ((a or b) and c and d)
  tel
```

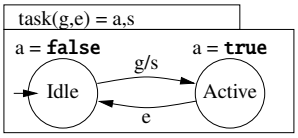
Fig. 2 Simple equation node.

A particular type of node which we consider in this paper is used to encode Mode Automata, which give the possibility of mixing equational programming with more imperative automata-based programming. We consider programs expressed as synchronous automata, with parallel and hierarchical composition. With each state of an automaton, a node can be associated, with equations, or a Mode Automaton. At each step, according to inputs and current state values, equations associated to the current state produce outputs, and conditions on transitions are evaluated in order to determine the state for the next step (i.e., transitions are considered weak). It can be noted that such higher-level constructs can be compiled towards the minimal kernel [11], hence they will not be represented explicitly in the semantics.

An example of Mode Automaton is a very basic task controller, distinguishing between its idle and active states. This example is shown in Figure 3 in graphical syntax, with an example of input/output trace. The node is named `task`. A "go" input g causes the transition from the initial idle state to the active state (step 2 on the example), where computations take place, with corresponding resources consumption. An output s is emitted on this transition[1], which will fire the concrete task starting in the controlled operating system. Another input e signals the termination of the task, and causes transition back to idle (step 5). Equations associated with the states define the value of an output a. This basic pattern will be used in different ways.

An interesting variant is the delayable task, for which Figure 4 gives the graphical and texual syntax. An additional input flow c enables the control of the request r, by

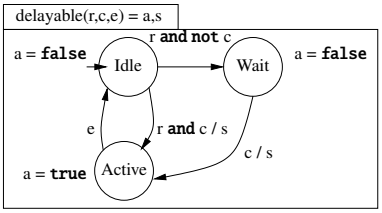
[1] Such emissions on transitions, used here for simplicity, are easily translated to equations associated with states, as in Figure 4.



Step	1	2	3	4	5	6	…
Current state	Idle	Idle	Active	Active	Active	Idle	…
g	**false**	**true**	**false**	**false**	**false**	**false**	…
e	**false**	**false**	**false**	**false**	**true**	**false**	…
a	**false**	**false**	**true**	**true**	**true**	**false**	…
s	**false**	**true**	**false**	**false**	**false**	**false**	…

Fig. 3 Basic task control.

either accepting it right away and going to the active state, or going to a wait state, from where *c* can later fire the starting. The output flow *a* appropriately defines the activity.



(a) Graphical syntax.

```
node delayable(r,c,e:bool) = (a,s:bool)
  let
    automaton
      state Idle
        do a = false ; s = r and c
        until r and c then Active
            | r and not c then Wait
      state Wait
        do a = false ; s = c
        until c then Active
      state Active
        do a = true ; s = false
        until e then Idle
    end
  tel
```

(b) Textual syntax.

Step	1	2	3	4	5	6	…
Current state	Idle	Idle	Wait	Wait	Active	Active	…
r	**false**	**true**	**false**	**false**	**false**	**false**	…
c	**false**	**false**	**false**	**true**	**false**	**false**	…
e	**false**	**false**	**false**	**false**	**false**	**false**	…
a	**false**	**false**	**false**	**false**	**true**	**true**	…
s	**false**	**false**	**false**	**true**	**false**	**false**	…

(c) Example: input/output trace

Fig. 4 Delayable task.

2.1.2 Node composition

The composition of equations constructs a system of equations, with a synchronous semantics. Nodes can be composed synchronously, e.g., automata, behaving as a synchronous product. Figure 5 shows the composite node `twotasks`, constructed by the synchronous composition of instances of the nodes `task` and `delayable` described above.

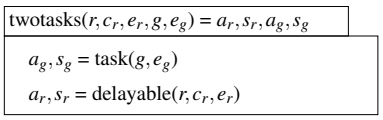


Fig. 5 Composite node with delayable task.

The corresponding composition performs a global transition at each step. The implementation takes the form of a reset function, to initialize state variables, and a step function, encoding the transition function. In such implementations, the synchrony hypothesis consists of considering that the function is guaranteed to return in bounded time.

2.1.3 Basic semantic framework for nodes

We represent the logical behavior of a Mode Automaton by a symbolic transition system (STS), as illustrated in Figure 6, in its equational form. Synchronous compilers essentially compute this transition system from source programs, particularly handling the synchronous parallel composition of nodes. For a node f, a transition function T takes the inputs X and the current state value, and produces the next state value, memorized by S for the next step. The output function O takes the same inputs as T, and produces the outputs Y.

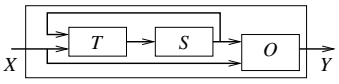


Fig. 6 Transition system for a program.

STS definition. Formally, from a node f, we can automatically derive an STS given by $\mathscr{S}_f(X,S,Y)$[2], defining a synchronous program of state variables $S \in \mathbb{B}^m$, input variables $X \in \mathbb{B}^n$, output variables $Y \in \mathbb{B}^p$. $\mathscr{S}_f(X,S,Y)$ is a four-tuple (T,O,Q,Q_0) with two functions T and O, and two relations Q and Q_0 as in (1), where the vectors S and S' respectively encode the current and next state of the system and are called

[2] Note that there exists a one to one mapping from a node f (only handling Boolean variables) to $\mathscr{S}_f$.

state variables. $T \in \mathbb{B}[S,X]$ represents the transition function.

$$\mathscr{S}_f(X,S,Y) = \begin{cases} S' = T(S,X) \\ Y = O(S,X) \\ Q(S,X) \\ Q_0(S) \end{cases} \tag{1}$$

It is a vector-valued function $[T_1,\ldots,T_n]$ from $\mathbb{B}^{m+n}$ to $\mathbb{B}^m$. Each predicate component T_i represents the evolution of the state variable S_i. $O \in \mathbb{B}[S,X]$ represents the output function. $Q_0 \in \mathbb{B}[S]$ is a relation for which the solutions define the set of initial states. The relation $Q \in \mathbb{B}[S,X]$ is the constraint between current states and events that defines which transitions are admissible, i.e., the (S,X) for which the transition function T is actually defined. This constraint can be used, e.g., to encode assumptions on the inputs, i.e., assumptions on the environment. The semantics of an STS $\mathscr{S}_f$ is defined as set of sequences $\overline{(s,x,y)} = (s_i,x_i,y_i)_i$ such that $Q_0(s_0)$ and $\forall i,\, Q(s_i,x_i) \wedge (s_{i+1} = T(s_i,x_i)) \wedge (y_i = O(s_i,x_i))$. This set of sequences is denoted by Traces$(\mathscr{S}_f)$.

Operations on STS. Given two STS $\mathscr{S}_{f_1}$ and $\mathscr{S}_{f_2}$, we note by $\mathscr{S}_{f_1} \| \mathscr{S}_{f_2}$, the synchronous parallel composition of $\mathscr{S}_{f_1}$ and $\mathscr{S}_{f_2}$ which consists in performing the conjunction of the constraint predicates of $\mathscr{S}_{f_1}$ and $\mathscr{S}_{f_2}$, and is defined whenever state and output variables are exclusive. Communications between the two systems are expressed via common inputs and outputs variables, which are considered as outputs of the composition. Formally, $\mathscr{S}_{f_1} \| \mathscr{S}_{f_2}$ is the STS $\mathscr{S}_{f_1} \| \mathscr{S}_{f_2}((X_1 \cup X_2) \setminus (Y_1 \cup Y_2), S_1 \cup S_2, Y_1 \cup Y_2)$:

$$\mathscr{S}_{f_1} \| \mathscr{S}_{f_2} = \begin{cases} S'_1, S'_2 = (T_1(S_1,X_1), T_2(S_2,X_2)) \\ Y_1, Y_2 = (O_1(S_1,X_1), O_2(S_2,X_2)) \\ Q_1(S_1,X_1) \wedge Q_2(S_2,X_2) \\ Q_{01}(S_1) \wedge Q_{02}(S_2) \end{cases}$$

Given an STS $\mathscr{S}_f(X,S,Y)$, we denote by $\mathscr{S}_f \rhd A$ the extension of constraints of $\mathscr{S}_f$ with the predicate $A \in \mathbb{B}[S,X]$, namely $\mathscr{S}_f \rhd A = (T,O,Q \wedge A,Q_o)$.

2.2 Discrete Controller Synthesis

DCS, emerged in the 80's [37,8], defines constructive methods, that ensure required properties on a system behavior. Starting from a behavioral model of the system and the set of properties that have to be satisfied, the synthesis produces the constrained system, so that only behaviors satisfying required properties are kept.

In our framework, DCS is an operation that applies on a transition system (originally uncontrolled), where inputs X are partitioned into uncontrollable (X^u) and controllable variables (X^c). It is applied with a given control objective: a property that has to be enforced by control. In this work, we consider invariance of a subset of the state space (typically, forcing a predicate over the state variables of the system to be

always true). But we can also use observer automata composed in parallel with the original system, to enable general safety properties[3].

The purpose of DCS is to obtain a controller, which is a constraint on values of controllable variables X^c, as a function of the current state and the values of uncontrollable inputs X^u, such that all remaining behaviors satisfy the property given as objective. The synthesized controller is maximally permissive, it is *a priori* a relation; it can be transformed into a control function. This is illustrated in Figure 7, where the transition system of Figure 6, as yet uncontrolled, is composed with the synthesized controller C, which is fed with uncontrollable inputs X^u and the current state value from S, in order to produce the values of controllables X^c which are enforcing the control objective. The transition system then takes $X = X^u \cup X^c$ as input and makes a step by computing the new state and producing the new outputs.

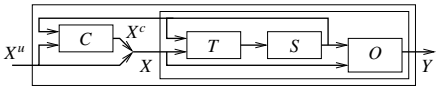


Fig. 7 Controlled transition system.

Formally, given an STS $\mathscr{S}_f$ as in (1) and a goal predicate $G(S) \in \mathbb{B}[S]$, to be *made invariant* (i.e., always maintained true by control), a controller is a predicate $K \in \mathbb{B}[S, X^c, X^u]$ that constrains the set of admissible events so that the state traces of the controller system always satisfy the predicate G. The behavior of the system supervised by the controller is then modeled by $\mathscr{S}_f \rhd K$. The controller describes how to choose the static controls; when the controlled system is in state s, and when an event x^u occurs, any value x^c such that $Q(s, x^c, x^u)$ and $K(s, x^c, x^u)$ can be chosen. One has to note that K is non-deterministic w.r.t. the controllable variables, in the sense that for each state of the system and for each valuation of the uncontrollable variables, there might exists several valuations for the controllable ones that respects K. Obviously, this non-determinism has to be solved in some ways. One possibility is to encapsulate in the system, a predicate solver, that either asks an external user to make a choice amongst the possible solutions or that itself performs a random choices amongst them. Following a method similar to the one described in [21,32], another possibility is to derive from the controller a set of functions F_i^c that depends on S, X^c, X^u and some fresh *phantom* variables ϕ_i, one for each controllable variables, namely:

$$K(S,X^c,X^u) \Leftrightarrow \exists (\phi_i)_{i\leq \ell} \begin{cases} X_1^c = F_1^c(S,X^u,\phi_1) \\ \cdots \\ X_i^c = F_i^c(S,X^u,X_1^c,\cdots,X_{i-1}^c,\phi_i) \\ \cdots \\ X_\ell^c = F_n^c(S,X^u,X_1^c,\cdots,X_{\ell-1}^c,\phi_\ell) \end{cases}$$

[3] An observer is simply an STS allowing to capture a safety property over the sequences of the systems (e.g. the event a does not occur twice in a row in the system). As usual, we assume that an observer is complete so that when performing the composition with the STS of the system, the behavior of the resulting STS is not changed.

In other words, whatever the valuation of a tuple (s, x^u, x^c) is, there exists a valuation $(v_{\phi_i})_{i\leq\ell}$ of $(\phi_i)_{i\leq\ell}$ such that $x^c_i = F^c_i(s, x^u, x^c_1, \cdots, x^c_{i-1}, v_{\phi_i})$.

At this point, either the variables (ϕ_i) can be seen as new inputs of the system or can be eliminated by choosing for each of them a value. Note that in this case, we loose the equivalence (only $\Rightarrow$ implication is kept). For clarity reasons, this is the second choice we have made in this paper. Hence, from the controller K, we derive a deterministic controller C which is a function from $\mathbb{B}^{S\cup X^u} \to \mathbb{B}^{X^c}$.

$$\mathscr{S}_f/C = \begin{cases} S' = T(S, C(S, X^u), X^u) \\ Y = O(S, C(S, X^u), X^u) \\ Q(S, C(S, X^u), X^u) \\ Q_0(S) \end{cases} \tag{2}$$

The result is a controlled STS as in (2) such that $\forall(s, x_u, y) \in \text{Traces}(\mathscr{S}_f/C),\ G(s)$. Note that the controllable variables of X^c are now encapsulated inside the STS and become internal variables. This definition is used in Section 5.3 to assess the correctness of the compilation of our language. Given an STS $\mathscr{S}_f$ with X^c as controllable variables and G the predicate to be made invariant, we denote $C = \text{DCS}(\mathscr{S}_f, X^c, G)$ the operation which consists in computing a controller C so that in $\mathscr{S}_f/C$, the predicate G is always true.

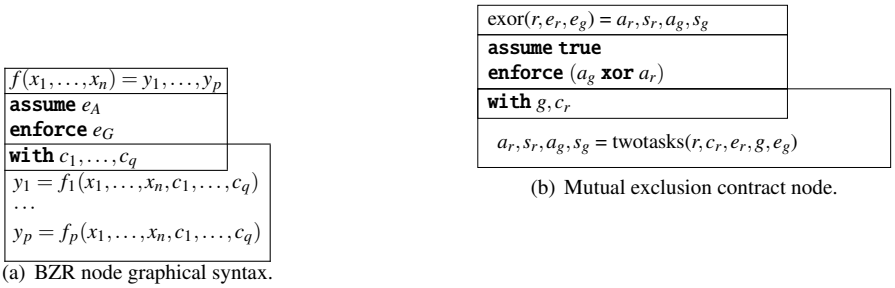
Remark 1 It might happen that given a node $\mathscr{S}_f$ and a control objective G, there is no admissible controller to ensure this goal (this is basically due to the uncontrollable aspects of input variables). In such a case, the node is said to be *uncontrollable* w.r.t. G (but might be controllable for another goal).

All the DCS procedure is actually automatic, and implemented in the tool SIGALI [33], which manipulates STS using Binary Decision Diagram (BDD), in order to avoid the state space enumeration when computing the controller. From a computational point of view, the translation of a node and its associated control objective to an STS is automatic as well as the computation of the controller C. This controller is then automatically translated in the original framework by adding a new node f_C derived from C in the original program following the scheme of Figure 7, which is essential in our approach, where we want to build a compiler using DCS.

Remark 2 In this section, we only focused on ensuring safety properties. However, it is worthwhile noticing that non-blocking properties can also be considered within this framework. It would basically consists in ensuring the reachability of a given set of states F by computing a controller C so that from every state reachable from the initial state under the control of C, F remains reachable. This procedure is also implemented in SIGALI and can be used as a possible contract within our framework (note however, that to be correct, we need to keep the maximal permissive controller; otherwise the reachability is not ensured).

3 Behavioral contracts language

We introduce a new language construct, supporting separation of concerns between description of components to be managed, and control policy to be enforced. The



(a) BZR node graphical syntax.

(b) Mutual exclusion contract node.

Fig. 8 BZR nodes.

advantage is that the programmer does not write the solution, but poses the control problem. Hence, when the policy changes for the same system, or when aspects of the system are changed but are managed with the same policy, modifications are limited, re-use is facilitated, and clarity is favored.

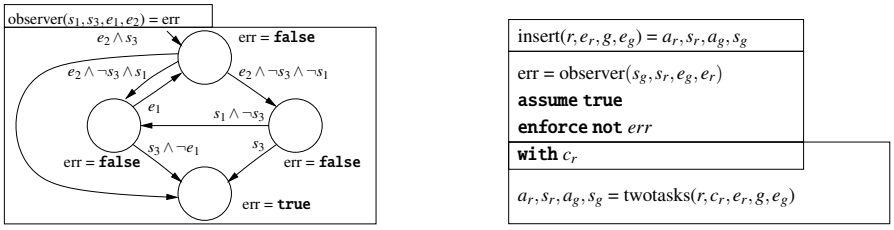
3.1 Contract construct

3.1.1 Simple contract node.

As shown in Figure 8(a), we associate to a node a *contract*, which is a program with two outputs: an output e_A representing the environment model of the node and an invariance predicate e_G that should be satisfied by the node. At the node level, the programmer declares controllable variables $c_1,\ldots,c_q$, that 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 done by computing a controller using DCS.

Figure 8(b) shows a simple problem of complementarity between activities of two tasks: one "background" and one delayable task. The contract node `exor` instantiates the node `twotasks` of Figure 5. We assume for this example and the following one that this instantiation gives access to the body of the sub-node (this option being available in the actual compiler); such assumption will not be true in further sections. A contract is given by stating that the assumption is empty (or **`true`**), and that the property to be enforced is that only one and at least one of the two tasks should be active at any time: $(a_g$ **`xor`** $a_r)$. In order to enforce this contract, g and c_r are defined locally to the contract node. Concretely, the control flows g and c_r are used to delay the starting of the delayable task when the background task is already active, until the latter stops; and conversely if the delayable task stops, the other is started.

Several nodes can have the same body, behaviorally specialized with different assumptions and enforcements. The other way around, it is possible to apply the same contract, to a different body (changing a sub-component in its refined description), and to re-obtain the updated controller simply by compilation. The contract can itself feature a program, typically automata observing traces and defining states, as men-



(a) Observer: always 1 between 2 and 3. (b) Contract node for task insertion.

Fig. 9 Observer and contract node.

tioned in Section 2.2, to express a variety of safety properties. For example, an error state can be defined where the intended property is false, with the intention to keep it outside an invariant subspace. Such an observer is illustrated in Figure 9(a): given input flows for the starting and stopping events of three tasks, it outputs value **`true`** on flow *err* when a sequence is observed such that task 3 is started (upon $s3$) after task 2 (upon its end event $e2$), without a complete execution of task 1, from s_1 to e_1, having taken place in between : this sequence violates the property that we have always 1 between 2 and 3.

The contract in Figure 9(b) uses this observer for having always an execution of the simple task between two executions of the delayable task; this amounts to make invariant the state space where *err* is **`false`**. To enforce this, c_r is used to delay the starting of the delayable task until a full execution of the other one stops.

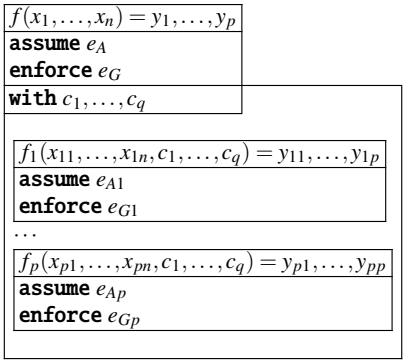


Fig. 10 BZR composite node.

3.1.2 Composite contract node.

A composite BZR node has a contract of itself, and sub-BZR-nodes with their own contracts, as in Figure 10. Sub-nodes may communicate, e.g., some of the inputs x_{pi} of sub-nodes can come from the outputs of other sub-nodes y_{1i} or from the values x_i produced by the node. This is where modularity gets involved, and the information

about contracts of the sub-nodes, which is visible at the level of the composite, will be re-used for the compilation of the composite node. The objective is still to control the body, by using the controllable variables $c_1, \ldots, c_q$, so that e_G is true, assuming that e_A is true. But here, we have information on sub-nodes: we do not keep their body as it would lead to a state space explosion, but these nodes are abstracted to their contracts, which can then be used in the DCS at that level. So, we can assume not only e_A, but also, in the case of two sub-nodes, $(e_{A1} \Rightarrow e_{G1})$ and $(e_{A2} \Rightarrow e_{G2})$. Accordingly, the control problem becomes that: assuming e_A and $(e_{A1} \Rightarrow e_{G1})$ and $(e_{A2} \Rightarrow e_{G2})$, we want to enforce e_G, and also e_{A1} and e_{A2} so that the contracts of the sub-nodes will be effectively satisfied. In particular, part of the control at the level of the composite can take care of making true the assumptions of the sub-nodes. More formal explanations are given in section 5.1.

3.2 Complete syntax of the minimal contract language

We focus on kernel of Figure 11, into which other constructs, e.g., automata, can be compiled [11]. A program P is a sequence of nodes $d_1 \ldots d_n$.

$$
\begin{array}{l|l}
P ::= d \ldots d & D ::= x = e \mid D\,\texttt{;}\,D \\
d ::= \texttt{node}\ f(x)\texttt{=}(x) & e ::= i \mid x \mid \mathrm{op}(e) \mid (e,e) \mid f(e) \\
\quad [\texttt{contract}\ (D,e,e)\ \texttt{with}\ x] & \mathrm{op} ::= \texttt{fby} \mid \texttt{fst} \mid \texttt{snd} \mid \texttt{not} \mid \texttt{or} \mid \texttt{and} \\
\quad \texttt{let}\ D\ \texttt{tel} & i ::= \texttt{true} \mid \texttt{false}
\end{array}
$$

Fig. 11 Syntax of the language with contracts.

A node is denoted:

$$
\begin{array}{l}
d = \texttt{node}\ f(x)\texttt{=}(y) \\
\qquad \texttt{contract}\ (D_1, e_A, e_G)\ \texttt{with}\ c \\
\qquad \texttt{let}\ D_2\ \texttt{tel}
\end{array}
$$

where f is the name of the node, x are its inputs, y its outputs. (D_1, e_A, e_G) **with** c is its contract, and D_2 the definitions of outputs and local variables. The contract part is optional. Within a contract, D_1 represents the exported definitions, e_A an expression for the "assume" part of the contract, e_G the "guarantee" part, and c the controllable variables. D_1 contains no sub-node application.

Definitions D_2 are a set of equations, separated by `;`, each defining a variable x by an expression e.

An expression can be Boolean constants (i), or refer to variables (x), operations op on sub-expressions, pairs of expressions, and applications of a function f on an expression.

Operations are:

- e_1 **fby** e_2 which defines a new flow with the first element of flow e_1 *followed by* the whole flow e_2: this puts a delay on a flow e_2, with an initial value given by e_1;
- **fst** and **snd** are the pair selectors (resp. first and second value);

– **not**, **or** and **and** are Boolean operators, applied point-to-point.

Binary operators (like **fby**, **or** and **and**) are considered as unary operators applied on pairs. We denote by $e_1 ope_2$ the expression $op(e_1, e_2)$.

4 Trace semantics

We give a trace semantics of our language, inspired from the denotational semantics of the LUCID SYNCHRONE language [16]. It is defined by a function denoted $[\![e]\!]$, which associates to an expression e the set of infinite traces corresponding to e's evaluation. We define some basic functions in Figure 12, upon the notion of infinite sequence of values $\mathscr{V}^\infty$. For Booleans, $True(s) \Leftrightarrow s = \texttt{true}.\texttt{true}\ldots$.

$\mathscr{N}$ from $\mathscr{V}^\infty$ to sets of triples of $\mathscr{V}^\infty$ is the set of functions defining nodes. The set of resulting values is a set of possible triples (s, s_A, s_G), where s is the result of the node, and s_A and s_G the value of respectively the "assume" and "guarantee" parts of the node's contract.

N defines node environments by, for a variable, its corresponding node function.

ρ defines trace environments by, for a variable, the set of its infinite traces of instantaneous values. $\rho_1 \oplus \rho_2$ denotes union of environments, only on distinct domains.

$[\![\cdot]\!]^N_\rho$ is the function giving the trace semantics of the language. From a node environment N and a trace environment ρ, this function gives:
  – from an expression, the set of infinite traces of its resulting values;
  – from an equation (or set of equations), the trace environment for the variable(s) it defines.

$[\![d]\!]^N$ is the function which, from a node environment N, associates to a node d the function from traces to set of traces representing this node.

Based upon this, Figure 13 gives the semantic rules as follows.

The rule (OpSeq) states that Boolean operators are applied point to point on infinite traces.

The semantics of the **fby** operator (rule (Fby)) is that the front element of the first trace is appended with the second trace. The resulting trace has the values of the second with a one-step delay; the first value of the first trace gives the initial value for the resulting delayed flow. This is the only kernel operator involving memory, from one step to the other.

Boolean constant flows i are flows of the Boolean constant (rule (Imm)). A variable x is evaluated in the trace environment ρ by extracting its value (rule (Var)). The rule (Op) describes the semantics of operators, which are applied on traces of their operands. The rule (Pair) matches pairs of traces to pairs of expressions.

$$
\begin{array}{l|l}
\mathscr{N} : \mathscr{V}^\infty \to \mathscr{P}(\mathscr{V}^\infty \times \mathscr{V}^\infty \times \mathscr{V}^\infty) & [\![\cdot]\!] : Exp \times NodeEnv \times TraceEnv \to \mathscr{P}(\mathscr{V}^\infty) \\
N : NodeEnv = Var \to \mathscr{N} & [\![\cdot]\!] : Eq \times NodeEnv \times TraceEnv \to TraceEnv \\
\rho : TraceEnv = Var \to \mathscr{P}(\mathscr{V}^\infty) & [\![\cdot]\!] : Node \times NodeEnv \to \mathscr{N} \\
\oplus : TraceEnv \times TraceEnv \to TraceEnv & [\![\cdot]\!] : Nodes \to NodeEnv
\end{array}
$$

Fig. 12 Functions for the trace semantics.

$$\mathrm{op}^\infty(v.s) = \mathrm{op}(v).\mathrm{op}^\infty(s) \quad \text{for op} \in \{\ \texttt{and}\ ,\ \texttt{or}\ ,\texttt{not}\ \} \tag{OpSeq}$$

$$\texttt{fby}^\infty(v_1.s_1, v_2.s_2) = v_1.v_2.s_2 \tag{Fby}$$

$$[\![i]\!]^N_\rho = \{i.i.\ldots\} \tag{Imm}$$

$$[\![x]\!]^N_\rho = \rho(x) \tag{Var}$$

$$[\![\mathrm{op}(e)]\!]^N_\rho = \{\mathrm{op}^\infty(s) | s \in [\![e]\!]^N_\rho\} \tag{Op}$$

$$[\![(e_1,e_2)]\!]^N_\rho = [\![e_1]\!]^N_\rho \times [\![e_2]\!]^N_\rho \tag{Pair}$$

$$[\![f(e)]\!]^N_\rho = \left\{ \begin{array}{l} s \text{ s.t. } (s,s_A,s_G) \in N(f)([\![e]\!]^N_\rho) \\ \quad \wedge \forall s_A,s_G, s_A = s_G = [\![\texttt{true}]\!]^N_\rho \end{array} \right\} \tag{App}$$

$$[\![x = e]\!]^N_\rho = \{x \mapsto [\![e]\!]^N_\rho\} \tag{Eq}$$

$$[\![D_1\,\texttt{;}\,D_2]\!]^N_\rho = [\![D_1]\!]^N_\rho \oplus [\![D_2]\!]^N_\rho \tag{Par}$$

$$\left[\!\!\left[\begin{array}{l} \texttt{node}\ f(x)\texttt{=}(y) \\ \quad \texttt{let}\ D\ \texttt{tel} \end{array} \right]\!\!\right]^N = \lambda s. \left\{ \begin{array}{l} (s_y, [\![\texttt{true}]\!]^N_\rho, [\![\texttt{true}]\!]^N_\rho) \text{ s.t.} \\ \quad \rho = fix(\lambda\rho.([\![D]\!]^N_\rho)(\{x \mapsto s\}) \\ \quad s_y \in \rho(y) \end{array} \right\} \tag{Node}$$

$$\left[\!\!\left[\begin{array}{l} \texttt{node}\ f(x)\texttt{=}(y) \\ \quad \texttt{contract}\ (D_1, e_A, e_G) \\ \quad \texttt{with}\ c \\ \quad \texttt{let}\ D_2\ \texttt{tel} \end{array} \right]\!\!\right]^N = \lambda s. \left\{ \begin{array}{l} (s_y, s_A, s_G) \text{ s.t. } \exists s_c, \\ \quad \rho = fix(\lambda\rho.([\![D_1\,\texttt{;}\,D_2]\!]^N_\rho) \\ \qquad\qquad (\{x \mapsto s, c \mapsto s_c\}) \\ \quad s_y \in \rho(y) \\ \quad s_A = [\![e_A]\!]^N_\rho \\ \quad s_G = [\![e_G]\!]^N_\rho \\ \quad True(s_A) \Rightarrow True(s_G) \end{array} \right\} \tag{NodeC}$$

$$[\![d_1 \ldots d_n]\!]^N = [\![d_2 \ldots d_n]\!]^{N \oplus \{d_1 \mapsto [\![d_1]\!]^N\}} \tag{Nodes}$$

$$[\![P]\!] = [\![P]\!]^\emptyset \tag{Prog}$$

Fig. 13 Trace semantics of the BZR language.

The rule (App) states that a function can be applied as the special case of a node where "assume" and "guarantee" parts are constantly true. This application has no valid semantics if either part of the contract is not constantly true, in particular the "assume" part (s_A).

The semantics of equations (rule (Eq)) is that infinite traces of the left-hand side of the equation are given by the semantics of the right-hand side of the equation.

The rule (Par) gives the semantics of parallel definitions, which is given by the union of the environments obtained from the composed definitions.

The rule (Node) defines the semantics of nodes without contracts. A node f defines a function which, given an input trace value s, gives a set of trace triplets (for consistency with nodes with contracts) which first value is the output value of f. This value is defined through a trace environment ρ, defined as a fix-point applying equations of D, initialized with the trace value of the input. This fix-point allows the incremental computation of values for the synchronous composition of parallel equations.

The rule (NodeC) gathers the specificity of our contribution: it gives us the semantics of the application of a node with a body D_2, with a contract having a body

D_1 and controllable variables c. It is defined iff for each input s, there exists a trace s_c associated to the controllable variable c (within the fix-point initialization), such that the implication between the evaluations of the "assume" (e_A) and "guarantee" (e_G) parts holds.

The rule (Nodes) builds the node environment from definitions in the sequence of nodes. Finally, the rule (Prog) builds the environment from a program P and an initial empty environment.

5 Compilation

We show in this section how our language is compiled towards STS, on which DCS can be applied.

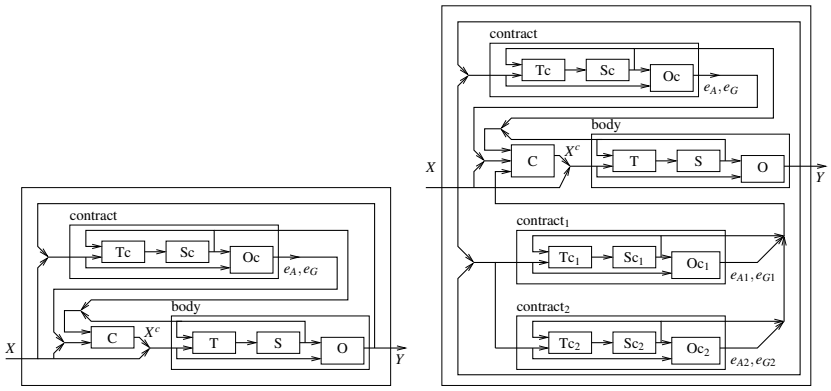
5.1 Principle and corresponding DCS problem

The purpose of the compilation principle presented here is to show how to use a DCS tool, within the compilation process of our language. We want to obtain, from each node, an STS as defined in Section 2.2, in order to apply DCS on it. The obtained controller is itself a node of equations, recomposed in the target language. Given the definition of the semantics of a node in Section 4, and given the definition of DCS in Section 2.2, the result obtained when recomposing the synthesized controller in a node as in our compilation, behaves like the semantics of a contract node given in Section 4.

5.1.1 Single contract enforcement

To compile a single contract node, we encode it as a DCS problem where, assuming e_A (produced by the contract program, which will be part of the transition system), we will obtain a controller for the objective of enforcing e_G (i.e., *making invariant* the subset of states where $e_A \Rightarrow e_G$ is true), with controllable variables X^c. This is illustrated in Figure 14(a), re-using instances of the transition system of Figure 6: one for the contract and one for the body of the node, and showing the controller as in Figure 7. The contract program has access to the inputs X and outputs Y of the body; its outputs e_A and e_G, and its state, which is part of the global state, are accessible to the controller, as well as the state of the body and its (uncontrollable) inputs X.

More formally, given a node f with its associated STS $\mathscr{S}_f(X^c \cup X^{uc}, S, Y)$, a contract will be given by a tuple $Cont = (\mathscr{S}_c, A, G)$ where $\mathscr{S}_c((X \cup Y), S_c, \emptyset)$ is an STS encoding the body of the contract, $A \in \mathbb{B}[S_c]$ and $G \in \mathbb{B}[S_c]$ are predicates, respectively encoding the signals e_A and e_G. Now, in order to enforce the contract we consider the STS $\mathscr{S}_{Cont} = (\mathscr{S} \| \mathscr{S}^c) \rhd A$ on which we enforce by control the invariance of G. The result is a controller $C = DCS(\mathscr{S}_{Cont}, X^c, G)$



(a) BZR node as DCS problem. (b) BZR composite as DCS problem.

Fig. 14 BZR and DCS problem

5.1.2 Compiling a composite contract node.

When compiling a composite contract node f, with sub-nodes, e.g. f_1 and f_2 as described in Figure 10, one can associate to each sub-node its corresponding STS $\mathscr{S}_{f_i}(X_i, S_i, Y_i)$. This is illustrated in Figure 14(b), re-using the same graphical notations. The STS $\mathscr{S}_f$ can then be represented by the STS

$$\mathscr{S}_f(X,S,Y) = (\mathscr{S}' \| \mathscr{S}_{f_1} \| \mathscr{S}_{f_2})$$

where $\mathscr{S}'$ corresponds to the STS derived from additional local code used to described f. Note that $X_i^{uc} \subseteq S \cup X^{uc} \cup X^c \cup Y$, namely the uncontrollable variables of the lower level can be defined either by state, uncontrollable or controllable inputs, or outputs variables of the upper system. Thus to proceed to the encapsulation we need to rename the variables Y_i^{uc} according to their new name in the new system.

We assume that each sub-node comes with a contract $Cont_i = (\mathscr{S}_{c_i}, A_i, G_i)$, with $\mathscr{S}_{c_i}(X_i \cup Y_i, S_{c_i}, \emptyset)$, $A_i \in \mathbb{B}[S_{c_i}]$, $G_i \in \mathbb{B}[S_{c_i}]$, and that a controller C_i to ensure the invariance of G_i.

We want now to obtain a controller C for the system $\mathscr{S}_f$ to fulfill a contract $Cont = (\mathscr{S}_c, A, G)$, with $S_c(Y \cup Z, S_c, \emptyset)$, $A \in \mathbb{B}[S_c]$ and $G \in \mathbb{B}[S_c]$. One way to do this is to compute the whole dynamic of S and to control it using the previous method, but this would lead to a state space explosion. Instead, we will use the contracts of the sub-components as an abstraction of them. Thus, we use an abstracted STS $\overline{\mathscr{S}_f}$, defined as the composition of $\mathscr{S}'$ with the system part of the subcontracts, constrained with the properties enforced by C_i on each of the sub-components. In other words, we take the assume and enforced parts of the subcontracts as environment model of the abstracted system.

Remark 3 The Y_i variables were outputs of the lower level. As we abstract away the body of this system, these variables have now to be considered as uncontrollable variables of the upper system (indeed, there is no way to know their value). Besides,

the value of these variables is normally computed according to the value of X_i^{uc} and internal variables. Hence, it exists causality problems between these variables and the variables of the upper level. See [12] for more details.

We define the new system to be controlled as follows:

$$\overline{\mathscr{S}}_f(Y^{uc} \cup Z_1 \cup \ldots \cup Z_n \cup X^c, S, Y) = \Big(S' \| \big(S_{c_1} \rhd (A_1 \Rightarrow G_1)\big) \| \big(S_{c_2} \rhd (A_2 \Rightarrow G_2)\big)\Big)$$

We should notice that, in order to the STSs $\mathscr{S}_i$, controlled by their controller, to be evaluated in a correct environment, the predicates A_i must be satisfied. Therefore, we define a new contract $Cont'$, which will be used to compute a controller on $\overline{\mathscr{S}}_f$:

$$Cont' = (\mathscr{S}_c, \hat{A}, \hat{G}) \text{ where } \begin{cases} \hat{A} = A \\ \hat{G} = G \wedge A_1 \wedge A_2 \end{cases}$$

We then compute controller C, enforcing contract $Cont'$ on STS $\overline{\mathscr{S}}_f$. We can further show that whenever $\mathscr{S}_{f_i}/C_i$ satisfies the invariance of G_i for $i = 1,2$ then

$$(\mathscr{S}' \| \mathscr{S}_{f_1}/C_1 \| \mathscr{S}_{f_2}/C_2)/C$$

satisfies the invariance of G.

Remark 4 As mentioned in Section 2.2, one can also consider non-blocking contract within our framework. However, even-though the controlled sub-nodes are non-blocking it might happen that the composition of these nodes gives access to a blocking node. In order to ensure the non-blocking aspect, we have to consider the whole system (with no abstraction) and ensure this property on this system (thus loosing the modular aspect of the controller synthesis).

5.2 Formal compilation rules

We describe the compilation towards STS through a function Tr, from BZR equations and expressions towards tuples $(\mathscr{S}, X^u, G)$ where

- $\mathscr{S}(X,S,Y) = (T,O,Q,Q_0)$ denotes the obtained STS: for expressions, it only defines one output value. For equations, the outputs are the variables they define.
- X^u denotes the additional uncontrollable inputs of the obtained STS, corresponding to the outputs of the applied sub-nodes.
- G corresponds to the synthesis objectives from contracts of sub-nodes.

This compilation function Tr, applied on nodes, produces nodes without contracts. We consider the compilation on normalized programs, following the restricted syntax given below, defined such that the expressions e correspond to those allowed in STS.

$$\begin{array}{lll} D & ::= & x = e \mid D\,\texttt{;}\,D \mid x = f(x) \mid x = v\ \texttt{fby}\ x \\ e & ::= & i \mid x \mid \mathrm{op}(e) \mid (e,e) \\ \mathrm{op} & ::= & \texttt{fst} \mid \texttt{snd} \mid \texttt{not} \mid \texttt{or} \mid \texttt{and} \\ i & ::= & \texttt{true} \mid \texttt{false} \end{array}$$

$$\mathrm{Tr}(e) = (\mathscr{S}, \emptyset, \emptyset) \text{ where } \mathscr{S}(\emptyset, \emptyset, \{y\}) = \{\, y = e \tag{C-Exp}$$

$$\mathrm{Tr}(y = v \ \mathbf{fby}\ x) = (\mathscr{S}, \emptyset, \emptyset) \text{ where } \mathscr{S}(\{x\}, \{s\}, \{y\}) = \begin{cases} s' = x \\ y = s \\ s_0 = v \end{cases} \tag{C-Fby}$$

$$\mathrm{Tr}(y = f(x)) = (\mathscr{S} \| \mathscr{S}_f^c, \{y\}, A_f) \text{ where } \mathscr{S}(\{y\}, \emptyset, \{z\}) = \begin{cases} z = y \\ A_f \Rightarrow G_f \end{cases} \tag{C-App}$$

$$\frac{\mathrm{Tr}(D_1) = (\mathscr{S}_1, Y_1, G_1) \qquad \mathrm{Tr}(D_2) = (\mathscr{S}_2, Y_2, G_2)}{\mathrm{Tr}(D_1 \,;\, D_2) = (\mathscr{S}_1 \| \mathscr{S}_2, Y_1 \cup Y_2, G_1 \wedge G_2)} \tag{C-Par}$$

$$\mathrm{Tr}\left(\begin{array}{l} \mathbf{node}\ f(X){=}(Y) \\ \ \mathbf{contract} \\ \ \ (D_1, A_f, G_f) \\ \ \ \mathbf{with}\ (c_1, \ldots, c_n) \\ \ \mathbf{let}\ D_2\ \mathbf{tel} \end{array}\right) = \left(\begin{array}{l} \mathbf{node}\ f(X){=}(Y) \\ \mathbf{let} \\ \ (c_1, \ldots, c_n) \\ \ \ = C(S, X \cup X^u)\,; \\ \ D_1 \,;\, D_2 \\ \mathbf{tel} \end{array}\right)$$

$$\text{where} \begin{cases} \mathrm{Tr}(D_1) = (\mathscr{S}_f^c, \emptyset, \emptyset) \\ \mathrm{Tr}(D_2) = (\mathscr{S}_2, X^u, G_2) \\ \mathscr{S} = (\mathscr{S}_f^c \| \mathscr{S}_2)(X \cup X^u, S, Y \setminus X^u) \\ C = \mathrm{DCS}(\mathscr{S} \triangleright A_f, \{c_1, \ldots, c_n\}, G_2 \wedge G_f) \end{cases} \tag{C-Node}$$

$$\mathrm{Tr}(d_1 \ldots d_n) = \mathrm{Tr}(d_1)\,\mathrm{Tr}(d_2 \ldots d_n) \tag{C-Prog}$$

Fig. 15 Compilation rules.

Particularly, equations with subnodes applications in the expression are decomposed into equations defining intermediate variables, with either an expression or a subnode application. Compilation rules are given in Figure 15.

C-Exp : expressions are directly translated to an STS (T, O, Q, Q_0) where only $Q \neq \emptyset$, in the form of an output function for y.

C-Fby introduces a fresh state variable s, with appropriate transition and initialization.

C-App translates applications by composing: $\mathscr{S}_f^c$, the STS of the contract of f; $\mathscr{S}$, where the output y of the applied sub-node is considered as an additional uncontrollable variable: as the body of f is abstracted, the value of y cannot be known. This composition represents the abstraction of the application. The assume/guarantee part of the contract of the applied sub-node, as in **C-Node**, $(A_f \Rightarrow G_f)$ is added as a constraint Q of the STS. It can be noted that the point of this is to favor DCS, by giving some information of behaviors of sub-nodes: this can enable to find control solutions, which a black box abstraction would not allow. Hence it is an optimization of the modular control generation, not a necessity w.r.t. the language semantics, which it should of course not jeopardize (see Section 5.3).

C-Par : STSs from parallel equations are composed; additional variables from subnodes are gathered; the synthesis objective is the conjunction of sub-objectives.

C-Node translates nodes with contracts to *controlled* nodes. It features the application of the DCS function of Section 2.2 to the composition of the STSs from the contract and the body. This composition is constrained with the operation

$\triangleright$ by the assumption part A_f of the contract. The additional variables induced by the abstractions of the applications are added as *uncontrollable inputs* to the STS on which the DCS is performed. This rule defines $\mathscr{S}_f^c$, A_f and G_f used for applications of f (rule C-App). The translation of nodes without contract is the identity, defining $\mathscr{S}_f^c$ as empty STS (neutral for parallel composition), and $A_f = G_f = \texttt{true}$.

C-Prog translates the sequence of nodes of the program.

5.3 Conformance to trace semantics

The above compilation rules show how to obtain, from nodes with contracts, nodes where the contracts have been replaced by a controller function obtained by DCS. The semantics of nodes with contracts defines the set of possible output traces from input traces, given the different possible controllable values. Using a computed controller function gives us, for one input trace, only one controllable variable trace and thus only one output trace. We expect that this specific output trace belongs to the set of traces defining the semantics of the node with its contract. Therefore, we define a relation $\sqsubseteq$ on program and node semantics, based on the set inclusions of expressions semantics:

$$\rho_1 \sqsubseteq \rho_2 \Leftrightarrow \mathrm{dom}(\rho_1) = \mathrm{dom}(\rho_2) \wedge \forall x \in \mathrm{dom}(\rho_1), \rho_1(x) \subseteq \rho_2(x) \tag{3}$$

$$\begin{aligned} N_1 \sqsubseteq N_2 \Leftrightarrow\ & \mathrm{dom}(N_1) = \mathrm{dom}(N_2) \wedge \forall f \in \mathrm{dom}(N_1), \\ & \forall s \in \mathscr{V}^\infty, N_1(f)(s) \subseteq N_2(f)(s) \end{aligned} \tag{4}$$

The theorem on semantics conformance is expressed as:

Theorem 1 *For all programs P,* $[\![\mathrm{Tr}(P)]\!] \sqsubseteq [\![P]\!]$

Proof The proof relies on induction on sequences of node definitions. The base case is a single node program, without sub-nodes applications. We first prove that then, the definition of DCS is sufficient to ensure conformance of the compiled node with the semantics.

– **Base step:** case where $P = d$:

$$\begin{aligned} d = \ & \texttt{node}\ f(X)\texttt{=}(Y) \\ & \texttt{contract}\ (D_1, A, G) \\ & \texttt{with}\ c \\ & \texttt{let}\ D_2\ \texttt{tel} \end{aligned}$$

D_2 has no sub-nodes applications, hence $\mathrm{Tr}(D_2) = (\mathscr{S}_2, \emptyset, \emptyset)$ as (C-App) is the only rule adding uncontrollable variables and synthesis objectives. Let $\mathrm{Tr}(D_1) = (\mathscr{S}_f^c, \emptyset, \emptyset)$ and $\mathscr{S} = \mathscr{S}_2 \| \mathscr{S}_f^c$ and $C = \mathrm{DCS}(\mathscr{S} \triangleright A, \{c\}, G)$. By definition of DCS, we have:

$$\mathrm{Traces}((\mathscr{S} \triangleright A)/C) \subseteq \{s \in \mathrm{Traces}(\mathscr{S} \triangleright A) \text{ s.t. } G(s)\} \tag{5}$$

From the definition of $\mathscr{S} \triangleright A$, we have

$$\{s \in \text{Traces}(\mathscr{S} \triangleright A) \text{ s.t. } G(s)\} \subseteq \{s \in \text{Traces}(\mathscr{S} \triangleright A) \text{ s.t. } A(s) \Rightarrow G(s)\} \quad (6)$$

Let s and $\rho = \{x \mapsto s\}$, such that s verifies A and let s_c and $\rho' = \rho \oplus \{c \mapsto s_c\}$ such that $[\![D_1 ; D_2]\!]^{\emptyset}_{\rho'}$ is defined and satisfies $A \Rightarrow G$.
As D_2 contains no applications, the translation towards STS preserves the semantics: $[\![D_2]\!]^{\emptyset}_{\rho'} = \text{Traces}(\mathscr{S}_2 \triangleright A)$. In a similar way: $[\![D_1]\!]^{\emptyset}_{\rho'} = \text{Traces}(\mathscr{S}^c_f \triangleright A)$. Thus, $[\![D_1 ; D_2]\!]^{\emptyset}_{\rho'} = \text{Traces}(\mathscr{S} \triangleright A)$. Hence $[\![D_1 ; D_2 ; c = C]\!]^{\emptyset}_{\rho} = \text{Traces}((\mathscr{S} \triangleright A)/C)$ which is the left-hand side of (5). Also, $[\![D_1 ; D_2]\!]^{\emptyset}_{\rho'} = \text{Traces}(\mathscr{S} \triangleright A)$, as featured in the right-hand side of (6).
As a consequence, we have:

$$[\![D_1 ; D_2 ; c = C]\!]^{\emptyset}_{\rho} \subseteq \{[\![D_1 ; D_2]\!]^{\emptyset}_{\rho \oplus \{c \mapsto s_c\}} \text{ s.t. } A \Rightarrow G\}$$

As $D_1 ; D_2 ; c = C$ is the body of $\text{Tr}(d)$, and right-hand side is the semantics of d, we conclude that:

$$[\![\text{Tr}(d)]\!]^{\emptyset} \sqsubseteq [\![d]\!]^{\emptyset}$$

– **Inductive step:** case where $P = d_1 \dots d_n$.
Let $N = [\![d_1 \dots d_{n-1}]\!]$ and $N' = [\![T(d_1 \dots d_{n-1})]\!]$. By induction hypothesis, we have $N' \sqsubseteq N$. Let

$$\begin{aligned} d_n = \ & \textbf{node } f(X)\text{=}(Y) \\ & \quad \textbf{contract } (D_1, A, G) \\ & \quad \textbf{with } c \\ & \textbf{let } D_2 \textbf{ tel} \end{aligned}$$

We focus on node applications and definitions, as other semantic rules are defined with operations preserving set inclusions. Particularly, parallel sub-node applications will be handled by rule (C-App). We assume that $D_2 = \ y = f(x)$.
Let s and $\rho = \{x \mapsto s\}$, such that s satisfies A. From rule (C-App) we have $\text{Tr}(y = f(x)) = (\mathscr{S}_2, \{y\}, A_f)$ where $\mathscr{S}_2 = \mathscr{S} \| \mathscr{S}^c_f$ and $\mathscr{S}(\{y\}, \emptyset, \{z\}) = \begin{cases} z = y \\ A_f \Rightarrow G_f \end{cases}$
Let $(\mathscr{S}_1, \emptyset, \emptyset) = Tr(D_1)$, $C = \text{DCS}(\mathscr{S}_1 \| \mathscr{S}_2 \triangleright A, \{c\}, G \wedge A_f)$.
$[\![y = f(x)]\!]^{N'}_{\rho}$ is defined, since A_f is a synthesis objective (hence, A_f is enforced by C). From the induction hypothesis:

$$[\![y = f(x)]\!]^{N'}_{\rho} \subseteq \{[\![y = f(x)]\!]^{N}_{\rho \oplus \{c \mapsto s_c\}} \text{ s.t. } A_f \Rightarrow G_f\}$$

Then, traces from $[\![y = f(x)]\!]^{N'}_{\rho}$ satisfy $A_f \Rightarrow G_f$, and:

$$[\![y = f(x)]\!]^{N'}_{\rho} \subseteq \text{Traces}(\mathscr{S} \| \mathscr{S}^c_f).$$

Then, $[\![D_1 ; D_2]\!]^{N'}_{\rho} \subseteq \text{Traces}(\mathscr{S}_1 \| \mathscr{S}_2)$.
As by definition of DCS,

$$\begin{aligned} \text{Traces}(\mathscr{S}_1 \| \mathscr{S}_2 / C) \subseteq & \{\text{Traces}(\mathscr{S}_1 \| \mathscr{S}_2) \text{ s.t. } A \Rightarrow G \wedge A_f\} \\ \subseteq & \{\text{Traces}(\mathscr{S}_1 \| \mathscr{S}_2) \text{ s.t. } A \Rightarrow G\} \end{aligned}$$

then:

$$[\![D_1 ; D_2 ; c = C]\!]^{N'}_{\rho} \subseteq \{[\![D_1 ; D_2]\!]^{\emptyset}_{\rho \oplus \{c \mapsto s_c\}} \text{ s.t. } A \Rightarrow G\}$$

and $[\![\mathrm{Tr}(d_n)]\!]^{N'} \sqsubseteq [\![d_n]\!]^{N}$.

5.4 Implementation

5.4.1 Compilation process

The compilation process is organized as shown in Figure 16.

A BZR program is compiled into two parts. The first part is the classical sequential code resulting from the compilation of the synchronous imperative part (automata and equations). The second part is the translation of automata and contracts into transition systems (STSs) and synthesis objectives. This part is used by the DCS tool (SIGALI) to produce a controller (constraint on inputs, states and controllable variables), which is then determinized and translated towards synchronous equations. Thus, the controller itself is produced as a BZR program (without contract) and can then in turn be compiled towards sequential code, in C or in Java; it is possible to develop simple back-ends for other target languages. The two sequential parts (from automata and the controller) can then be composed by simple link edition, defining their synchronous composition.

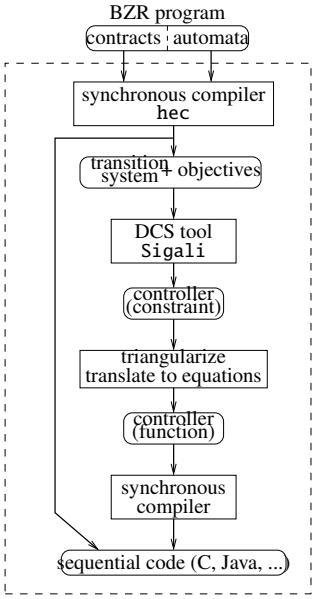


Fig. 16 BZR compilation process.

This compilation process is modular, meaning that this process is applied on each node, independently on (i) the body of its non-inlined subnodes and (ii) its calling context in upper-level nodes.

5.4.2 Costs issues

At its kernel, our compiler calls the DCS tool SIGALI. The complexity of the involved algorithms is exponential in the number of variables, just like other comparable operations like model checking. On a more practical level, our techniques benefit from the level of abstraction and granularity of control which is handled: we manage just the reactive control kernel, not the whole system, thereby modelling key things, abstracting the rest. Therefore the size of the controller on an adaptive system is much smaller than the more data-related parts.

The table 1 shows the synthesis time and controller size on some examples[4]. The four first columns give the number of variables (state variables, input and controllable variables, and total number), thus giving an indication about the size of the example. Although the synthesis cost is theoretically exponential, we can see that for examples of reasonable sizes (30 state variables correspond to, e.g., the parallel composition of 10 automata of 5 to 8 states each), the synthesis time is of the order of seconds. In the example of Section 6, the full compilation (synthesis included) takes hardly a second.

The size of the controller generated is theoretically exponential in the number of variables ; but its online execution is linear. On the examples given on table 1, all execution times for one step of the controlled system (thus including execution of the controller itself) are of the order of few μseconds.

Example name	# state vars	# inputs	# cont.	total # vars	synthesis time (s)	contr. size (# C loc)
bzradmin	20	7	3	30	0.11	494
bzrlang (Sec. 6)	47	6	5	58	0.52	2502
cellphone	29	17	6	52	0.61	2346
radiotrans	14	13	2	29	0.10	428
migration [7]	220	7	8	235	1188.06	49660
httpserver [13]	36	2	3	41	0.11	659
robot arm [1]	31	23	3	57	0.13	422
provadm	11	5	2	18	0.01	197
prog2 [12]	8	4	2	14	0.01	217
prog4 [12]	16	8	4	28	0.06	506
prog6 [12]	24	12	6	42	0.40	986
prog8 [12]	32	16	8	56	74.07	1535
prog10 [12]	40	20	10	70	0.97	2134
prog12 [12]	48	24	12	84	2.75	3051
prog14 [12]	56	28	14	98	5.73	3976
prog16 [12]	64	32	16	112	2990.58	6341

Table 1 Synthesis time and controller size of different examples

[4] Experiments carried out on a 64 bits dual-core PC, 2.93 GHz, with 3.8 Gb of RAM

Moreover, we have some more thorough experiments regarding performance and size of systems that can be managed [12], which show that scalability is greatly enhanced by modular synthesis.

5.4.3 Back end: executable code

At the back end, code generators from the synchronous compiler can be used, producing typically, in C or Java, a function for the reactive step, and a function for the initialization of state variables. The generated executable controller is integrated into the adaptive system, linked with the particular host operating system and computation model, and with functional (non-control) code. We can recall that for a synchronous program, there are two ways of interacting with an execution platform:

- one of them is the calling of external functions in the host language from the synchronous program: this enables interfacing easily with the non-synchronous world, typically for features not available in the synchronous languages, e.g., arbitrary functions and types (e.g., involving pointers and dynamical data structures), libraries, numerical computations, side-effects. In this scheme, care has to be taken that the synchrony hypothesis is respected, i.e., the functions have to be guaranteed to return in bounded time.
- the other is being called from the global executive, initialization or resetting, and then each step of the reactive controller is launched by calling the generated controller code; this has to be done at appropriate control points in the system. This involvesthe following phases:
  - the input event has to constructed e.g., by reading queues or buffers, or testing flags set by callbacks since the last step.
  - then the step is called e.g., from the body of a loop (which can be infinite), or attached to an exception handling mechanism, or through an interrupt.
  - finally, there is an interpretation of the output event, e.g., by the emission of signals, call of functions, raising of flags.

  It is possible to have several controllers obtained separately that way in the same system, but their interactions can be taken into account by synchronous compilation or DCS only if they are assembled in the same global program. No conflicts are generated as long as only safety properties are considered.

We have experiences in such integrations in various contexts. In an experiment with the design environment Orccad for control systems, on top of a Posix real-time operating system [1], the *step* function is called each time an event is placed in the automaton input FIFO. With the component-based framework Fractal, in its C implementation developed in the MIND project [13], we instrumented the controlled system with monitors related to the occupation of FIFOs, and these events are fed to the *step* function. We are currently exploring the autonomic administration of deployment of a virtual machine.

6 Example

Our language introduces a different, unusual programming methodology. Whereas classically computer programming consists of writing a control solution, in BZR we specify the problem. This is related to the control theory way of approaching things:

- first write nodes that describe the process to be controlled (the "plant"), with all its possible, uncontrolled behaviors; thereby identify its possible control points, independently of their use;
- then write contracts that specify control objectives or desired behaviors; it can be noted that different objectives can make sense for the same "plant", and that controllability of the plant for the given objective is not always given;
- compile the program to derive the controller, using DCS; like type synthesis does for types, control synthesis can be described as a form of completion of the control automaton, that was uncompletely specified.

This section illustrates these points, in the specific domain of embedded systems, with the example of a robot manipulator arm.

6.1 The robot arm case study

Our example is a simplified form of a case study [1] concerning a robot arm, the articulations of which define a mechanically reachable workspace. Such a robot must always be under the control of a control law, otherwise the movements would become erratic, depending on gravity, wind or any mechanical forces around. There is also an exclusion constraint between these control laws, the actuator being an exclusive resource for them. They are implemented in real-time tasks, that can be started, and which can emit a termination event when their goal is reached, or predefined exception events.

We consider six such control tasks. The robot arm can move its end, carrying a tool, inside the workspace, using control based on Cartesian coordinates (C). However, some movements can lead its articulations to their limits, called singularities, which causes an exception event to be raised. This requires to make an intermediary move in order to turn around a singularity, using a different control, based on joint coordinates (angles of the articulations) (J). These two control laws are grouped in task CJ. Another possible control consists of trajectory tracking, used typically for pointing towards a target outside the workspace (F). A second task of the same kind takes care of positions on the borderline of the workspace (B). Another task is defined for the change of tools (CT): it includes moving to the tool rack, and actually 1 taking the tool. There are two tools available: one is a gripper, and can be used to grip the target when it is inside the workspace; the other is a camera, which can be pointed towards the target when it is outside. Finally, a background task can be activated in the absence of other control laws, to maintain the current position (M) (as a robot must never be out of control).

The application for this robot system consists in, when a target is indicated: if it inside the workspace, go and grip it with the gripper; if borderline, go to a central

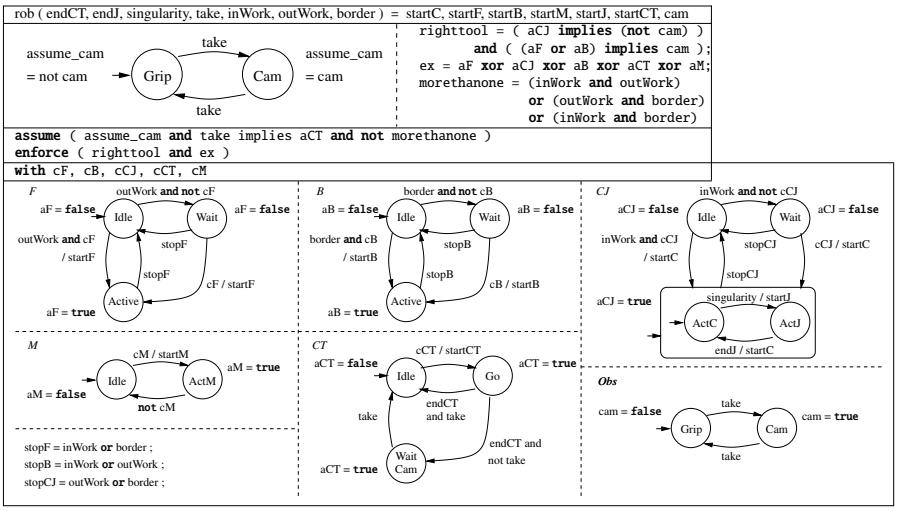


Fig. 17 Example of the robot controller: BZR node, with contract.

position with the camera aimed at it; or else if outside, extend to the border and point at it with the camera.

6.2 Behaviors

Figure 17 shows the BZR node for the case study. The body describes the behaviors of the different underlying real-time control tasks, at the level of abstraction of their activation, which is appropriate for managing their interactions. From left to right we have first, for task F, a simple variation of the delayable task of Figure 4: from an initial inactive state, upon reception of the input `outWork` signaling a target outside of the workspace, a transition is taken according to the choice variable (to be controlled) `cF`: if **`true`** then the output `startF` is sent out to the real-time tasks handler, the trajectory following control law is started, and the next state is `Active`; otherwise it is **`false`**, and then control goes to the `Wait` state, from where, when `cF` is **`true`**, the task can be started. From both `Active` and `Wait`, the reception of `stopF` causes a transition back to `Idle`. For B, when the target is at the border, we have a second instance of the same behavior. The next one describes CJ, the movements inside the workspace: it also follows the delayable pattern, with the choice variable `cCJ`, and a more elaborate active state: it is hierarchically refined into a sub-automaton, where initially the Cartesian control task is active, and upon reception of the exception `singularity`, a switch is made to the joint control task; upon termination, control reverts to the Cartesian mode.

On the lower side of the node, we have the automaton for the M task, where start and end are controllable through `cM`. Underneath we have some equations defining the stopping of tasks. Next to it is the automaton for the tool change task CT: it can be triggered by the controller using variable `cCT`. Once active, when arriving at the

tool rack upon endCT, it either takes the tool if it is available (when take is **true**), or waits there until it is. We also have an observer automaton *Obs* for the current tool, switching states each time a new tool is taken.

This parallel automaton describes all possible sequencings of the tasks: 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.

6.3 Contract

The application must launch robot tasks corresponding to the current state of the target (inside, outside or at the border of the workspace) and change the tool to get the right one for each task. So the control objective is first to ensure that we have the right tool, and second, to allow at most one task to be active at a time, and also at least one, as mentioned in Section 6.1.

The set $\{cF, cB, cCJ, cCT, cM\}$ of local controllable variables, defined in the with part, is used for ensuring this objective. The contract specifies that the node will be controlled, such that, given any uncontrollable input trace, the output trace will satisfy the two objectives. It can be seen in the upper part of Figure 17: it is itself a program, with equations defining variables. For the right tool for the right task, a Boolean variable righttool is defined as the conjunction of two implications: they state that when a task is active (aCJ, respectively aF or aB), it implies that the arm carries the right tool (not cam, respectively cam). For mutual exclusion and default control, an equation defines ex, which is the exclusive disjunction of active states for the tasks. The contract also has an automaton, which will be visible when the node is re-used, and makes the relation between cam and take. Given that only the body of the node can produce outputs, we keep the observer there to produce cam, so these two automata have different roles.

The assumption is that: assume_cam is **true**, which makes the relation between the two automata mentioned above; the input take is only present when *CT* has been activated (i.e., correspond to actual tool changes); only one of the inputs inside, outside and borderline is true at the same time. The contract is to enforce both Boolean expressions.

6.4 Simulation and typical scenario

Here is a typical scenario showing the intervention of the controller on the system, so that control objectives are enforced. At some point task *CJ* is active, the target inside the workspace, and the tool carried by the arm corresponds to not cam. Then, the user clicks outside of the workspace, so the application receives the outWork input. This causes the flow stopCJ to be **true**, and the automaton for *CJ* to move by the transition conditioned by stopCJ to its Idle state.

It also causes the automaton for task *F* to quit its initial state; here, we have a choice point conditioned by cF. Due to the first contract property, righttool must be kept true, so given that the current tool is not cam, the controller can not allow

the transition to `Active` of F, and must give the value **`false`** to `cF`. Hence task F goes to its `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 `cCT`, which will change the tool.

In a later reaction, at the end of the task CT, with the `endCT` event, if `take` is **`true`**, the automaton observing the current tool goes to a state where `cam` is **`true`**. Thus we have the right tool for task F, and the controller can release F from `Wait` to `Active`, by giving value **`true`** to controllable variable `cF`.

This shows how mutual exclusion and, more interestingly because it is dynamical, insertion of a reconfiguration task between two other tasks, can be obtained.

6.5 Example of modular contracts

We illustrate modular contracts by considering two robot systems, sharing the camera tool, while each has its own gripper. The model for such a robot workshop is illustrated in Figure 18, where two instances of the `rob` node are in parallel. The contract simply says that the exclusivity of `cam1` and `cam2` should be enforced, with no further assumption, with the controllables `take1` and `take2`.

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

Fig. 18 Two robots sharing an exclusive camera.

Another modular contract example has been developed, with simplified behaviors involving only delayable tasks, but showing the use of modularity, and also the methodology: we first constructed a contract node for n such tasks, and then built a $2n$ tasks node, with a first contract that revealed itself being not controllable, and then refinements of the problem leading to a solution. On this example performance evaluation showed a drastically improved scalability of the approach [12].

7 Related work

As was noted by other authors, while classical control theory has been readily applied to computing systems [20], applying Discrete Control Theory to computing systems is more recent: some focus on controlling multi-thread code [3, 15] or workflow scheduling [39], or on the use of Petri nets [23, 22, 30] or finite state automata [36]. The work closest to ours [40] is a programming language-level approach, that focuses on deadlock avoidance in shared-memory multi-threaded programs, and relies upon Petri net formal models, where control logic is synthesized, in the form of additional

control places in the Petri nets, in order to inhibit behaviors leading to interlocking. A difference in motivation is that they apply Discrete Control internally to the compilation, only for deadlock avoidance, in a way independent of the application, whereas we treat expression of objectives as a first class programming language feature: we know of no other programming language doing this.

Some related work can be found in computer science, in the notions of program synthesis. It consists in translating a property on inputs and outputs of a system, expressed in temporal logics, into a lower-level model, typically in terms of transition systems. For example, it is proposed as form of liberated programming [17] in a UML-related framework, with the synthesis of StateChart from Live Sequence Charts [18,27]. Other approaches concern angelic non-determinism [6], where a non-deterministic operator is at the basis of refinement-based programming. These program synthesis approaches do not seem to have been aware of Discrete Control Theory, or reciprocally: there seems to be a relationship between them, as well as with game theory, but it is out of the scope of this paper. One difference is that we synthesize a constraint (on the controllable variables) from a state machine (given as a model of the object to be controlled) and a control objective (safety), as usual in the control approach [37,40]. In this sense, our language is mixed imperative (writing the automata for not yet controlled components) and declarative (specifying the properties to be enforced by control). Also, a meaningful difference is that we distinguish between controllables and uncontrollables, which is more general. On the other hand, we consider only safety properties; we are aware that it is possible to consider liveness properties in synthesis, but we feel that it is more difficult to handle it in a compositional and modular way. These declarative approaches encounter methodological problems of incomplete specification, complexifying the obtention of the state machine, whereas we obtain a maximal permissive controller (meaning a minimal constraints on behaviors, which is a relation). However, when the control objectives are not tight, we also have to find ways for completion of the constraint to make it a function (deterministic) of uncontrollable inputs. The readability of state machines synthesized in this work can be a motivation [18], whereas we do not expect our automatically generated constraint to be read.

The notion of design by contracts has been introduced first in the Eiffel language [35]; contracts are require/ensure pairs on Eiffel functions which are then used at compilation time to add defensive code to these functions. The same design principle have been extended for reactive systems in [31], where reactive programs are given logical-time contracts, validated automatically by model-checking. We use here the same principle of logical-time contract, the difference with this latter work is essentially that our contracts are enforced by controller synthesis, instead of being validated. A more generic model of contracts has been proposed in [4], defining an algebra of contracts, which allows to consider the relation between sets of contracts defining one system, whereas our language only allows one contract to be associated to one node. Interface synthesis [10] is also related to our approach, consisting in generating interfacing wrappers for components, in order to adapt them for the composition into given component assemblies, w.r.t. the communication protocols between them. The difference is that this work is about identifying constraints on the environment of a component so that it is used correctly, whereas we constrain

the component so that it works correctly whatever the environment does (within the assumptions).

Performing control using modularity/hierarchy and abstraction has also been subject to various studies [25,26,38,14,24,34,29]. For the modularity and hierarchical aspect, the difference lies in the model that is used (asynchronous versus synchronous automata) as well as in the abstraction techniques. In our case, abstraction consists in abstracting the sub-systems by their contract, the abstraction techniques used in control theory consists projecting the behavior of the sub-system to some sub-alphabet and computing controllers on the resulting abstracted systems. If both theory share hierarchical, modularity and abstraction features, the underlying techniques are thus completely different.

8 Conclusion and perspectives

We propose an original contribution on the role of formal methods in software and systems engineering: we encapsulate the formal DCS method into a language compilation process. This way, it is integrated into a development process, where the user/programmer is provided with tool support of the formal technique of DCS, and the generation of executable code. The tool is concretely built upon the basis of a reactive programming language compiler, where the nodes describe behaviors that can be modeled in terms of transition systems. Our compiler integrates this with a DCS tools, making it a new environment for formal methods. For this, we define a construct for behavioral contracts in reactive programs, enabling mixed imperative/declarative programming. We thereby exploit the dynamical behavior of programs in the compilation, by using state and trace-based models of their control.

Future and ongoing work in this new research direction is addressing the limitations of our current results. We are addressing language-level expressiveness, notably w.r.t. quantitative aspects: we already have features of cost functions for bounding or one-step optimal control, but timed aspects would be an improvement (see e.g. [9]). We could exploit more powerful DCS techniques, e.g., dynamical controller synthesis (i.e., relying on more states than in the automaton to be controlled), and combination with static analysis and abstract interpretation techniques as in [28]. We are exploring distributed execution schemes for controllers programmed in BZR. Assistance and diagnosis in this uncommon programming style is a very interesting issue: several situations can lead to compilation failure (e.g., DCS failure), or unsatisfying result (e.g., too restrictive controller). Currently, it can happen that a program in BZR can not be compiled because the control problem has no solution : then the compiler returns an error message and no code is generated. The user has to debug the program, by relaxing the contract, or changing the behaviors. Tools and precise methodologies should be developed so as to handle such situations.

We have ongoing work exploring the application of our language for adaptive systems at different levels: control of FPGA(Field Programmable Gate Array)-based reconfigurable architectures, design and coordination of administration loops in virtual machines, component-based adaptive middleware [7] and control and robot systems design [1].

References

1. S. Aboubekr, G. Delaval, R. Pissard-Gibollet, E. Rutten, and D. Simon. Automatic generation of discrete handlers of real-time continuous control tasks. In *Proc. 18th World Congress of the International Federation of Automatic Control (IFAC),* Milano, Italy, pages 786–793, August 2011.
2. K. Altisen, A. Clodic, F. Maraninchi, and E. Rutten. Using controller synthesis to build property-enforcing layers. In *European Symposium on Programming*, volume 2618 of *LNCS*, pages 126–141, Warsaw, Poland, April 2003.
3. A. Auer, J. Dingel, and K. Rudie. Concurrency control generation for dynamic threads using discrete-event systems. In *Communication, Control, and Computing, 2009. Allerton 2009. 47th Annual Allerton Conference on*, pages 927 –934, 30 2009-oct. 2 2009.
4. A. Benveniste, B. Caillaud, and R. Passerone. A generic model of contracts for embedded systems. Res. Rep. RR-6214, INRIA, 2007.
5. 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):64 – 83, January 2003.
6. R. Bodik, S. Chandra, J. Galenson, D. Kimelman, N. Tung, S. Barman, and C. Rodarmor. Programming with angelic nondeterminism. In *Principles of Programming Languages, POPL*, pages 339–352, January 2010.
7. T. Bouhadiba, Q. Sabah, G. Delaval, and E. Rutten. Synchronous control of reconfiguration in fractal component-based systems – a case study. In *Int. Conf. on Embedded Software. EMSOFT 2011*, pages 309–318, Taipei, Taiwan, October 2011.
8. C. Cassandras and S. Lafortune. *Introduction to Discrete Event Systems*. Springer, 2007.
9. F. Cassez, A. David, E. Fleury, K. Larsen, and D. Lime. Efficient on-the-fly algorithms for the analysis of timed games. In *Conf. on Concurrency Theory (CONCUR)*, volume 3653 of *LNCS*, pages 66–80, August 2005.
10. A. Chakrabarti, L. de Alfaro, T. A. Henzinger, and F. Y. C. Mang. Synchronous and bidirectional component interfaces. In *Computer Aided Verification*, volume 2404 of *LNCS*, pages 414–427, Copenhagen, Denmark, July 2002.
11. J.-L. Colaço, B. Pagano, and M. Pouzet. A Conservative Extension of Synchronous Data-flow with State Machines. In *Embedded Software (EMSOFT)*, pages 173 – 182, New Jersey, USA, September 2005.
12. G. Delaval, H. Marchand, and E. Rutten. Contracts for modular discrete controller synthesis. In *Languages, Compilers and Tools for Embedded Systems*, pages 57–66, Stockholm, Sweden, April 2010.
13. G. Delaval and E. Rutten. Reactive model-based control of reconfiguration in the Fractal component-based model. In *Component Based Software Engineering,* Prague, June, volume 6092 of *LNCS*, pages 93–112, Prague, Czech R., June 2010.
14. M.H. deQueiroz and J.E.R. Cury. Synthesis and implementation of local modular supervisory control for a manufacturing cell. In *Proceedings of the 6th International Workshop on Discrete Event Systems*, pages 377–382, October 2002.
15. C. Dragert, J. Dingel, and K. Rudie. Generation of concurrency control code using discrete-event systems theory. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, SIGSOFT '08/FSE-16, pages 146–157, New York, NY, USA, 2008. ACM.
16. G. Hamon. *Calcul d'horloge et structures de contrôle dans Lucid Synchrone, un langage de flots synchrones à la ML.* PhD thesis, Univ. P. et M. Curie, Paris, France, November 2002.
17. D. Harel. Can programming be liberated, period? *Computer*, 41(1):28–37, 2008.
18. D. Harel, H. Kugler, and A. Pnueli. Synthesis revisited: Generating statechart models from scenario-based requirements. In *Formal Methods in Software and Systems Modeling*, volume 3393 of *LNCS*, pages 309–324, 2005.
19. D. Harel and A. Naamad. The STATEMATE semantics of statecharts. *ACM Transactions on Software Engineering and Methodology*, 5(4):293–333, 1996.
20. J. Hellerstein, Y. Diao, S. Parekh, and D. Tilbury. *Feedback Control of Computing Systems*. Wiley-IEEE, 2004.
21. Y. Hietter, J.-M. Roussel, and J.-J. Lesage. Algebraic Synthesis of Transition Conditions of a State Model. In *Proc. of 9th Int. Workshop On Discrete Event Systems (WODES'08)*, pages 187–192, Göteborg, June 2008.
22. M. Iordache and P. Antsaklis. Concurrent program synthesis based on supervisory control. In *2010 American Control Conference*, 2010.

23. M. V. Iordache and P. J. Antsaklis. Petri nets and programming: A survey. In *Proceedings of the 2009 American Control Conference*, pages 4994–4999, 2009.
24. S. Jiang and R. Kumar. Decentralized control of discrete event systems with specializations to local control and concurrent systems. *IEEE Transactions on Systems, Man, and Cybernetics, Part B*, 30(5):653–660, October 2000.
25. J. Komenda, T. Masopust, and J.H. van Schuppen. Synthesis of safe sublanguages satisfying global specification using coordination scheme for discrete-event systems. *Discrete Event Dynamical Systems*, 10:426–431, 2010.
26. J. Komenda and J.H. van Schuppen. Supremal sublanguages of general specification languages arising in modular control of discrete-event systems. In *44th IEEE Conference on Decision and Control*, pages 2775–2780, 2005.
27. H. Kugler, C. Plock, and A. Pnueli. Controller synthesis from LSC requirements. In *Fundamental Approaches to Software Engineering, FASE'09, York, UK, March 22-29*, 2009.
28. T. Le Gall, B. Jeannet, and H. Marchand. Supervisory control of infinite symbolic systems using abstract interpretation. In *44nd IEEE Conference on Decision and Control (CDC'05) and Control and European Control Conference ECC 2005*, pages 31–35, Seville, Spain, December 2005.
29. S.-H. Lee and Wong K.C. Structural decentralized control of concurrent discrete-event systems. *European Journal of Control*, 8(5), 2002.
30. Cong Liu, A. Kondratyev, Y. Watanabe, J. Desel, and A. Sangiovanni-Vincentelli. Schedulability analysis of petri nets based on structural properties. In *Application of Concurrency to System Design, 2006. ACSD 2006. Sixth International Conference on*, pages 69 –78, june 2006.
31. F. Maraninchi and L. Morel. Logical-time contracts for the development of reactive embedded software. In *30th Euromicro Conference, Component-Based Software Engineering Track (ECBSE)*, pages 48–55, Rennes, France, September 2004.
32. H. Marchand. *Méthodes de synthèse d'automatismes décrits par des systèmes à événements discrets finis*. PhD thesis, Université de Rennes 1, IFSIC, October 1997.
33. H. Marchand, P. Bournai, M. Le Borgne, and P. Le Guernic. Synthesis of discrete-event controllers based on the signal environment. *Discrete Event Dynamic Systems: Theory and Applications*, 10(4):325–346, October 2000.
34. H. Marchand and B. Gaudin. Supervisory control problems of hierarchical finite state machines. In *41th IEEE Conference on Decision and Control*, pages 1199–1204, Las Vegas, USA, December 2002.
35. B. Meyer. Applying "design by contract". *Computer*, 25(10):40–51, October 1992.
36. V.V. Phoha, A.U. Nadgar, A. Ray, and S. Phoha. Supervisory control of software systems. *Computers, IEEE Transactions on*, 53(9):1187 – 1199, sept. 2004.
37. P. J. Ramadge and W. M. Wonham. Supervisory control of a class of discrete event processes. *SIAM J. Control Optim.*, 25(1):206–230, 1987.
38. K. Schmidt and C. Breindl. On maximal permissiveness of hierarchical and modular supervisory control approaches for discrete event systems. In *Discrete Event Systems, 2008. WODES 2008. 9th International Workshop on*, pages 462–467. IEEE, 2008.
39. C. Wallace, P. Jensen, and N. Soparkar. Supervisory control of workflow scheduling. In *Advanced Transaction Models and Architectures Workshop (ATMA), Goa, India*, 1996.
40. Y. Wang, S. Lafortune, T. Kelly, M. Kudlur, and S. Mahlke. The theory of deadlock avoidance via discrete control. In *Principles of Programming Languages, POPL*, pages 252–263, Savannah, USA, 2009.