HAL
open science

BZR Contracts for Modular Discrete Controller Synthesis

Gwenaël Delaval, Hervé Marchand, Eric Rutten

To cite this version:

Gwenaël Delaval, Hervé Marchand, Eric Rutten. BZR Contracts for Modular Discrete Controller Synthesis. [Research Report] RR-7111, INRIA. 2009. inria-00436560

HAL Id: inria-00436560

https://inria.hal.science/inria-00436560v1

Submitted on 27 Nov 2009

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

BZR Contracts for Modular Discrete Controller Synthesis

Gwenaël Delaval — Hervé Marchand — Eric Rutten

N° 7111

November 24, 2009

Thème COM

Rapport de recherche

ISSN 0249-6399 ISRN INRIA/RR--7111--FR+ENG

DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE

INRIA

centre de recherche
RENNES - BRETAGNE ATLANTIQUE

BZR Contracts for Modular Discrete Controller Synthesis*

Gwenaël Delaval , Hervé Marchand , Eric Rutten

Thème COM — Systèmes communicants
Équipes-Projets VerTeCs and Sardes

Rapport de recherche n° 7111 — November 24, 2009 — 25 pages

Abstract: We describe the extension of a reactive programming language with a behavioral contract construct. It is particularly dedicated to the programming of reactive control of applications in embedded systems, and involves principles of the supervisory control of discrete event systems. Our contribution is in a language approach where modular discrete controller synthesis (DCS) is integrated, and it is concretized in the encapsulation of DCS into a compilation process. From transition system specifications of possible behaviors, DCS automatically produces controllers that make the controlled system satisfy the property given as objective. Our language features and compiling technique hence provide correctness-by-construction in that sense, and enhance reliability and verifiability. An application domain particularly targeted at is that of adaptive and reconfigurable systems: closed-loop adaptation mechanisms enable flexible execution of functionalities w.r.t. changing resource and environment conditions. Our language can serve programming such adaption controllers. This paper particularly describes the compilation of the language. We present a method for the modular application of discrete controller synthesis on synchronous programs, and its integration in the BZR language. We consider structured programs, as a composition of nodes, and first apply DCS on particular nodes of the program, in order to reduce the complexity of the controller computation; then, we allow the abstraction of parts of the program for this computation; and finally, we show how to recompose the different controllers computed from different abstractions for their correct co-execution with the initial program. Our work is illustrated with examples, and we present quantitative results about its implementation.

Key-words: Discrete controller synthesis, modularity, components, contracts, synchronous programming, adaptive and reconfigurable systems

* This work was partially supported by the Minalogic project MIND.

Centre de recherche INRIA Rennes – Bretagne Atlantique
IRISA, Campus universitaire de Beaulieu, 35042 Rennes Cedex
Téléphone : +33 2 99 84 71 00 — Télécopie : +33 2 99 84 71 71

Contrats BZR pour la synthèse de contrôleurs discrets modulaire

Résumé : Nous décrivons l'extension d'un langage de programmation réactive par un constructeur de contrat comportemental. Elle vise particulièrement la programmation du contrôle réactif d'applications dans les systèmes embarqués, et comporte des principes de contrôle supervisé des systèmes à événements discrets. Notre contribution est dans une approche orientée langage où la synthèse de contrôleurs discrets (SCD) est intégrée, et est concrétisée par l'encapsulation de la SCD dans le processus de compilation. A partir de spécifications des comportements possibles sous forme de systèmes de transitions, la SCD produit automatiquement des contrôleurs qui font que le systême contrôlé satisfait la prpriété donnée comme objectif de contrôle. Les instructions de notre langage et sa compilation assurent donc en ce sens la correction par construction, et favorisent la fiabilité et la vérifiabilité. Un domaine d'application particulièrement visé est celui des systèmes adaptatifs et reconfigurables : des mécanismes d'adaptation en boucle fermée permettent l'exécution flexible de fonctionalités vis-à-vis de ressources et de conditions d'environnement changeants. Notre langage peut servir à programmer de tels contrôleurs d'adaptation corrects.

Ce rapport décrit plus particulièrement la compilation du langage. Nous présentons une méthode pour l'application modulaire de la SCD sur des programmes synchrones, et son intégration dans la langage BZR. Nous considérons des programmes structurés, comme une compositon de nœuds, et appliquons d'abord la SCD aux nœuds particuliers du programme, de façon à réduire la complexité du calcul du contrôleur; ensuite, nous permettons l'abstraction de parties du programme pour ce calcul; enfin, nous montrons comment recomposer les différents contrôleurs calculés à partir des différentes abstractions, pour leur co-exécution correcte avec le programme initial. Notre travail est illustré par des exemples, et nous présentons des résultats quantitatifs quant à sa mise en œuvre.

Mots-clés : Synthèse de contrôleurs discrets, modularité, contrats, programmation synchrone, systèmes adaptatifs et reconfigurables

1 Motivations

We integrate discrete controller synthesis (DCS) into a modular compilation process for a synchronous language: BZR, with motivations concerning the design of programming languages, the use of DCS, and the control of adaptive and reconfigurable systems.

Programming languages. We propose a compilation concretely exploiting a representation of the dynamical behavior of the program. Whereas classically, compilation considers properties holding for all states (i.e., static), we propose a technique to consider, in the compilation, state and path-dependent aspects (i.e., dynamical) [20].

DCS is a constructive operation, as it computes not a diagnostic about correctness, but a correct solution. A few works exist about its integration into a programming language framework [3]. We associate it to a contract mechanism, making it easier to use by programmers and favoring scalability (see further). Symmetrically, we propose a new point of view on the design by contracts principle: our programming language allows contracts to be enforced in non-deterministic programs, instead of being checked or proved correct.

Discrete controller synthesis. The modular application of DCS, which we are addressing in this paper, is based on contract enforcement and abstraction of components, with the aim of improving the scalability of the techniques devoted to DCS. Furthermore, the integration of these techniques in a high-level programming language also contributes to make it more widely useable in computer systems, and to study implementations of the controllers at a higher level than programmable logic controllers used in automation.

Control of adaptive and reconfigurable systems. Embedded systems have to be predictable, for safety-criticality issues. They also more and more have to be able of dynamical adaptivity and reconfiguration, in reaction to environment changes, related to resources or dependability. This requires abilities for sensing the state of a system, deciding, based on a representation of the system, upon reconfiguration actions, and performing and executing them. These functionalities are assembled into a decision loop as illustrated in Figure 1(a).

(a) Adaptive system.

(b) BZR programming.

Figure 1: BZR programming of adaptation control.

Approach followed in our work. We want to combine these two different requirements, i.e., to be *adaptive and predictable*. Our programming language is specially suited for user-friendly design of safe control loops of adaptive systems, relying on discrete control theory techniques. It separates concerns, as illustrated in Figure 1(b), by supporting separate specification of, on one

hand, the possible behaviors of the components, and on the other hand,the control objectives for the components assembly. From these two specifications, DCS can automatically generate, if a solution exists, a correct control decision component.

Outline of the paper. Section 2 briefly recalls notions upon which we base our contribution (synchronous mode automata, DCS). Section 3 describes the BZR programming language, extending the previous ones with a notion of contract. Section 4 then formally describes the compilation of this language, where DCS operations are applied upon nodes with contracts. Section 5 outlines implementation and Section 6 discusses performances and scalability.

2 Context

2.1 Synchronous mode automata

We place our work in the framework of reactive systems and synchronous programming [5], and adopt its classical basic notions. More particularly, we will adopt notations inspired from the Lucid Synchrone language, concerning mixed dataflow and automata construct [9].

2.1.1 Behavior

For our examples, we consider programs expressed as synchronous Moore machines, with parallel and hierarchical composition. The states of such machines define data-flow equations, as in Lucid Synchrone or Lustre. 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 next step. Figure 2 illustrates an example. Ar the highest level, we have a three-state automaton, initially in state A, associated with equation $y = \texttt{false}$. Upon condition c, it takes the transition to state B, which is itself associated with the parallel composition of three sub-nodes: two sub-automata and one simple equation, defining y in terms of y_1 and y_2 defined in each of the sub-automata, by equations associated with the states, following the same principle as previously for state A. Upon $c \wedge d$, a transition is taken to state C, from where, upon d, a transition is taken to A.

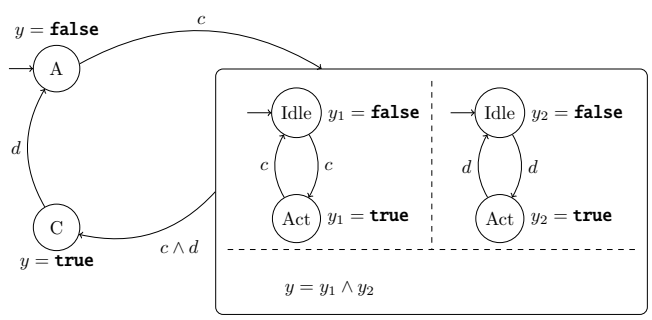

Figure 2: Mixed state / dataflow example

2.1.2 Structure

For scalability and abstraction purpose, synchronous programs are structured in *nodes*, with a name f, inputs $x_1, \ldots, x_n$, outputs $y_1, \ldots, y_p$ and declarations D. y_i variables are to be defined

in D, using operations between values of x_j and y_j. Figure 3 gives the graphical syntax of node definitions. The nodes are the abstraction level we will use in BZR to perform modular application of discrete controller synthesis.

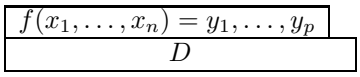

Figure 3: Node definition graphical syntax

The program of Figure 2 can then be structured as in Figure 4. The high-level automaton is specified in node defining f, with inputs c and d, and output y, and with state B is associated a an equation calling g. The latter is defined as a node with a body with three equations in parallel, two of which calling the node h. Finally, node f is defined with a body containing a two-state automaton, with associated equations in the states.

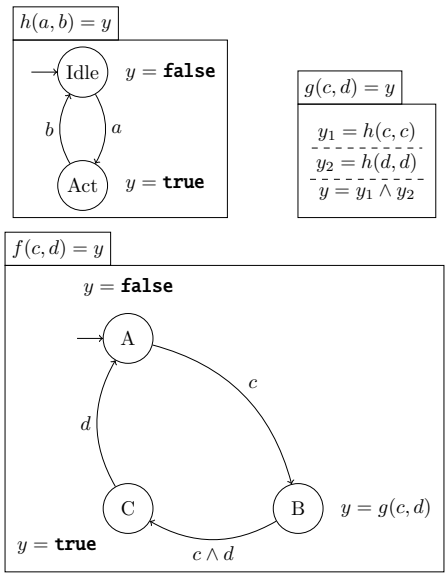

Figure 4: Structured program example

2.1.3 Corresponding transition system

Behavior of such programs can be represented by a transition system, as illustrated in Figure 5, 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 as in Figure 3, a transition function $Trans$ takes as inputs X as well as the current state value, and produces the next state value. The latter is memorized by $State$ for the next step. The output function Out takes the same inputs as T, and produces the outputs Y. We will use this representation to explain the notion of DCS, and to illustrate the behavior of our language.

2.2 Discrete Controller Synthesis

Discrete controller synthesis (DCS), emerged in the 80's [19, 7] allows to use constructive methods, that ensure, off-line, required properties on the system behavior. DCS is an operation that applies on a transition system (originally uncontrolled), where inputs X are partitioned into

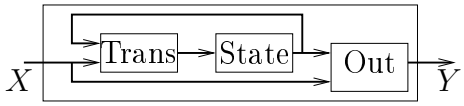

Figure 5: Transition system for a program

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 essentially invariance of a subset of the state space.

The purpose of DCS is to obtain a controller, which is a constraint on values of controllable variables X^c, 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 therefore *a priori* a relation; it can be transformed into a control function (up to the maximality). This is illustrated in Figure 6, where the transition system of Figure 5, as yet uncontrolled, is composed with the synthesized controller *Ctrlr*, which is fed with uncontrollable inputs X^u and the current state value from *State*, 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 performs a step.

Modular DCS. In the applications we are considering, transitions systems are the starting point to model fragments of a large scale system, which usually consists of several composed and nested subsystems. To avoid state space explosion induced by the concurrent nature of the systems, there has been a growing interest in designing algorithms that perform the controller synthesis phase by taking advantage of the structure of the system without expanding the system. For the compositional aspect one can consider the works of [10, 21, 2]. More related to our framework are the methods of [14, 1, 13, 17]. However, their models is different from ours (asynchronous vs synchronous) as well as the way the low-level controlled system is abstracted in order to compute the controller of a higher level.

2.3 Their combinations

There is previous work on the integration of DCS in a synchronous programming environment. Initially related to the Polychrony environment, SIGALI [16] is a tool that offers functionalities for verification of synchronous reactive systems and discrete controller synthesis. It manipulates Symbolic Transitions Systems (See Section 4.1), an equational and symbolic representation of automata. A wide variety of properties, such as invariance, reachability and attractivity, optimality w.r.t. to some quantitative criteria can be ensured by control. In the Polychrony environment, DCS is available as a formal tool amongst others, but not integrated at the programming language-level. A methodology for property-enforcing layers was proposed, [3] related to Mode Automata, but did not propose the language-level integration of objectives and DCS

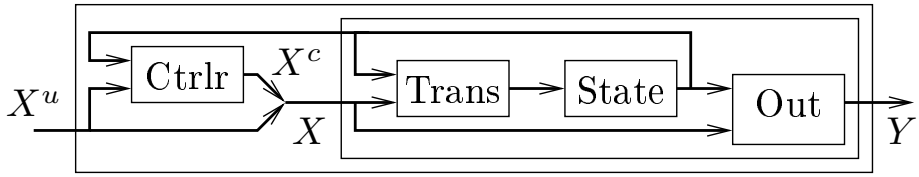

Figure 6: Controlled transition system

operations, as we do here. A deeper integration was proposed in a domain-specific language called NEMO [11], but that work remained at the front-end of synchronous compilation, whereas our work proposes full integration.

2.4 Contracts, validation and DCS

The notion of "design by contracts" have been introduced first in the Eiffel language [18]; 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 [15], 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 [6], where an algebra of contracts is defined. This latter approach allow to consider the relation between set of contracts defining one system, whereas our language only allow one contract to be associated to one node/implementation.

On the use of contracts for controller synthesis, [4] proposes a synthesis method based on the game theory, where contracts are used as assumptions to help the synthesis process. The modular aspect of the "design by contract" approach is not exploited.

Another related approach is interface synthesis [8]: the difference is that it is about constraining the environment of a component so that the component is used correctly, whereas in this work we constrain the component so that it works correctly whatever the environment does (within the assumptions) : the latter way is a more usual application of DCS. In particular, at the uppermost level, the assumption is taken as a hypothesis, to be checked by the programmer, in a way similar to the synchrony hypothesis, which has to be checked by the programmer on his system; on this basis the controller enforces the "guarantee" part. However, one can think of enforcing a guaranty condition at the top level but without any assumptions on the environment variables.

3 The BZR language

In this section we describe the original language construct which we add to the mode automata language introduced previously. We then show an example illustrating how it can be used to design controllers in a modular way.

3.1 Nodes with contracts

We define basic contract nodes, and then composite contract nodes, each time giving the corresponding DCS problem, which will be solved at compilation.

3.1.1 Basic BZR node, with a contract

Definition As illustrated in Figure 7, we associate to each node f a *contract*, which is a program associated 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. Its inputs are the inputs x_i and outputs y_i of the node f. At the node level, we assume the existence of a set $C = \{c_1, \ldots, c_n\}$ of controllable variables, that will be further used for ensuring this objective. This contract means that the node will be controlled, i.e., that values will be given to $c_1, \ldots, c_n$ such that, given any input trace yielding e_A *at each instant*, the output trace will yield the true value for e_G *at each instant*. This will be done by DCS. One can remark that the contract can

itself feature a program, e.g., automata, observing traces, and defining states (for example an error state where e_G is false, to be kept outside an invariant subspace). Also, one can define several such nodes with instances of the same body, that differ in assumptions and enforcements.

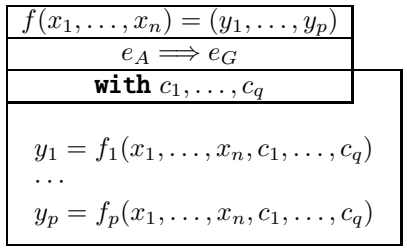

Figure 7: BZR node graphical syntax

Corresponding DCS problem For the compilation of such a BZR node, we will encode it into 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 sub-set of states where $e_A \Rightarrow e_G$ is true), with controllable variables C.

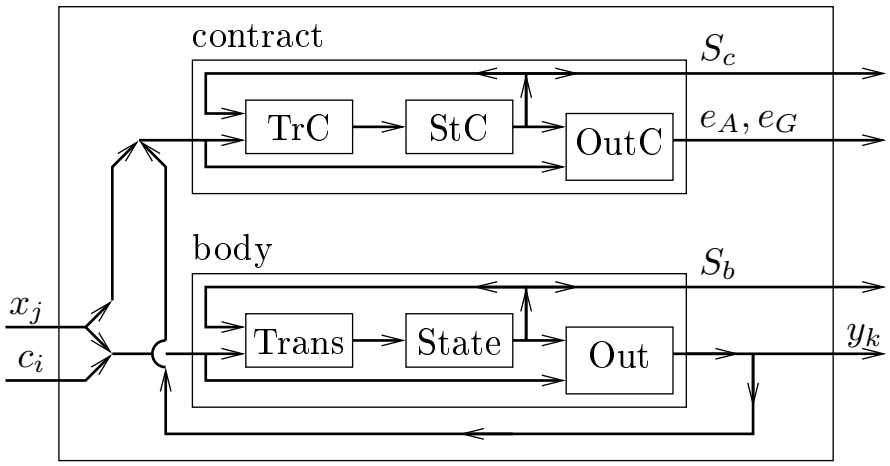

Figure 8: BZR node transition system

This is illustrated in Figure 8, re-using instances of the transition system of Figure 5: one for the contract (with transition function TrC, state memory StC and output function $OutC$) and one for the body of the node. The contract program has access to the node inputs $x_1, \ldots, x_n$ and outputs $y_1, \ldots, y_p$ of the body. Its outputs are e_A and e_G, and the variables $c_1, \ldots, c_q$ are inputs of the body. We show explicitly the states S_c of the contract and S_b of the body for explanation purposes.

Then in Figure 9, we see how the controller is synthesized and integrated in order to obtain the contract enforcing node, similarly to Figure 6. The global state comprises both body state and contract program state, and is taken as input by the controller, as well as the (uncontrollable) inputs $x_1, \ldots, x_n$ and contract Boolean outputs e_A and e_G.

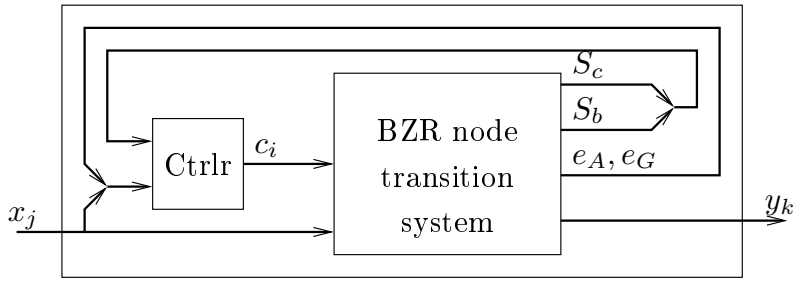

Figure 9: BZR node as DCS problem

3.1.2 BZR composite node

Definition A composite BZR node has a contract of itself, and sub-nodes which are also BZR nodes, with their own contracts, as illustrated in Figure 10. Sub-nodes may communicate. 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.

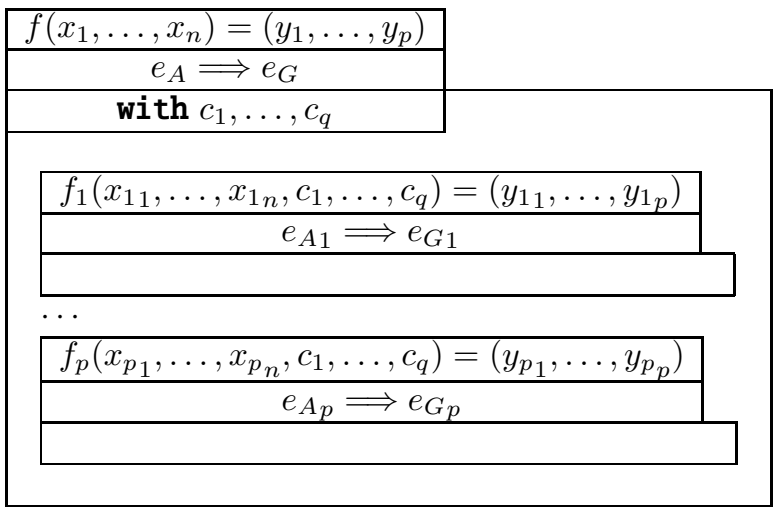

Figure 10: BZR composite node

The objective is still to control the body, with the use of 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, so that 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 problem becomes that: assuming e_A, $(e_{A1} \Rightarrow e_{G1})$ and $(e_{A2} \Rightarrow e_{G2})$, we want to enforce e_{A1}, e_{A2} and e_G. In particular, control at the level of the composite takes care of enforcing assumptions of the sub-nodes.

Corresponding DCS problem The control objective is to make invariant the subset of states where $e_A \Rightarrow e_G \wedge e_{A1} \wedge e_{A2}$ is true. This objective is applied on the global transition system composing the contract and the body of the node, as well as the contracts for each of the sub-nodes. Note that the bodies of the sub-nodes are not used for the controller computation.

Instead, we use the contracts as an abstraction of these sub-nodes. This is to partially avoid the classical state space explosion occurring when computing the whole system.

Figure 11 illustrates this, where only the state variables of contracts of sub-nodes are used by the controller of the upper-level node.

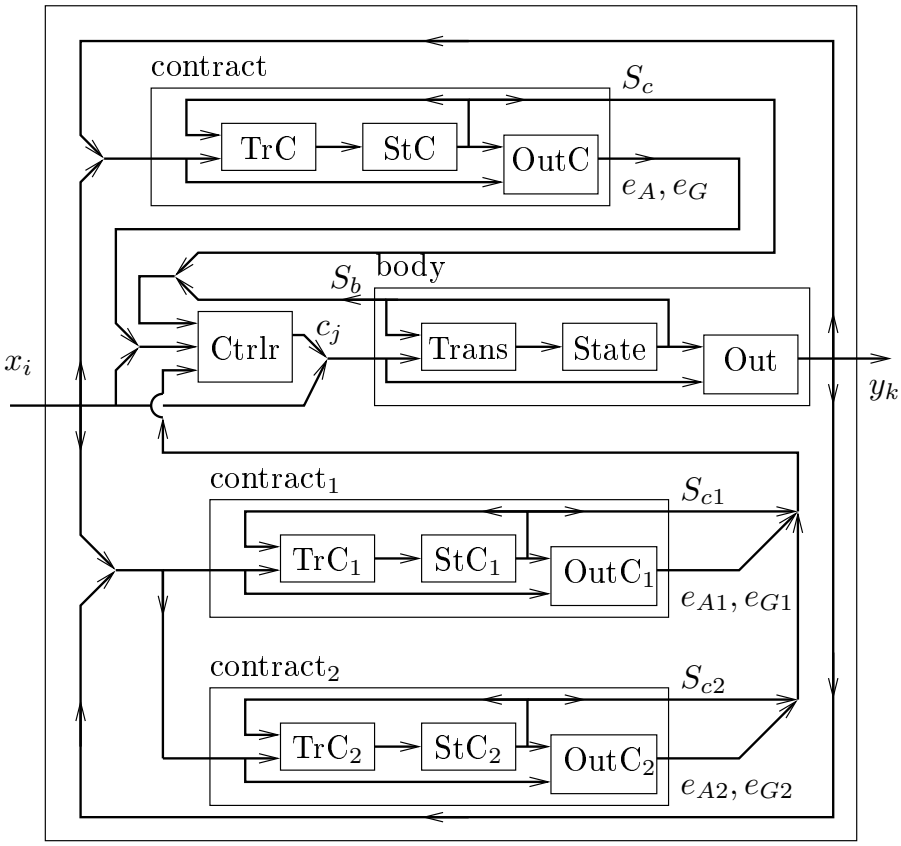

Figure 11: BZR composite DCS problem

3.2 Example: Multi-task System

We now illustrate the previous section through a simple example of a multi-task system. This example is shown for readability purpose with an ad-hoc graphical syntax; the whole final example in concrete textual syntax can be seen in appendix A.

We first specify the controller for one task, and then build a n-tasks server. We then illustrate how the composition of two n-tasks servers in order to build a $2n$-tasks server involves introducing controllability, without which solutions can not be found by DCS. For this, we voluntarily show pedagogic examples where the control can not be found satisfactorily, before fixing them into a fully working program.

3.2.1 Delayable Tasks

Figure 12 shows the control of a delayable task specified in mode automata. A delayable task takes three inputs `r`, `c` and `e`: `r` is the task launch request from the environment, `e` is the end

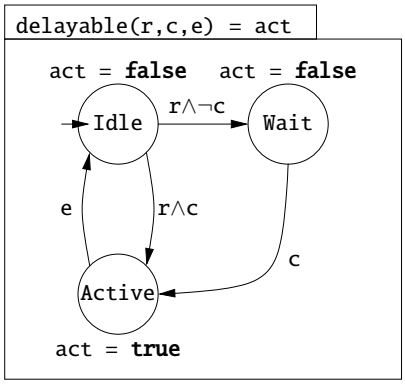

Figure 12: Delayable task (graphical syntax)

request, and `c` is meant to be a controllable input controlling whether, on request, the task is actually launched (and therefore goes in the active state), or delayed (and then forced by the controller to go in the waiting state by stating the false value to `c`). This node outputs a boolean `act` which is true when the task is active.

The Figure 13 shows a node `ntasks` where n delayable tasks have been put in parallel. The tasks are inlined i.e., their code is expanded at the location of the call, so as to be able to perform DCS on this node, taking into account the tasks' states. Until now, the only interest of modularity is, from the programmer's point of view, to be able to give once the delayable task code.

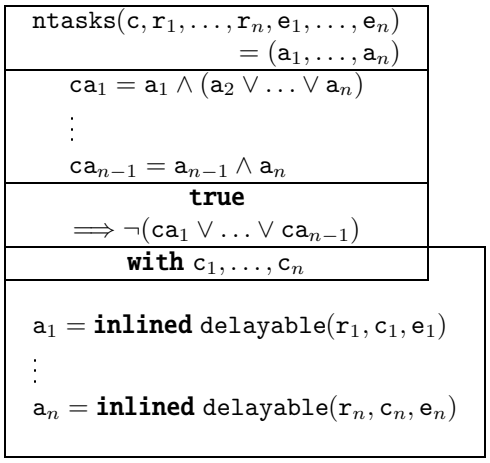

Figure 13: `ntasks` node: n delayable tasks in parallel

This `ntasks` node is provided with a contract, stating that its composing tasks are exclusive, i.e., that there are no two tasks in the active state at the same instant. This is encoded in the equations defining conflicts: ca_i is true if task i is active and some other task $j, i < j \leq n$ is active too. This contract is enforced with the help of the controllable inputs c_i. Typically, the

expected behavior of the obtained controller is to force to the false value the c_i variables, when the task i is requested while another task is in the active state.

3.2.2 Contract composition

We now want to reuse the **ntasks** node, in order to build modularly a system composed of $2n$ tasks. The Figure 14 shows the parallel composition of two **ntasks** nodes. We associate to this composition a new contract, which role is to enforce the exclusivity of the $2n$ tasks.

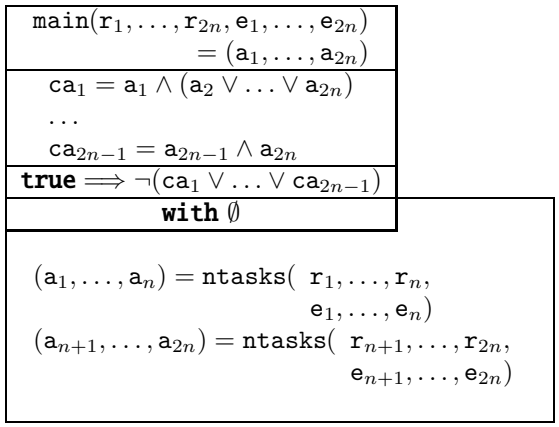

Figure 14: Composition of two **ntasks** nodes

It is easy to see that the contract of **ntasks** is not precise enough to be able to compose several of these nodes, because there is no way to control them to avoid that one task is active in each of the two subsystems. Therefore, we need to refine this contract by adding some way to externally control the activity of the tasks.

3.2.3 Contract refinement

We first add an input **c**, meant to be controllable at an upper level. The refined contract will enforce that:

1. the tasks are exclusive, i.e., that there are no two tasks in the active state at the same instant;
2. one task is active only at instants when the input **c** is true. This property, appearing in the contract, allows a node instantiating **ntasks** to forbid any activity of the n tasks instantiated.

The Figure 15 contains this new **ntasks** node.

However, the controllability introduced here is now too strong. The synthesis will succeed, but the computed controller, without knowing how **c** will be instantiated, will actually block all tasks in their idle state. Indeed, if the controller allows one task to go in its active state, the input **c** (uncontrollable at the **ntasks** level) can become false at the next instant, violating the property to enforce.

$$\mathtt{ntasks}(\mathtt{c}, \mathtt{r}_1, \ldots, \mathtt{r}_n, \mathtt{e}_1, \ldots, \mathtt{e}_n) = (\mathtt{a}_1, \ldots, \mathtt{a}_n)$$

$$\begin{array}{l}
\mathtt{ca}_1 = \mathtt{a}_1 \wedge (\mathtt{a}_2 \vee \ldots \vee \mathtt{a}_n) \\
\ldots \\
\mathtt{ca}_{n-1} = \mathtt{a}_{n-1} \wedge \mathtt{a}_n \\
\mathtt{one} = \mathtt{a}_1 \vee \ldots \vee \mathtt{a}_n
\end{array}$$

$$\begin{array}{c}
\textbf{true} \\
\Longrightarrow \neg(\mathtt{ca}_1 \vee \ldots \vee \mathtt{ca}_{n-1}) \wedge (\mathtt{c} \vee \neg\mathtt{one})
\end{array}$$

$$\textbf{with}\ \mathtt{c}_1, \ldots, \mathtt{c}_n$$

$$\begin{array}{l}
\mathtt{a}_1 = \textbf{inlined}\ \mathtt{delayable}(\mathtt{r}_1, \mathtt{c}_1, \mathtt{e}_1) \\
\vdots \\
\mathtt{a}_n = \textbf{inlined}\ \mathtt{delayable}(\mathtt{r}_n, \mathtt{c}_n, \mathtt{e}_n)
\end{array}$$

Figure 15: First contract refinement for the **ntasks** node

Thus, we propose to add an assumption to this contract: the input **c** will not become false if a task was active an instant before. This assumption will be enforced correct by the controller of the upper level. This new contract is visible in Figure 16.

$$\mathtt{ntasks}(\mathtt{c}, \mathtt{r}_1, \ldots, \mathtt{r}_n, \mathtt{e}_1, \ldots, \mathtt{e}_n) = (\mathtt{a}_1, \ldots, \mathtt{a}_n)$$

$$\begin{array}{l}
\mathtt{ca}_1 = \mathtt{a}_1 \wedge (\mathtt{a}_2 \vee \ldots \vee \mathtt{a}_n) \\
\ldots \\
\mathtt{ca}_{n-1} = \mathtt{a}_{n-1} \wedge \mathtt{a}_n \\
\mathtt{one} = \mathtt{a}_1 \vee \ldots \vee \mathtt{a}_n \\
\mathtt{pone} = \textbf{false}\ \textbf{fby}\ \mathtt{one}
\end{array}$$

$$\begin{array}{c}
(\neg\mathtt{pone} \vee \mathtt{c}) \\
\Longrightarrow \neg(\mathtt{ca}_1 \vee \ldots \vee \mathtt{ca}_{n-1}) \wedge (\mathtt{c} \vee \neg\mathtt{one})
\end{array}$$

$$\textbf{with}\ \mathtt{c}_1, \ldots, \mathtt{c}_n$$

$$\begin{array}{l}
\mathtt{a}_1 = \textbf{inlined}\ \mathtt{delayable}(\mathtt{r}_1, \mathtt{c}_1, \mathtt{e}_1) \\
\ldots \\
\mathtt{a}_n = \textbf{inlined}\ \mathtt{delayable}(\mathtt{r}_n, \mathtt{c}_n, \mathtt{e}_n)
\end{array}$$

Figure 16: Second contract refinement for the **ntasks** node

We can then use this new **ntasks** version for the parallel composition, by instantiating the **c** input by two controllable variables. This composition can be found in Figure 17.

4 BZR compilation

This section describes in a formal way the compilation process of our language. This compilation process is modular, meaning that each node will be compiled in an independent way; and it

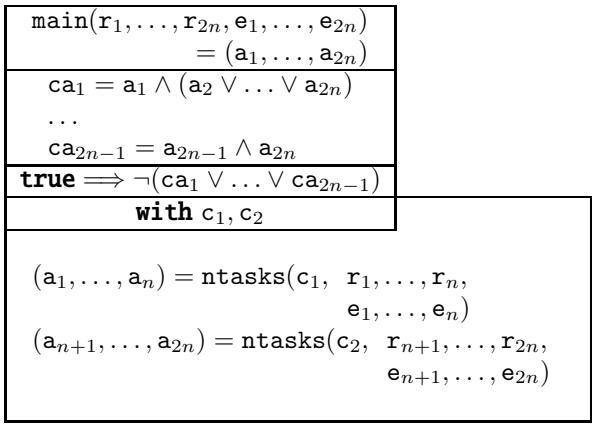

Figure 17: Two refined **ntasks** parallel composition

comprises a discrete controller synthesis stage for each of these nodes. We first recall the notations used, then present the controller synthesis for one contract, and last show how the synthesized controllers are recombined to obtain the whole controlled system.

4.1 Control of Symbolic Transition Systems (STS)

4.1.1 Definitions

Notations. Given a set of boolean variables $Z = Z_1, \cdots, Z_k$, we define a valuation of Z as a function $val : Z \to \mathbb{B}^k$ that assigns to each variables in Z a value either true or false. In the sequel, we shall use X, Y, Z as vectors of boolean variables and x, y, z for a possible valuation of these vectors. Given a predicate $B \in \mathbb{B}[Z]$ a polynomial over Z and $z \in val(Z)$, we denote by $B(z) \in \mathbb{B}$ the predicate B valuated by z. We further denote $\overline{z} = z_0.z_1.z_2 \cdots$ an infinite sequence of valuations of Z. Given a sequence $\overline{z}$ and a predicate $G \in \mathbb{B}[Z]$, we denote $\overline{z} \models \Box G$ the fact that G hold for the sequence $\overline{z}$ at every instant.

$$\overline{z} \models \Box G \text{ iff } \overline{z} = z_0.z_1.z_2 \cdots \text{ and } \forall i, G(z_i).$$

Symbolic Transition Systems. We represent synchronous programs by Synchronous Symbolic Transition Systems (STS). A STS $S(X, Y, Z)$, defining a synchronous program of state variables $X \in \mathbb{B}^m$, input event variables $Y \in \mathbb{B}^n$, output event variables $Z \in \mathbb{B}^p$, is a tuple (P, O, Q, Q_0):

$$S = \begin{cases} X' = P(X, Y) \\ Z = O(X, Y) \\ Q(X, Y) \\ Q_0(X) \end{cases}$$

where:

- the vectors X and X' respectively encode the current and next states of the system and are called *state variables* (the states of the system contain the memory necessary for describing the system behavior).

- $P \in \mathbb{B}[X, Y]$ represent the transition function. It can be considered as a vector-valued function $[P_1, \ldots, P_n]$ from $\mathbb{B}^{n+m}$ to $\mathbb{B}^n$. Each polynomial component P_i represents the evolution of the state variable X_i. It characterizes the dynamic of the system. between the current state X and events Y and next state X'.
- $O \in \mathbb{B}[X, Y]$ represents the output function.
- $Q_0 \in \mathbb{B}[X]$ defines the set initial states, and $Q \in \mathbb{B}[X, Y, Z]$ the constraints between current states and events and encodes the *static* part of the system (invariant for all instants t).

The semantics of a STS S is that it defines a set of sequences $\overline{(x, y, z)}$ such that : $Q_0(x_0)$ and $\forall i$,

$$Q(x_i, x_i) \wedge \Big(x_{i+1} = P(x_i, x_i)\Big) \wedge \Big(z_i = O(x_i, y_i)\Big).$$

We denote by Traces(S) this set of sequences.

STS transformations. Given two STS S_1 and S_2, we note by $S_1 \| S_2$, the synchronous parallel composition of S_1 and S_2 which consists in performing the conjunction of the composing predicate of S_1 and S_2, and is defined iff 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, $S_1 \| S_2$ is a STS $S_1 \| S_2(X_1 \cup X_2, (Y_1 \cup Y_2) \setminus (Z_1 \cup Z_2), Z_1 \cup Z_2)$:

$$S_1 \| S_2 = \begin{cases} X_1', X_2' = P_1(X_1, Y_1) \wedge P_2(X_2, Y_2) \\ Z_1, Z_2 = (O_1(X_1, Y_1), O_2(X_2, Y_2)) \\ Q_1(X_1, Y_1) \wedge Q_2(X_2, Y_2) \\ Q_{01}(X_1) \wedge Q_{02}(X_2) \end{cases}$$

Finally, we denote by $S \rhd A$ the extension of constraints of S with the predicate $A \in \mathbb{B}[X, Y, Z]$:

$$S \rhd A = \begin{cases} X' = P(X, Y) \\ Z = O(X, Y) \\ Q(X, Y, Z) \wedge A(X, Y, Z) \\ Q_0(X) \end{cases}$$

Contracts satisfaction. In the sequel, we shall consider properties expressed by means of contracts that are defined as follows:

Definition 1 (Contract)**.** *A contract is a tuple $C = (S^c, A, G)$ where $S^c(X^c, (Y \cup Z), \emptyset)$ is a STS, $A \in \mathbb{B}[X^c]$ and $G \in \mathbb{B}[X^c]$ are predicates.*

Intuitively, S^c can be seen as an abstraction of a component program, G is the property to be satisfied by the traces of the component on which this contract is placed providing the fact that the model of the environment A is satisfied. For the sake of brevity and clarity, we define the contract predicates A and G on only state variables of the contract. We remark though that this does not restrict the expressiveness of these properties, as one can add dummy state variables constrained with inputs or outputs values, so as to be able to express properties upon inputs and outputs variables.

Definition 2 (Contract fulfilment)**.** *A STS $S(X, Y)$ fulfills a contract $C = (S^c, A, G)$, noted $S \models C$, if $\forall \overline{(x, x^c, y, z)} \in$ Traces$(S \| S^c)$, $\overline{(x^c)} \models \Box A \Rightarrow \overline{(x^c)} \models \Box G$.*

Hence, a contract is satisfied whenever the traces of S, composed with S^c and satisfying $\Box A$, satisfy $\Box G$. This constitutes one of the main difference with the work of [6], as in their framework, A and G are assertions of any kind on traces. Our restriction is due to the technique used: only safety properties (vs liveness or equity) can be ensured by abstracted state space exploration, and preserved by synchronous composition.

As the above definition does not allow to be easily applied on STS, we give below a property on contracts: the environment model can be viewed as additional constraints of the STS composed of S and the contract program S^c.

Proposition 1. $(S \| S^c) \rhd A \models G \Rightarrow S \models (S^c, A, G)$.

4.1.2 Contracts enforcement

Assume given a system S and a contract C on S. Our aim is to restrict the behavior of S in order to fulfil the contract.

The control of a STS relies on a distinction between events. We distinguish between the *uncontrollable* event variables Y^{uc} which are defined by the environment, and the *controllable* event Y^c which are defined by the controller of the system (they are considered as internal variables). Now, in order to enforce $C = (S^c, A, G)$ with $S^c(X^c, Y^{uc} \cup Z, \emptyset)$, $A \in \mathbb{B}[X^c]$ and $G \in \mathbb{B}[X^c]$ on S we consider the STS $(S \| S^c) \rhd A$:

$$(S \| S^c) \rhd A = \begin{cases} X', X'^c = P(X, X^c, Y^c, Y^{uc}) \\ Z = O(X, X^c, Y^c, Y^{uc}) \\ Q(X, X^c, Y^c, Y^{uc}) \\ Q_0(X, X^c) \end{cases}$$

The property we wish to enforce by control on this system is given by the invariant G. In our framework, a controller is a predicate $K \in \mathbb{B}[X, X^c, Y^c, Y^{uc}]$ that constraints the set of admissible event so that the traces of the controlled system always satisfy the predicate G. We do not detail here how such a controller can be computed. It relies on a fix-point computation w.r.t. the function $Pre_{uc}(E) = \{(x, x^c) \mid \forall y^{uc}, \exists y^c, Q(x, y^{uc}, y^c) \Rightarrow P(x, y^{uc}, y^c) \in E\}$. We will present a more generic algorithm in the next section. The controller describes how to choose the static controls; when the controlled system is in state (x, x^c), and when an event y^{uc} occurs, any value y^c such that $Q(x, x^c, y^{uc}, y^c)$ and $K(x, x^c, y^{uc}, y^c)$ can be chosen. The behavior of the system supervised by the controller is then modeled by $(S \| S^c) \rhd K$.

Determination of the controller. Assume now that we have computed a controller $K \in \mathbb{B}[X, X^c, Y^c, Y^{uc}]$. 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 performed a random choices amongst them. Following a method similar to the one described in [12], another possibility is to derive from the controller a set of functions F_i^c that depends on X, X^c, Y^{uc} and some fresh *phantom* variables ϕ_i, one for each controllable variables, namely:

$$K(X, X^c, Y^c, Y^{uc}) \quad \Leftrightarrow \quad \exists(\phi_i)_{i\leq\ell} \begin{cases} Y_1^c = F_1^c(X, X^c, Y^{uc}, \phi_1) \\ \cdots \\ Y_i^c = F_i^c(X, X^c, Y^{uc}, Y_1^c, \cdots, Y_{i-1}^c, \phi_i) \\ \cdots \\ Y_\ell^c = F_n^c(X, X^c, Y^{uc}, Y_1^c, \cdots, Y_{\ell-1}^c, \phi_\ell) \\ K'(X, X^c, Y^{uc}) \end{cases}$$

In other words, whatever the valuation of a tuple (x, x^c, y^{uc}, y^c), there exists a valuation $(v_{\phi_i})_{i\leq\ell}$ of $(\phi_i)_{i\leq\ell}$ such that $y_i^c = F_i^c(x, x^c, y^{uc}, v_{\phi_i})$.

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.

Remark 1. *With the determination of the controller, part of the solutions can be actually lost, when the synthesized controller is not deterministic (but the safety property is kept); and in this sense the modular control is sub-optimal. It is an interesting perspective indeed to replace local random selections with a heuristic taking into account some of the interactions between components. Note however, that it is possible to keep the maximal behavior by keeping the new phantom variables.*

4.2 Modular control of STS

4.2.1 Contracts composition

Let consider now a hierarchically designed program, i.e., a STS $S(X, Y, Z)$ composed of sub-components $S_1, \ldots, S_n$, together with additional local code S' (as in Figure 18). We have then :

$$S(X, Y, Z) = (S' \| S_1 \| \ldots \| S_n)$$

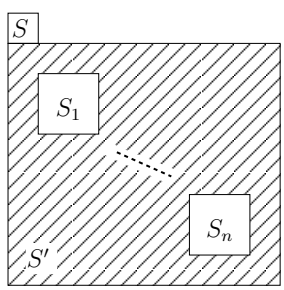

Figure 18: STS composed of several subcomponents

Note that $Y_i^{uc} \subseteq X \cup Y^{uc} \cup Y^c \cup Z$, 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-component $S_i(X_i, Y_i, Z_i)$ comes with a contract $C_i = (S_i^c, A_i, G_i)$, with $S_i^c(X_i^c, Y_i \cup Z_i, \emptyset)$, $A_i \in \mathbb{B}[X_i^c]$, $G_i \in \mathbb{B}[X_i^c]$, and that a controller K_i has been computed such as, for all i, $(S_i \| S_i^c) \rhd K_i \models C_i$.

We want now to obtain a controller K for the system S to fulfill a contract $C = (S^c, A, G)$, with $S^c(X^c, Y \cup Z, \emptyset)$, $A \in \mathbb{B}[X^c]$ and $G \in \mathbb{B}[X^c]$. 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 $\hat{S}$, defined as the composition of S' with the system part of the subcontracts, constrained with the properties enforced by K_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. Moreover, the Z_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 Y_1^{uc} and internal variables. Hence there exists causality problems between these variables and the variables of the upper level. This causality constraint will be resolved by the design of a more general controller synthesis algorithm, on which we will come back in Section 4.2.2. We define the new system to be controlled as follows:

$$\hat{S}(X, Y^{uc} \cup Z_1 \cup \ldots \cup Z_n, Y^c, Z) \;=\; \Big(S' \| \big(S_1^c \rhd (A_1 \;\Rightarrow\; G_1)\big) \| \ldots \| \big(S_n^c \rhd (A_n \;\Rightarrow\; G_n)\big)\Big)$$

We should notice that, in order to the STSs 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 $\hat{C}$, which will be used to compute a controller on $\hat{S}$:

$$\hat{C} = (S^c, \hat{A}, \hat{G}) \text{ where } \begin{cases} \hat{A} = A \\ \hat{G} = G \wedge A_1 \wedge \ldots \wedge A_n \end{cases}$$

We compute now the controller K, enforcing the contract C on the STS $\hat{S}$:

$$\hat{S} \rhd K \models \hat{C}$$

The correction of this method is given by the following result, stating that the controller K, computed for the abstracted system $\hat{S}$, is a correct controller for the concrete system, i.e., S controlled by K fulfils the contract C.

Theorem 1.
If $\hat{S} \rhd K \models \hat{C}$ and for all i, $(S_i \| S_i^c) \rhd K_i \models C_i$, then:

$$\Big(S' \| (S_1 \| S_1^c) \rhd K_1 \| \ldots \| (S_n \| S_n^c) \rhd K_n\Big) \rhd K \models C$$

4.2.2 Control of an STS with sub-contracts

As mentioned in the previous section, there exists causality dependencies between the Z_i variables and the Y_i^c variables, in the sense that the computation of the value of the variables Z_i depends on the value of Y_i^c, which must then be computed before. Moreover, the Y_i^c can be computed according to the value of some other variables of the upper-level system. In the following, we denote by $X_1 \prec X_2$ the fact that X_2 depends on X_1. This relation is straightforwardly extended to variable sets:

$$\tilde{X} \prec \tilde{Y} \stackrel{def}{=} \forall (X, Y) \in \tilde{X} \times \tilde{Y}, X \prec Y$$

In the sequel, we shall consider the following subsets of Z_i : $(\tilde{Z}_i)_{i \leq n}$, where $\forall i \leq n, \tilde{Z}_i \subseteq \tilde{Z}_{i-1}$, and such that $Y_i^c \prec \tilde{Z}_i$. Furthermore, we note $\tilde{Z}'_i = \tilde{Z}_i \setminus \tilde{Z}_{i-1}$.

$$Y \prec \tilde{Z}_0 \prec \{Y_1^c\} \prec \tilde{Z}_1 \prec \cdots \prec \{Y_n^c\} \prec \tilde{Z}_n \tag{1}$$

Control synthesis algorithm overview. Computing the controller K as in Section 4.1.2 is not suitable as it does not take into account the dependencies between the variables. Intuitively, in order to be able to compute the value of Y_i^c we need to know the value of the variables $\tilde{Z}_i$. Thus, if we want to be sure that there exists a solution to the determination, the controller should take into account that the value of the variable Y_i^c is correct whatever the value of $\tilde{Z}_i$. This is what we capture by computing the Pre_{uc} operator in the following way:

$$Pre_{uc}(E) = \Big\{(x, x^c) \mid \forall y^{uc}, \tilde{z}_0, \exists y_1^c, \forall \tilde{z}_1', \ldots, \exists y_n^c, \forall \tilde{z}_n', \\ Q(x, y^{uc}, y^c, z) \Rightarrow P(x, y^{uc}, y^c, z) \in E\Big\}$$

Further the computation of K is similar to the one of Section 4.1.2.

Deterministic controller. Concerning the determination of K, we also need to take into account the dependencies between the variables. Indeed, the order relation (1) implies that the deterministic controller, as defined in Section 4.1.2, will be a set of functions $\tilde{F}_i^c$ that depends on X, X^c, Y^{uc}, ϕ_i and the additional uncontrollable inputs $\tilde{Z}_{i-1}$:

$$K(X, X^c, Y^c, Y^{uc}) \\ \Leftrightarrow \\ \exists(\phi_1, \cdots, \phi_l), \begin{cases} Y_1^c = \tilde{F}_1^c(X, X^c, Y^{uc}, \tilde{Z}_0, \phi_1) \\ \cdots \\ Y_i^c = \tilde{F}_i^c(X, X^c, Y^{uc}, Y_1^c, \cdots, Y_{i-1}^c, \tilde{Z}_{i-1}, \phi_i) \\ \cdots \\ Y_n^c = \tilde{F}_n^c(X, X^c, Y^{uc}, Y_1^c, \cdots, Y_{n-1}^c, \tilde{Z}_{n-1}, \phi_l) \end{cases}$$

These $\tilde{F}_i^c$ functions are related with F_i^c, defined in Section 4.1.2, by $\tilde{F}_i^c = \forall \tilde{Z}_i, F_i^c$. The existence of $\tilde{F}_i^c$ is then ensured by the modified version of the control synthesis algorithm we give further. Once again, we can further eliminate the variables (ϕ_i) by choosing a particular value for each of them.

5 Implementation

Figure 19 illustrates the compilation process for BZR implementing our method. Boxes with round corners indicate data (source code, target code, intermediate formats); rectangular boxes indicate tools and operations. It is built around synchronous compilation and DCS technology, and borrows essentially two pre-existing tools (indicated in stripped boxes). One is a synchronous compiler: HEPTAGON, used in order to (i): compile the nodes into a format accepted by the DCS tool, and (ii): compile the composition of the triangularized controller with the originally uncontrolled automaton, hence building the controlled automaton, and generating executable code for it. The other is a DCS tool: SIGALI.

The BZR compiler has also been used for a case study of a video display scenario on a mobile phone.

6 Performances

The bottleneck of our approach is clearly the synthesis time, as the algorithm is based on the exploration of the state space of each node. This section thus shows some typical synthesis time on multitasks systems, with and without use of modularity.

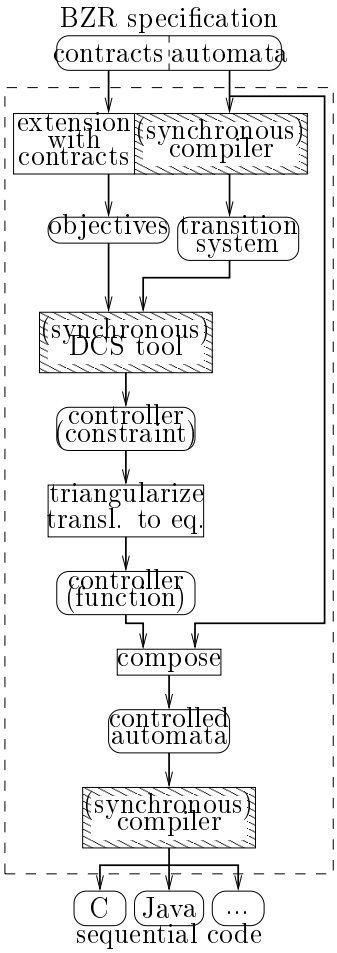

Figure 19: BZR compilation process.

The system considered here is composed of $3n$ tasks, to be enforced exclusive, as showed in Section 3.2. These $3n$ tasks are composed of n delayable tasks (Figure 12) and $2n$ rejectable tasks (Figure 20: tasks requests can be rejected but are not memorized). Three experiments are fulfilled:

1. the $3n$ tasks are inlined into the same node: the state space to explore corresponds then to the entire state space of the 3n automata composition;
2. the $3n$ tasks are decomposed into three nodes instantiation of n inlined tasks, as in Section 3.2: three separate synthesis are then performed (one for n delayable tasks, one for n rejectable tasks and one for the main node);
3. likewise, the $3n$ tasks are decomposed into three nodes, but we also use the modularity to perform the three synthesis in parallel: this is possible, since they do not depend on each other's results.

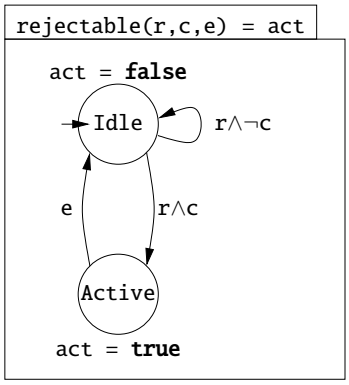

Figure 20: Rejectable task

Figure 21 shows the compared synthesis time for each of these experiments. They have been fulfilled on a standard PC with two 2.33 GHz cores, and 4 Gb of RAM. The non-monotony of the two curves are explained by the sensitivity of the underlying algorithms, which makes the synthesis time hard to predict. Nevertheless, these measurements shows that modularity can improve the usability of discrete controller synthesis. We can see, e.g., that while synthesis fail (for lack of memory) in "inlined" mode for $n = 14, 15$ and $n > 16$, such systems when adequately structured can be handled in few seconds. Moreover, modularity allows to gain some synthesis time by parallelizing the synthesis computations.

7 Conclusion

We have proposed a programming-level method, based on the "design by contracts" principle, to apply modular discrete controller synthesis, integrated into a compilation process. This method allows to apply DCS on subcomponents of a system, in order to compute one single controller for each of these subcomponents. These controllers can then be composed with their associated components, before their composition in an upper-level component. The contracts can then be used to compute a controller for this upper-level component, abstracting the bodies of its subcomponents; and so on. On the other hand, in the comparison between modular and monolithic

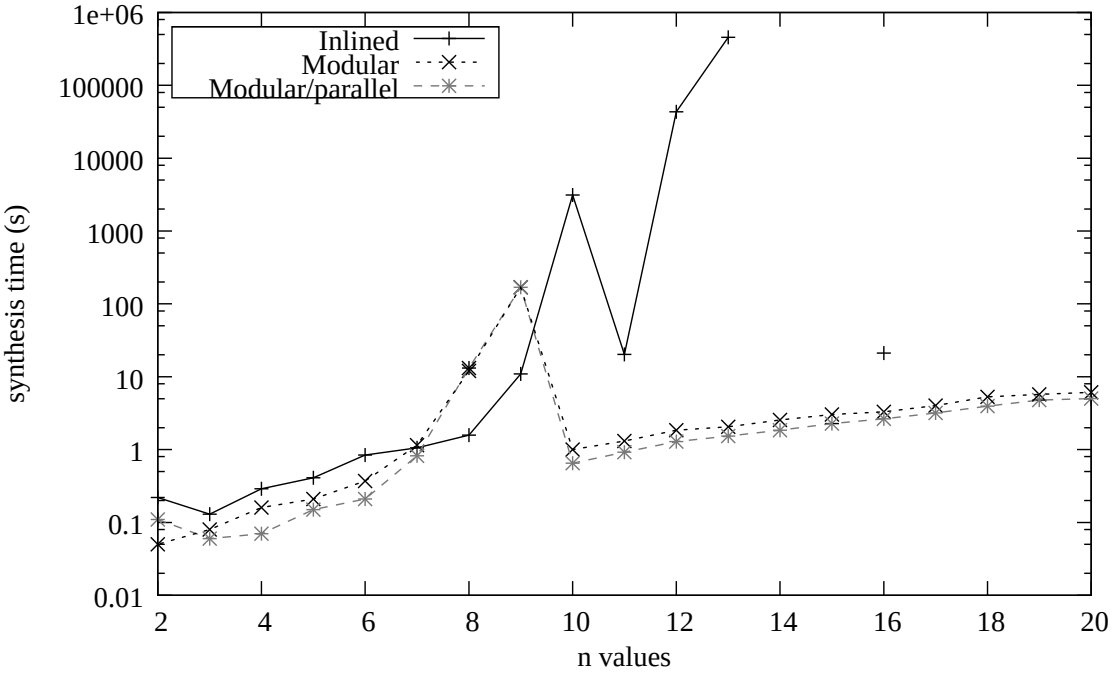

Figure 21: Compared synthesis time for $3n$ automata in inlined, modular and modular+parallel modes

synthesis, we have the advantage of breaking down the cost of synthesis computations (as we only keep the guaranty at the upper level and not the current node that can be itself a composition of several nodes), for which real-size evaluations are in our projects. We have also proposed a new nondeterministic synchronous language with contracts, named BZR, as implementation of this approach. We have shown how this language can be compiled towards symbolic transition systems, and how modular controllers are computed and recomposed.

Further work can be fulfilled toward several directions. This method could be applied in different component framework, so as to explore its interaction with actual industrial design flow. Some work is in progress within the Fractal framework. Such integration in actual design flow entails a greater interactivity with the programmer: thus, efforts on diagnosis should be made. Some methods should be proposed to get intelligible informations to the programmer when the synthesis fail, which is not currently possible. Another interesting prospect would also be, when synthesis fail, to use the synthesis algorithm to infer some additional constraints on the contracts of the program to in order to allow the synthesis to succeed.

In a more technical concern, some other controller synthesis methods or algorithms could replace the SIGALI tool. This work only address invariance objectives: other kind of synthesis objective (accessibility, attractivity) would jeopardize the modularity properties. An interesting prospect would then to deal with such synthesis objectives.

References

[1] S. Abdelwahed and W. Wonham. Supervisory control of interacting discrete event systems. In *41th IEEE Conference on Decision and Control*, pages 1175–1180, Las Vegas, USA,

December 2002.

[2] K. Akesson, H. Flordal, and M. Fabian. Exploiting modularity for synthesis and verification of supervisors. In *Proc. of the IFAC*, 2002.

[3] K. Altisen, A. Clodic, F. Maraninchi, and E. Rutten. Using controller synthesis to build property-enforcing layers. In *Proceedings of the European Symposium on Programming (ESOP'03)*, number 2618 in LNCS, Warsaw, Poland, April 2003.

[4] Ralph-Johan Back and Cristina Cerschi Seceleanu. Contracts and games in controller synthesis for discrete systems. *Engineering of Computer-Based Systems, IEEE International Conference on the*, 0:307, 2004.

[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, Special issue on embedded systems*, 91(1):64–83, January 2003.

[6] Albert Benveniste, Benoit Caillaud, and Roberto Passerone. A generic model of contracts for embedded systems. Research Report RR-6214, INRIA, 2007.

[7] C. Cassandras and S. Lafortune. *Introduction to Discrete Event Systems*. Kluwer Academic Publishers, 1999.

[8] A. Chakrabarti, L. de Alfaro, T.A. Henzinger, and F.Y.C. Mang. Synchronous and bidirectional component interfaces. In *CAV 2002: 14th International Conference on Computer Aided Verification, LNCS, Springer Verlag*, 2002.

[9] Jean-Louis Colaço, Bruno Pagano, and Marc Pouzet. A Conservative Extension of Synchronous Data-flow with State Machines. In *ACM International Conference on Embedded Software (EMSOFT'05)*, Jersey city, New Jersey, USA, September 2005.

[10] M. H. De Queiroz and J. Cury. Modular control of composed systems. In *Proceedings of the American Control Conference*, pages 4051–4055, Chicago, Illinois, June 2000.

[11] G. Delaval and E. Rutten. A domain-specific language for multi-task systems, applying discrete controller synthesis. *J. on Embedded Systems*, 2007(84192):17, January 2007. `www.hindawi.com/journals/es`.

[12] Yann Hietter, Jean-Marc Roussel, and Jean-Jacques Lesage. Algebraic Synthesis of Transition Conditions of a State Model. In *Proceedings of 9th International Workshop On Discrete Event Systems (WODES'08)*, pages pp. 187–192, Göteborg, 2008.

[13] R.J. Leduc, W.M. Wonham, and M. Lawford. Hierarchical interface-based supervisory control: Parallel case. In *Proc. of the 39th Allerton Conf. on Comm., Contr., and Comp.*, pages 386–395, October 2001.

[14] C. Ma and W.M. Wonham. A symbolic approach to the supervision of state tree structures. In *13th Mediterranean Conference on Control and Automation*, Limassol, Cyprus, June 2005.

[15] 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)*, Rennes, France, September 2004.

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

[17] H. Marchand and B. Gaudin. Supervisory control problems of hierarchical finite state machines. In *41th IEEE Conference on Decision and Control*, Las Vegas, USA, December 2002.

[18] B. Meyer. Applying "design by contract". *Computer*, 25(10):40–51, Oct 1992.

[19] P. J. Ramadge and W. M. Wonham. The control of discrete event systems. *Proceedings of the IEEE; Special issue on Dynamics of Discrete Event Systems*, 77(1):81–98, 1989.

[20] Yin Wang, Stéphane Lafortune, Terence kelly, Manjunath Kudlur, and Scott Mahlke. The Theory of Deadlock Avoidance via Discrete Control. In *ACM Symposium on Principles of Programming Languages (POPL '09)*, Savannah, Georgia, USA, January 2009.

[21] Y. Willner and M. Heymann. Supervisory control of concurrent discrete-event systems. *International Journal of Control*, 54(5):1143–1169, 1991.

A BZR Example in Concrete Textual Syntax

```
node delayable(r,c,e:bool)
     returns (act:bool)
let
  automaton
    state Idle
      do act = false
      until r & c then Active
          | r & not c then Wait
    state Wait
      do act = false
      until c then Active
    state Active
      do act = true
      until e then Idle
  end
tel

node ntasks(c,r_1,...,r_n,e_1,...,e_n)
    returns (a_1,...,a_n:bool)
  contract
  let
    ca_1 = a_1 & (a_2 or ... or a_n);
    ⋮
    ca_{n-1} = a_{n-1} & a_n;
    one = a_1 or ... or a_n;
    pone = false fby one;
  tel
  assume (not pone or c)
```

```
  enforce not (ca_1 or ... or ca_{n-1}) & (c or not one)
     with (c_1,...,c_n:bool)
let
  a_1 = inlined delayable(r_1,c_1,e_1);
  ⋮
  a_n = inlined delayable(r_n,c_n,e_n);
tel

node main(r_1,...,r_{2n},e_1,...,e_{2n})
    returns (a_1,...,a_{2n}:bool)
  contract
  let
       ca_1 = a_1 & (a_2 or ... or a_{2n});
       ⋮
       ca_{2n-1} = a_{2n-1} & a_{2n};
  tel
  assume true
  enforce not (ca_1 or ... or ca_{2n-1})
  with (c_1,c_2:bool)
let
  (a_1,...,a_n) = ntasks(c_1,r_1,...,r_n,e_1,...,e_n);
  (a_{n+1},...,a_{2n}) = ntasks(c_2,r_{n+1},...,r_{2n},e_{n+1},...,e_{2n});
tel
```

Centre de recherche INRIA Rennes – Bretagne Atlantique
IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)

Centre de recherche INRIA Bordeaux – Sud Ouest : Domaine Universitaire - 351, cours de la Libération - 33405 Talence Cedex
Centre de recherche INRIA Grenoble – Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier
Centre de recherche INRIA Lille – Nord Europe : Parc Scientifique de la Haute Borne - 40, avenue Halley - 59650 Villeneuve d'Ascq
Centre de recherche INRIA Nancy – Grand Est : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex
Centre de recherche INRIA Paris – Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex
Centre de recherche INRIA Saclay – Île-de-France : Parc Orsay Université - ZAC des Vignes : 4, rue Jacques Monod - 91893 Orsay Cedex
Centre de recherche INRIA Sophia Antipolis – Méditerranée : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399