HAL
archives-ouvertes.fr

Reach Scheduling for Embedded Systems

Gregor Gössler

▶ To cite this version:

Gregor Gössler. Reach Scheduling for Embedded Systems. RR-5651, INRIA. 2005. inria-00071230

HAL Id: inria-00071230
https://hal.inria.fr/inria-00071230

Submitted on 23 May 2006

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

Reach Scheduling for Embedded Systems

Gregor Gössler

N° 5651

Août 2005

Thème COM

ISSN 0249-6399 ISRN INRIA/RR--5651--FR+ENG

Reach Scheduling for Embedded Systems

Gregor Gössler

Thème COM — Systèmes communicants
Projet Pop Art

Rapport de recherche n° 5651 — Août 2005 — 17 pages

Abstract: Reachability of a state in an embedded system, and steering the system into that state, are a frequent requirement. The technique we propose ensures reachability by construction. It pre-computes a set of local schedulers which are chosen at run-time depending on the current system state and on a predicate characterizing the set of states to be reached. This quasi-static scheduling approach combines the efficiency of pre-computed schedulers with flexibility during execution, making it possible to change at run-time the predicate to be reached. Our method uses both local controller synthesis and sufficient conditions to compositionally ensure reachability properties, in order to ensure a small memory footprint and efficient execution. The constructed schedulers are composable with further constraints and scheduling policies, which enables an incremental construction. The underlying component model uses transition systems to express the component behaviors, and composition and restriction to express coordination and constraints between components, and allow for heterogeneous models. The results have been implemented and validated in several case studies.

Key-words: reachability, correctness by construction, controller synthesis, quasi-static scheduling, composability, compositionality, embedded systems

Unité de recherche INRIA Rhône-Alpes

Atteignablité par construction pour systèmes embarqués

Résumé : L'atteignabilité d'un état, et la possibilité de contrôler le système dans cet état, sont des besoins fréquents dans la conception de systèmes embarqués. La technique que nous proposons assure l'atteignabilité par construction. Elle pré-calcule un ensemble d'ordonnanceurs locaux qui sont choisis lors de l'exécution en fonction de l'état actuel du système et d'un prédicat caractérisant l'ensemble d'états à atteindre. Cette approche d'ordonnancement quasi-statique combine l'efficacité des ordonnanceurs pré-calculés avec la souplesse à l'exécution et permet de modifier, pendant l'exécution, le prédicat à atteindre. Notre méthode utilise la synthèse locale de contrôleur et des conditions suffisantes pour assurer compositionnellement des propriétés d'atteignabilité, afin d'obtenir un exécutif efficace en taille de mémoire et en temps d'exécution. Les ordonnanceurs construits peuvent être composés avec d'autres contraintes et politiques d'ordonnancement, ce qui rend possible une construction incrémentale. Le cadre de modélisation à base de composants supporte des modèles hétérogènes. Il utilise des systèmes à transitions pour exprimer les comportements des composants. La coordination et les contraintes entre composants sont pris en compte par la composition et une restriction. Les résultats ont été implémentés et validés dans plusieurs études de cas.

Mots-clés : atteignabilité, correction par construction, synthèse de contrôleur, ordonnancement quasi-statique, composabilité, compositionnalité, systèmes embarqés

1 Introduction

With the increasing complexity of embedded real-time systems, coupled with the need for faster time-to-market and high confidence in the reliability of the product, design methods that ensure correctness by construction are, when available, the solution of choice.

In this paper we focus on reachability properties. At *design time*, being able to guide simulation of a complex system to check if and how some particular state can be reached, is often crucial for efficient validation of the system. At *run-time*, guaranteeing reachability of a state and controlling the system so that the state is reached, is a frequent requirement in embedded systems. For example, upon some emergency condition, steer an embedded system into some safe state; enter modes with different quality of service to answer non-functional requests; more generally, react to events by enabling corresponding predicates. Of course, the system (that is, its components and the architecture) may be specifically designed to have this behavior. However, since the predicates may characterize global system states, and the components interfere, this may be a non-trivial job to do manually, which still requires formal verification a posteriori. This work aims at guaranteeing reachability by construction.

Embedded systems design often means *heterogeneous* design. This is the case when different intellectual property (IP) blocks are used for a system-on-a-chip, or more generally, when different models of interaction (e.g., blocking and non-blocking) and execution (e.g., concurrent and run-to-completion) have to be combined. The component model we use supports this heterogeneity. The *behavior* of a component is modeled as a transition system; an *interaction model* specifies how components interact; and a *restriction* constrains the behavior of components, allowing to express and compose different models of execution. As components may block each other, controlling the system so as to reach some state requires scheduling of its components, even if the components are otherwise executed concurrently.

The different methods for correctness-by-construction are compromises between two extreme cases. On one side of the spectrum, sufficient conditions (in the mathematical sense, or in the form of design guidelines) allow for correct systems, with respect to some property to be guaranteed. They are usually easy to compute and apply but are conservative. On the other side, techniques such as controller synthesis restrict an existing system such that exactly every undesirable behavior is prevented, see for example [14]. These techniques are exact (no correct behavior is excluded) but notoriously complex to compute. In addition, the solution should meet the usual requirements placed on embedded systems, in particular, small memory footprint and efficient execution. Therefore, the solution we propose here combines both local controller synthesis and sufficient conditions on the way components constrain each other.

More precisely, we pre-compute a set of local schedulers, each one in charge of leading the system from any (current) state into a state where a given action is enabled. At run-time, the schedulers to be applied are chosen depending on the system state and on the predicate to be reached. Therefore, the system can flexibly react to runtime events. It is important to note that the predicate to be reached may not be known at compile-time. For example, the

system may be led, upon occurrence of some event, into a state depending on that event. It is also possible to concatenate such goals to ensure liveness of a set of actions or components.

This work uses two design principles to efficiently achieve correctness by construction, namely *compositionality* and *composability*. Compositionality means that (global) properties of the system are guaranteed by ensuring properties on its components and on the way they are composed. Composability of a property means that it is still satisfied if the system is further restricted by other constraints. That is, the schedulers we use to ensure reachability, can be composed with other safety or scheduling constraints. A simple sufficient condition establishes consistency of the composition. Both techniques have been extensively studied in [1, 6].

Our method is loosely inspired by previous work using pre-computation of fragments of schedulers which are chosen at run-time to ensure some property. *Quasi-static scheduling* (QSS) [13] is a clever technique to guarantee that no buffer overflow occurs during execution of a free-choice Petri net. The Petri net is partitioned into components for which schedulers are pre-computed. A sequence of local scheduler decisions then yields a correct global execution. QSS has been used for software synthesis, e.g. for task scheduling [4], and extended with time [9]. In the TAXYS project [3, 2], a technique has been studied that allows to pre-compute schedulers that coordinate reactions to events so as to guarantee the respect of timing constraints in reactive systems. [5] describes a debugging tool for reactive programs with numeric variables, where a verification tool is used to (non-compositionally) construct a path to some target state. To our knowledge, there is no previous work on quasi-static scheduling techniques for reachability properties.

The paper is organized as follows. In section 2 we introduce the component model. Section 3 states the problem, presents the main result, and shortly discusses implementation issues. In section 4 we illustrate the method with a case study. Section 5 concludes.

2 Component Model

In the following, we present a simplified version of the component model adopted in [7]. For a set of variables X, let $\mathbf{X}$ denote the set of valuations of X, and let $\mathcal{P}(X) = 2^{\mathbf{X}}$ be the set of predicates on $\mathbf{X}$.

Definition 1 (Transition system) *A* transition system B *is a tuple* (X, A, G, F) *where*

- X *is a finite set of variables;*
- A *is a finite set of actions, union of two disjoint sets* A^u *and* A^c*, the sets of the* uncontrollable *and* controllable *actions, respectively;*
- $G : A \to \mathcal{P}(X)$ *associates with every action its* guard *specifying when the action can occur;*
- $F : A \to (\mathbf{X} \to \mathbf{X})$ *associates with every action its* transition function.

Uncontrollable actions are used to represent input events that cannot be triggered nor prevented by the modeled system. For convenience, we write G^a and F^a for $G(a)$ and $F(a)$, respectively.

Definition 2 (Semantics of a transition system) *A transition system $B = (X, A, G, F)$ defines a* transition relation $\rightarrow$: $\mathbf{X} \times A \times \mathbf{X}$ *such that:* $\forall \mathbf{x}, \mathbf{x}' \in \mathbf{X}\ \forall a \in A\ .\ \mathbf{x} \stackrel{a}{\rightarrow} \mathbf{x}' \iff G^a(\mathbf{x}) \wedge \mathbf{x}' = F^a(\mathbf{x})$.

Notations: as usual, we write $\rightarrow^*$ for the transitive and reflexive closure of $\rightarrow$. Given $B = (X, A, G, F)$, an action $a \in A$ is *reachable* from some state $\mathbf{x}$ if $\exists \mathbf{x}', \mathbf{x}''\ .\ \mathbf{x} \rightarrow^* \mathbf{x}' \wedge x' \stackrel{a}{\rightarrow} \mathbf{x}''$. We represent by $G(B)$ the disjunction of its guards, that is $G(B) = \bigvee_{a \in A} G^a$.

Definition 3 (Predecessors) *Given a transition system $B = (X, A, G, F)$ and a predicate $U \in \mathcal{P}(X)$, the* predecessors *of U by action a is the predicate $pre_a(U) = G^a \wedge U[F^a(X)/X]$ where $U[F^a(X)/X]$ is obtained from U by uniform substitution of X by $F^a(X)$.*

Clearly, $pre_a(U)$ characterizes all the states from which execution of a leads to some state satisfying U.

Definition 4 (Controllable predecessors) *Given $B = (X, A, G, F)$ and $U \in \mathcal{P}(X)$, the predicate $pre\_c(U) = \bigvee_{a \in A^c} pre_a(U) \wedge \neg \bigvee_{a \in A^u} pre_a(\neg U)$ characterizes the* controllable predecessors *of U in one step.*

Let $pre\_c^0(U) = U$, and $pre\_c^{i+1}(U) = pre\_c\big(pre\_c^i(U)\big)$ $(i \geqslant 0)$ be the controllable predecessors of U in i steps.

Let $PRE(U)$ be the least solution of $Y = U \cup pre\_c(Y)$, that is, the controllable predecessors of U in an arbitrary number of steps.

Intuitively, U can be reached from $pre\_c(U)$ by some controllable action, independent of the occurrence of uncontrollable actions.

Definition 5 (Invariant) *Given $B = (X, A, G, F)$ and a predicate $U \in \mathcal{P}(X)$, U is an* invariant *of B if $U \implies \bigwedge_{a \in A} \neg pre_a(\neg U)$. An invariant $U \neq$ false is called* deadlock-free *if $U \implies G(B)$.*

We define two operations on components: composition and restriction. The restriction of a component is a component again, and so is the composition of components.

2.1 Restriction

Restrictions allow to constrain the behavior of a transition system.

Definition 6 (Restriction) *A* restriction *of $B = (X, A, G, F)$ is a tuple of predicates $V = (U^a)_{a \in A^c}$. B/V denotes the transition system B* restricted by *V, $B/V = (X, A, G', F)$ where for any $a \in A^c$, $(G^a)' = G^a \wedge U^a$ is the (restricted) guard of a in B/V. The guards*

of uncontrollable actions remain unchanged. V is a deadlock-free restriction *if* $G(B/V) = G(B)$.

Let $blocked(a) = G^a \wedge \neg(G^a)'$, and let $EA(\mathbf{x}) = \{a \in A \mid (G^a)'(\mathbf{x})\}$ be the set of actions that are enabled at $\mathbf{x}$.

As a special case of restriction, priorities have been shown to be *composable deadlock-free control invariants* [1]. In other words, they allow to ensure safety properties in a modular way, while preserving deadlock freedom.

Definition 7 (Priority) *A* priority *on a set of actions $A = A^c \cup A^u$ is a relation $\prec \subseteq A^c \times A$.*

Notations: for $A' \subseteq A$, let $A'/\prec = \{a \in A' \mid \neg\exists a' \in A' \,.\, a \prec a'\}$ be the set of maximal actions in A'. Given sets of actions A_1, A_2 we write $A_1 \prec A_2$ for $\forall a_1 \in A_1 \, \forall a_2 \in A_2 \,.\, a_1 \prec a_2$.

A *dynamic priority* is a function associating a priority with every state of B.

Definition 8 (Dynamic priority) *A* dynamic priority *on a transition system $B = (X, A, G, F)$ is a tuple of predicates $pr = (C_{ij})_{a_i \in A^c, a_j \in A}$. The restriction defined by pr, $V(B, pr) = (U^a)_{a \in A^c}$ is $U^{a_i} = \bigwedge_{a_j \in A} \neg(C_{ij} \wedge G^{a_j})$.*

The predicates C_{ij} specify priority between actions a_i and a_j. If C_{ij} is true at some state, then in the system restricted by pr the action a_i cannot be executed if a_j is enabled. We write $C_{ij} \mapsto a_i \prec a_j$ to express the fact that a_i is dominated by a_j when C_{ij} holds. Given a dynamic priority pr and a state $\mathbf{x}$, let $pr(\mathbf{x}) = \{a_i \prec a_j \mid C_{ij}(\mathbf{x})\}$ be the priority defined by pr at $\mathbf{x}$. A dynamic priority is *irreflexive* if $C_{ij} \implies \neg C_{ji}$ for all $(a_i, a_j) \in A^c \times A$.

Definition 9 (Transitive closure) *Given a priority $\prec$ we denote by $\prec^+$ the transitive closure of $\prec$. Given a dynamic priority $pr = (C_{ij})_{a_i \in A^c, a_j \in A}$ we denote by pr^+ the least dynamic priority such that $\forall i, j, k \,.\, C_{ij} \wedge C_{jk} \implies C_{ik}$.*

Proposition 1 (Activity preservation) *A dynamic priority pr defines a deadlock-free restriction if pr^+ is irreflexive.*

Definition 10 (Composition $\oplus$ of priorities) *Given two priorities $\prec^1$ and $\prec^2$ their composition is $\prec^1 \oplus \prec^2 = (\prec^1 \cup \prec^2)^+$. Given two dynamic priorities pr^1 and pr^2 with $pr^k = (C^k_{ij})_{a_i \in A^c, a_j \in A}$, $k = 1, 2$, their composition is $pr^1 \oplus pr^2 = \left((C^1_{ij} \vee C^2_{ij})_{a_i \in A^c, a_j \in A}\right)^+$.*

Proposition 2 *The operation $\oplus$ is associative and commutative, and $\forall \mathbf{x} \,.\, (pr^1 \oplus pr^2)(\mathbf{x}) = pr^1(\mathbf{x}) \oplus pr^2(\mathbf{x})$.*

2.2 Composition

Given two actions α_1 and α_2, we write $\alpha_1|\alpha_2$ for the interaction synchronizing both of them. We require $|$ to be commutative, associative, and idempotent. For example, $\alpha_1|(\alpha_2|\alpha_1) = \alpha_1|\alpha_2$.

Definition 11 (Composition) *Let $B_i = (X_i, A_i, G_i, F_i)$, $i = 1, 2$, with $X_1 \cap X_2 = \emptyset$ and $A_1 \cap A_2 = \emptyset$. Let $A_{12} \subseteq \{a_1|a_2 \mid a_1 \in A_1 \wedge a_2 \in A_2\}$. $B_1 \|_{A_{12}} B_2$ is defined as the transition system $(X_1 \cup X_2, A, G, F)$ where*

- *$A = A_1 \cup A_2 \cup A_{12}$ with $A^c = A_1^c \cup A_2^c \cup \{a_1|a_2 \mid a_1 \in A_1^c \wedge a_2 \in A_2^c\}$;*
- *G is defined such that $G(a) = G_i(a)$ if $a \in A_i$, and $G(a) = G_1(a_1) \wedge G_2(a_2)$ if $a = a_1|a_2$ with $a_i \in A_i$, $i = 1, 2$;*
- *$F(a) = F_i(a)$ if $a \in A_i$, and $F(a) = F_1(a_1) \cup F_2(a_2)$ (vector of disjoint functions) if $a = a_1|a_2$ with $a_i \in A_i$, $i = 1, 2$.*

This is the standard synchronized product with interactions A_{12}. A higher-level definition of composition can be found in [7]. It is parameterized with an *interaction model* describing the system architecture.

Proposition 3 (Associativity) $(B_1 \|_{A_{12}} B_2) \|_{A_{12,3}} B_3 = B_1 \|_{A_{1,23}} (B_2 \|_{A_{23}} B_3)$ *if* $A_{12} \cup A_{12,3} = A_{1,23} \cup A_{23}$.

We can therefore extend $\|$ to an n-ary operation, and write $\|_{IC}\{B_1, \ldots, B_n\}$ for the composition of n components with the set of interactions IC.

3 Reachability

We now come to the main result. Based on pre-computed *unblocking schedules*, an algorithm allows to ensure reachability and liveness properties at runtime. Intuitively, the algorithm works as follows.

Consider some action a, and suppose that it is not enabled in the current state. In order to reach a state where a is enabled, choose a product c of component states such that $c \Longrightarrow (G^a)'$, and ask the local scheduler of each component which action to execute in order to approach c. Enabled actions are executed; blocked actions are unblocked by recursively applying this same procedure. A priority relation keeps track of scheduling goals and makes sure that no decision is made that conflicts with previous (higher-level) scheduling goals. In particular, it guarantees that no action is fired that moves the system away from c. A sufficient condition makes sure that a is actually unblocked this way.

In the sequel we consider a system $B = (A, X, G, F) = \|_{IC}\{B_1, \ldots, B_n\}/V$ with $B_i = (A_i, X_i, G_i, F_i)$, $i \in K = \{1, \ldots, n\}$, and V a restriction. For any $a \in A^c$, let $IA(a) = \{a' \in A^c \mid \exists a'' \in A \,.\, a' = a|a''\}$ be the set of controllable interactions encompassing a

(thus, $a \in IA(a)$). We extend IA to sets of actions in the obvious way. For $c \in \mathcal{P}(X)$ a conjunction of literals on X and $k \in K$, let $c|_k$ be the projection (existential abstraction) of c on X_k (thus, $c = \bigwedge_{k\in K} c|_k$). For $U \in \mathcal{P}(X)$ a predicate on X, let $U\bullet = \{a \in A \mid \exists \mathbf{x} \,.\, \big(U \wedge G^a(\mathbf{x})\big) \wedge \neg U\big(F^a(\mathbf{x})\big)\}$ be the set of actions leaving U, and let $c(U)$ be the set of prime implicants of U (thus, $\bigvee_{c\in c(U)} c = U$). For $a \in A \setminus IC$ (the set of component actions), let $owner(a) = k$ such that $a \in A_k$.

Let $s_k : \mathcal{P}(X_k) \times \mathcal{P}(X_k) \to 2^{A_k}$ be a local scheduler of k telling which action to take in order to get from some current state towards a destination state. This scheduler can be computed locally: for any predicate $U \in \mathcal{P}(X)$ and state $\mathbf{x}$ such that $\big(PRE_k(U)\big)(\mathbf{x})$, let

$$s_k(\mathbf{x}, U) = \begin{cases} \emptyset & \text{if } U(\mathbf{x}) \\ \{a \in A_k^c \mid \big(pre_a(pre\_c^{j-1})(U)\big)(\mathbf{x})\} & \text{otherwise} \end{cases}$$

where $j = \min\{i \mid \big(pre\_c^i(U)\big)(\mathbf{x})\}$ is the number of steps to steer B from $\mathbf{x}$ to U. Let $d_k(\mathbf{x}, U) = j$ denote this distance, and let $d(\mathbf{x}, U) = \Sigma_{k\in K} d_k(\mathbf{x}, U)$. Intuitively, $s_k(\mathbf{x}, U)$ is the set of actions that, when executed in state $\mathbf{x}$, bring B_k closer to U, while keeping the component within the controllable predecessors $PRE_k(U)$. In computing s_k we assume that components are sufficiently small to allow local controller synthesis; otherwise large components have to be further decomposed, or an abstraction be used to compute the schedulers.

Let $s : \mathcal{P}(X) \times \mathcal{P}(X) \to 2^A$ be the "union" of the local schedulers such that $s(\mathbf{x}, \mathbf{y}) = \bigcup_{k\in K}\{s_k(\mathbf{x}|_k, \mathbf{y}|_k)\}$. Let $s_k(U) = \bigcup_{\mathbf{x}} s_k(\mathbf{x}, U)$, and $s(U) = \bigcup_{\mathbf{x}} s(\mathbf{x}, U)$ be the set of all actions used by s to reach U.

Definition 12 (Unblocking order) *For $a \in A$, $c \in \mathcal{P}(X)$ a conjunction on literals in X, and $k \in K$, let*

$$pr^{a,c,k}_{sched} = \bigoplus_{i\geqslant 1} \big(C \mapsto \{a \prec b \mid a, b \in IA(A_k^c) \ \wedge \ pre_a(\neg C') \wedge pre_b(C') \neq \mathit{false}\}\big)$$

where $C = blocked(a) \wedge pre\_c_k^i(c|_k)$ and $C' = pre\_c_k^{i-1}(c|_k)$.

The unblocking order $pr^{a,c,k}_{sched}$ resolves conflicts within component B_k, by giving priority to actions leading towards c over actions leading away from c, in order to unblock a.

Definition 13 (Unblocking schedule) *Given $a \in A^c$ with $(G^a)' \neq G^a$, we call*

$$unblock(a) = \Big\{\Big(c,\ \big(blocked(a) \mapsto c\bullet \cup \{a\} \prec IA(s(c))\big) \oplus \bigoplus_{k\in K} pr^{a,c,k}_{sched}\Big) \mid c \in c\big((G^a)'\big) \wedge c\bullet \subseteq A^c\Big\}$$

the set of unblocking schedules *of a.*

An unblocking schedule for some action a is thus a tuple of a (convex) predicate in which a is enabled, and a dynamic priority. The latter is composed of an unblocking order for each component, and a dynamic priority giving priority to scheduler actions leading towards c over actions leaving c.

An *unblocking domain* specifies the states for which unblocking schedules are defined:

Definition 14 (Unblocking domain) *Given a conjunction c on X, let $reach(c) = \bigwedge_{k\in K} PRE_k(c|_k)$ be the* unblocking domain *of c.*

Algorithm 1 Algorithm enabling action a.

while $\neg(G^a)'(\mathbf{x})$ do
  $a' :=$ choice $\left(IA(s^*(\mathbf{x},a)^A)\right)$;
  $\mathbf{x} := F^{a'}(\mathbf{x})$
od

Consider some action $a \in A^c \setminus IC$. To effectively reach a state where a is enabled, we compute, for a current state $\mathbf{x}$, a sequence of scheduling decisions that converges towards the choice of an action that brings the system closer to a state enabling a. That is, if a is not enabled, fire some action a' moving the system towards a state where a is enabled; if no such a' is enabled, fire some a'' bringing the system closer to a', and so on. In doing this, we have to disable actions increasing the distance of any component from c. This is what algorithm 1 does. $s^*(\mathbf{x},a)^A$ is the fixed-point of a sequence $[s^i(\mathbf{x},a)^A]_{i\geqslant 0}$, where any element $s^i(\mathbf{x},a)$ is a *scheduling decision*. It is a tuple $(c^i, \prec^i, A^i)$ consisting of a predicate to be reached, a priority to coordinate system execution so as to avoid conflicts with previous scheduling decisions $s^j(\mathbf{x},a)$ with $j<i$, and a set of actions to be scheduled in order to approach c^i. The sequence of scheduling decisions is defined as follows. $s^0(\mathbf{x},a) = \left(c,\ pr(\mathbf{x}),\ IA(s(\mathbf{x},c))\right)$ for some $(c,pr) \in unblock(a)$ such that $reach(c)(\mathbf{x})$ and $\forall (c',pr') \in unblock(a)$ with $reach(c')(\mathbf{x})$, $d(\mathbf{x},c) \leqslant d(\mathbf{x},c')$ (that is, choose some optimal unblocking schedule). For any $i \geqslant 0$, let

$$s^{i+1}(\mathbf{x},a) = \begin{cases} s^i(\mathbf{x},a) & \text{if } legal\_enabled(a,i,\mathbf{x}) \neq \emptyset \\ \left(s^0(\mathbf{x},a')^c, \prec, s^0(\mathbf{x},a')^A\right) & \text{otherwise} \end{cases}$$

where $legal\_enabled(a,i,\mathbf{x}) = EA(\mathbf{x})/s^i(\mathbf{x},a)^{\prec} \cap IA(s^i(\mathbf{x},a)^A)$, is the set of actions to be scheduled that are enabled, $a' = choice(A/s^i(\mathbf{x},a)^{\prec} \cap s^i(\mathbf{x},a)^A)$, $\prec = s^i(\mathbf{x},a)^{\prec} \oplus s^0(\mathbf{x},a')^{\prec}$, and *choice* is an arbitrary (but deterministic) choice.

Theorem 1 (Reachability) *All actions in $A^c \setminus IC$ are reachable in $B = (X,A,G,F)$ if*

1. $\forall a \in A^c \setminus IC \,.\, G^a \implies \bigvee_{(c,pr)\in unblock(a)} reach(c)$, *and*
2. $pr_{consist} = \bigoplus_{a\in A^c\setminus IC} \bigoplus_{(c,pr)\in unblock(a)} pr$ *is irreflexive.*

Intuitively, reachability is guaranteed if all components are controllable with respect to any of their action guards, and the composed unblocking orders are consistent in the worst case.

Proof. Consider some controllable component action $a \in A^c \setminus IC$ and state $\mathbf{x}$. We show that $(G^a)'$ is reached from $\mathbf{x}$ by applying algorithm 1.

We write $\bar{s}(\mathbf{x}, a)$ for the list $[s^0(\mathbf{x}, a), s^1(\mathbf{x}, a), \dots]$, and $\bar{s}(\mathbf{x}, a)^c$ for the list $[c^0, c^1, \dots]$. Let $a^0 = a$ and $a^{i+1} = choice(A/s^i(\mathbf{x}, a)^{\prec} \cap s^i(\mathbf{x}, a)^A)$, $i \geqslant 0$.

First we show that for any $i \geqslant 0$ and any action b that is non-maximal in $\prec^i$, $b \prec^i A^i$. This is the case for $i = 0$ by definition of unblocking schedules. Suppose that $b \prec^i A^i$. Since $a^{i+1} \in A^i$, $b \prec^i a^{i+1} \prec^{i+1} A^{i+1}$ by definition of $unblock(a^{i+1})$, and with $\prec^i \subseteq \prec^{i+1}$, $b \prec^{i+1} A^{i+1}$. By consistency (2) it follows that all actions in A^i are maximal in $\prec^i$.

s^i is well-defined for any $i \geqslant 0$, in the sense that

- $reach(c)(\mathbf{x})$ for all used unblocking schedules (c, pr), that is, all components not in c are controllable with respect to c,
- $A^i \neq \emptyset$ by the fact that $\prec^i$ is irreflexive according to hypothesis (2), and
- $A/s^i(\mathbf{x}, a)^{\prec} \cap s^i(\mathbf{x}, a)^A \neq \emptyset$ by activity preservation and maximality of the actions $s^i(\mathbf{x}, a)^A$ in $s^i(\mathbf{x}, a)^{\prec}$.

The sequence $[s^i]_i$ eventually converges, that is, there is some i such that $legal\_enabled(a, i, \mathbf{x}) \neq \emptyset$: suppose on the contrary that $\forall j \;.\; legal\_enabled(a, i, \mathbf{x}) = \emptyset$. By (1) and definition of $pr_{consist}$, $\mathbf{x} \mapsto a^i \prec_{consist} a^{i+1}$ for any $i \geqslant 0$ such that $legal\_enabled(a, i, \mathbf{x}) = \emptyset$. Finiteness of A implies the existence of a circuit in $pr_{consist}$, which is in contradiction to hypothesis (2). Thus, $[s^i]_i$ converges, that is, some action to be scheduled is enabled in $\mathbf{x}$. Let $s^*(\mathbf{x}, a) = s^j(\mathbf{x}, a)$ such that $j = \min\{i \mid legal\_enabled(a, i, \mathbf{x}) \neq \emptyset\}$, and let $(c^*, \prec^*, A^*) = s^*(\mathbf{x}, a)$.

Next we show that a is effectively reached by repeatedly firing some action in A^*. Take some κ such that $\kappa > \Sigma_{k \in K} \max_{a \in A_k} \max_{(c,pr) \in unblock(a)} \max_{\mathbf{x}} d_k(\mathbf{x}, c)$. We proceed by induction on the distance (in the number of steps) of $\mathbf{x}$ from c^0 under the algorithm. Let $dist^0(\mathbf{x}, a) = \kappa^{|A|} \times d(\mathbf{x}, c^0)$, $dist^{i+1}(\mathbf{x}, a) = dist^i(\mathbf{x}, a) + \kappa^{|A|-(i+1)} \times d\big(\mathbf{x}, c^{i+1}\big)$ $(i \geqslant 0)$, and $dist^*(\mathbf{x}, a) = dist^j(\mathbf{x}, a)$ where $j = \min\{i \mid legal\_enabled(a, i, \mathbf{x}) \neq \emptyset\}$. With every execution of the loop body, $dist^*(\mathbf{x}, a)$ strictly decreases: by definition of $\prec^*$, for any $i \geqslant 0$, $k \in K$ with $\neg(c^i|_k)(\mathbf{x})$, and action $a_1 \in IA(A_k^c)$ such that $pre_{a_1}\big(\neg \bigvee_{j \leqslant i-1} pre\_c^j(U)\big)(\mathbf{x})$ there is some action $a_2 \in IA(A_k^c)$ such that $pre_{a_2}\big(pre\_c^{i-1}(U)\big)(\mathbf{x})$ and $a_1 \prec_{sched}^{a^i, c^i, k} a_2$, and thus $a_1 \prec^* A^*$. By definition of "/", no action increasing the distance from any of the predicates in $\bar{s}(\mathbf{x}, a)$ is ever taken. Similarly, by construction of $\prec^*$, no predicate c^i is ever left by a component, unless some higher-ranked c_j $(j < i)$ is approached. Thus, either $\bar{s}(\mathbf{x}, a)^c = \bar{s}(\mathbf{x}', a)^c$ and $\mathbf{x}'$ is closer to c^* and at least as close to all other c^i $(i \geqslant 0)$, or some c^i is reached. In both cases, $dist^*(\mathbf{x}', a) < dist^*(\mathbf{x}, a)$. It follows that $dist^*(\mathbf{x}, a) = 0$ after a finite number of steps, and thus $G'_a(\mathbf{x})$. ∎

This is a remarkable result for two reasons. First, it is not parameterized by a property to guarantee. Unlike most controller synthesis algorithms it allows to construct a controller not just guaranteeing reachability of some set of states, but of the set of states enabling some action that is only known at runtime. Second, it constructs a set of unblocking schedules whose unblocking orders are local to the components. Each of the unblocking schedules is in charge of doing a limited amount of work, namely unblocking a specific action under a specific condition, before enabling another unblocking schedule.

The construction of the unblocking schedules and the subsequent consistency check are the more expensive part, and pre-computed. As the methodology is based on an abstraction of the system behavior, it may be pessimistic in the sense that the hypothesis may not be satisfied although the property is satisfied.

Corollary 1 (Reachability) *Let $U \in \mathcal{P}(X)$ be a predicate on $\mathbf{X}$. U is reachable if an observer component $o(U) = (\emptyset, \{alive\}, \{G^{alive} = true\}, \{F^{alive} = id_\emptyset\})$ with restriction $U^{alive} = U$ is deadlock-free in $B \|_\emptyset o(U)/U^{alive}$. Moreover, a scheduler constructed as above controls the system so that U is effectively reached.*

Notice that composing with observer $o(U)$ and restricting with U^{alive} do not change the behavior of the other components. Of course, if some action a already has guard $(G^a)' = U$, no observer needs to be added to check for reachability of U and construct a scheduler.

Example 1 Figure 1 shows three components modeling a scheduler with two processes. All actions are controllable and there is no synchronization between actions, but the following constraints need to be ensured: c1.run and c2.run are only enabled if sched is in state enable1 and enable2, respectively; sched.choose1 and sched.choose2 are disabled as long as the other processes is running, and sched.done is disabled while no process is running. We thus have the restriction $U^{\mathsf{c1.run}} = \mathsf{sched.enable1}$, $U^{\mathsf{c2.run}} = \mathsf{sched.enable2}$, $U^{\mathsf{sched.choose1}} = \neg\mathsf{c2.running}$, $U^{\mathsf{sched.choose2}} = \neg\mathsf{c1.running}$, $U^{\mathsf{sched.done}} = \mathsf{c1.running} \vee \mathsf{c2.running}$, and $U^a = true$ for all other actions a.

According to definition 13 there are six unblocking schedules (we omit the pre-conditions of the dynamic priorities for the sake of readability): $unblock(\mathsf{c1.run}) = \big\{\big(\neg\mathsf{c1.running} \wedge \mathsf{sched.enabled1}, \{\mathsf{sched.choose2} \prec \mathsf{sched.choose1}, \mathsf{c1.run} \prec \mathsf{sched.choose1}, \mathsf{c1.run} \prec \mathsf{sched.enable1}, \mathsf{c1.run} \prec \mathsf{sched.enable2}, \mathsf{c1.run} \prec \mathsf{sched.done}\}\big)\big\}$, that is, in order to reach $\neg\mathsf{c1.running} \wedge \mathsf{sched.enabled1}$ and unblock c1.run, give priority to sched.choose1 over sched.choose2. Similarly, $unblock(\mathsf{c2.run}) = \big\{\big(\neg\mathsf{c2.running} \wedge \mathsf{sched.enabled2}, \{\mathsf{sched.choose1} \prec \mathsf{sched.choose2}, \mathsf{c2.run} \prec \mathsf{sched.choose2}, \mathsf{c2.run} \prec \mathsf{sched.enable1}, \mathsf{c2.run} \prec \mathsf{sched.enable2}, \mathsf{c2.run} \prec \mathsf{sched.done}\}\big)\big\}$. Furthermore, $unblock(\mathsf{sched.choose1}) = \big\{\big(\neg\mathsf{c2.running} \wedge \mathsf{sched.idle}, \{\mathsf{sched.choose1} \prec \mathsf{c2.stop}\}\big)\big\}$, $unblock(\mathsf{sched.choose2}) = \big\{\big(\neg\mathsf{c1.running} \wedge \mathsf{sched.idle}, \{\mathsf{sched.choose2} \prec \mathsf{c1.stop}\}\big)\big\}$, and $unblock(\mathsf{sched.done}) = \big\{\big(\mathsf{c1.running} \wedge \mathsf{sched.enabled}, \{\mathsf{sched.done} \prec \mathsf{c1.run}\}\big), \big(\mathsf{c2.running} \wedge \mathsf{sched.enabled}, \{\mathsf{sched.done} \prec \mathsf{c2.run}\}\big)\big\}$. Therefore there is at least one unlocking order for each restricted action. The unblocking domains of all unblocking schedules are *true*, that is, they apply to all states.

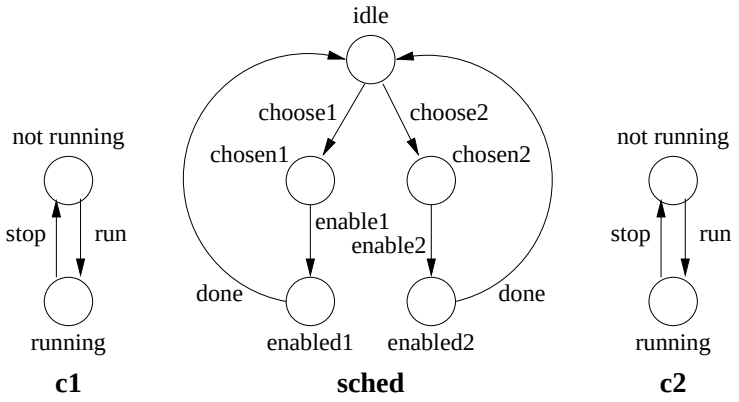

Figure 1: Scheduler with two processes.

Let us have the algorithm unblock c2.run from $\mathbf{x} =$ sched.idle $\wedge$ c1.running. The constructed sequence of scheduling decisions is:

1. $s^0(\mathbf{x}, \text{c2.run}) = \big(c, \{\text{sched.choose1} \prec \text{sched.choose2}, \text{c2.run} \prec \text{sched.choose2}\}, s(\mathbf{x}, \text{c2.run})\big) = \big(c, \{\text{sched.choose1} \prec \text{sched.choose2}, \text{c2.run} \prec \text{sched.choose2}\}, \{\text{sched.choose2}\}\big)$ where $c =$ sched.enabled2 $\wedge \neg$c2.running. As the requested action sched.choose2 is not enabled, the goal of the next scheduling decision is to enable it:

2. $s^1(\mathbf{x}, \text{c2.run}) = s^0(\mathbf{x}, \text{sched.choose2}) = \big(c', \{\text{sched.choose1} \prec \text{sched.choose2}, \text{c2.run} \prec \text{sched.choose2}\} \oplus \{\text{sched.choose2} \prec \text{c1.stop}\}, s(\mathbf{x}, \text{sched.choose2})\big) = \big(c', \{\text{sched.choose1} \prec \text{sched.choose2}, \text{c2.run} \prec \text{sched.choose2}, \text{sched.choose2} \prec \text{c1.stop}, \text{c2.run} \prec \text{c1.stop}, \text{sched.choose1} \prec \text{c1.stop}\}, \{\text{c1.stop}\}\big)$ where $c' =$ sched.idle $\wedge$ $\neg$c1.running. Action c1.stop is enabled and fired. The scheduling decision $s^1(\mathbf{x}, \text{c2.run})$ has now led the system to state $\mathbf{x}' =$ sched.idle $\wedge \neg$c1.running.

3. $s^0(\mathbf{x}', \text{c2.run}) = \big(c, \{\text{sched.choose1} \prec \text{sched.choose2}, \text{c2.run} \prec \text{sched.choose2}\}, \{\text{sched.choose2}\}\big)$. sched.choose2 is fired, leading the system to $\mathbf{x}'' =$ sched.chosen2 $\wedge$ $\neg$c1.running.

4. $s^0(\mathbf{x}'', \text{c2.run}) = \big(c, \{\text{c2.run} \prec \text{sched.enable2}\}, \{\text{sched.enable2}\}\big)$. sched.enable2 is fired and leads to $\mathbf{x}''' =$ sched.enabled2 $\wedge \neg$c1.running.

c1.run enabled in $\mathbf{x}'''$, the goal has been reached. If c1.stop had been disabled in state $\mathbf{x}$, the priority $\{\text{sched.choose1} \prec \text{sched.choose2}, \text{sched.choose1} \prec \text{c1.stop}\}$ of scheduling decision $s^1(\mathbf{x}, \text{c2.run})$ would have prevented the firing of sched.choose1, which would be the wrong decision with respect to the goal of enabling c2.run.

Implementation. Algorithm 1 can be implemented as a set of local schedulers for each component, and a global scheduler. The latter activates, depending on its scheduling decisions, at most one local schedulers for each component. For simplicity of implementation, it may be more convenient to use generalized static priorities as proposed in [6], rather than dynamic priorities. The former are defined on the set of enabled actions, and do not require observability of the current system state. Figure 2 illustrates a possible architecture with its interactions between schedulers. The actual implementation will depend on the given platform.

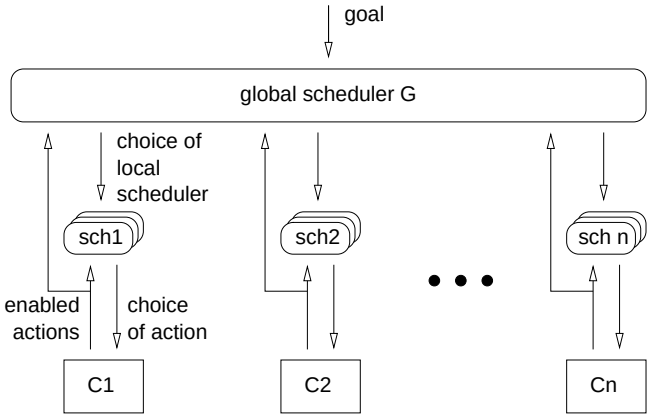

Figure 2: Interaction between schedulers G, sch_i and components C_i.

4 Case Study

Algorithm 1 has been implemented in the PROMETHEUS tool. We illustrate the construction of unblocking schedules and their application with a transaction level model (TLM) of two master processes that can call slave processes via a shared bus. The example, taken from [12], is originally written in SystemC [10, 8], using a TLM library provided by ST Microelectronics. We have rewritten it in the input language of PROMETHEUS, slightly simplifying it but preserving the SystemC semantics.

The example system consists of five components: two processes status_master and signal_master, a bus arbiter tac_seq, and two slave processes status_slave and signal_slave. In order to trigger their respective slave process, the master processes send a request to the bus arbiter, along with the address of the slave process to be called, and some optional data to be transmitted. The bus arbiter calls the slave process mapped to the address, or sets a status flag if the address does not correspond to any of the slave processes, and then returns control to the invoking process. signal_master sets a signal to *false* and then triggers sig-

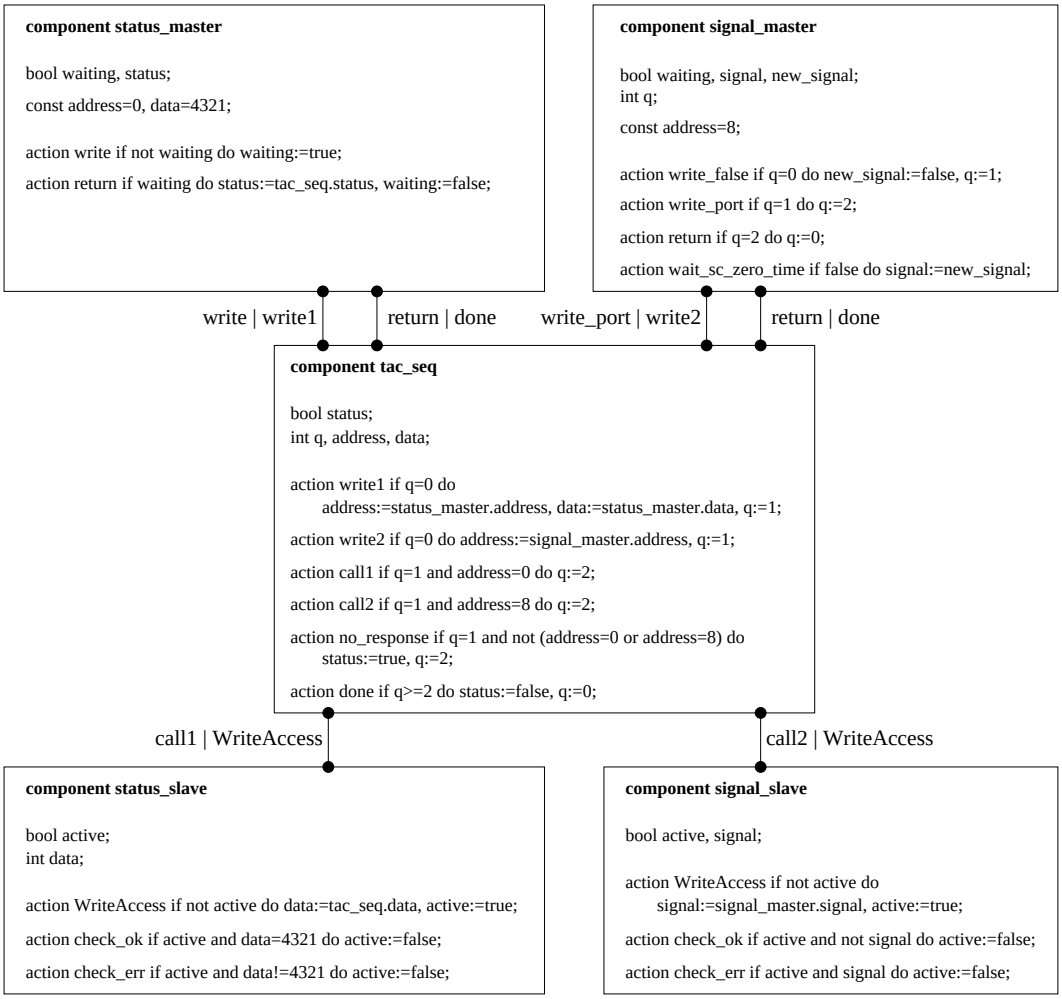

Figure 3: Four processes communicating over a shared bus.

nal_slave. The latter reads the signal and checks that it is *false*. However, according to the SystemC semantics, signal_slave does not read the newly written value unless logical time has progressed, modeled by a wait_sc_zero_time action of signal_master. This model uses the possibility to read the variables of another component. These are extensions we have

chosen not to present in the framework of section 2 for the sake of simplicity; our approach can deal with them with some minor generalizations.

All actions are controllable. In figure 3, interactions between components are sketched by lines between the component boxes. The full circles mean that all interactions are blocking rendez-vous. In our component model this is expressed by the restriction $U^a = \mathit{false}$ for $a \in$ {status_master.write, status_master.return, signal_master.write_port, signal_master.return, tac_seq.write1, tac_seq.write2, tac_seq.call1, tac_seq.call2, tac_seq.done, status_slave.WriteAccess, signal_slave.WriteAccess}, and $U^a = \mathit{true}$ for all other actions. PROMETHEUS constructs 13 unblocking schedules required to unblock any of the actions. The first condition of theorem 1 is found to be satisfied, whereas the second one is not.

We want to check correctness of the model, in the sense that signal_slave always reads *false*. That is, we are interested in the reachability of action signal_slave.check_err. By applying the unblocking schedules, PROMETHEUS finds a path from the initial state tac_seq.q $=$ $0 \wedge$ signal_master.q $= 0 \wedge \neg$signal_slave.active to the error state: $\langle$signal_master.write_false, signal_master.write_port|tac_seq.write2, tac_seq.call2|signal_slave.WriteAccess$\rangle$.

Finally, in tac_seq.q $= 2 \wedge$ tac_seq.address $= 8 \wedge$ signal_master.q $= 2 \wedge$ $\neg$signal_master.signal $\wedge$ signal_slave.active, action signal_slave.check_err is enabled. That is, we have found a path refuting the claim that signal_slave always reads *false*. In this example, the shortest path has been found. PROMETHEUS constructs the unblocking schedules in the fraction of a second; construction of the failure path takes less than 10 ms.

5 Conclusion

We have presented a method for the compositional construction of schedulers ensuring reachability of all controllable actions of the system. The constructed unblocking schedules are composable with priorities ensuring other properties, for instance safety properties or real-time scheduling policies. The results presented in this paper have been implemented in the PROMETHEUS tool for compositional verification of reactive systems, and tested with different systems and models of computation. For instance, the model of a genetic regulatory network consisting of 11 components (with a state space of half a million states) and complex restriction is handled by PROMETHEUS in less than one second, including construction of the schedules. A path of length 16 enabling a requested action from some initial state is then constructed in 40 ms.

We intend to apply these results to the compositional verification and design of systems-on-a-chip, in the context of an ongoing project with ST Microelectronics. The second application domain we are interested in, is the analysis of genetic regulatory networks. The results are encouraging, as suggested by the case study mentioned above.

One can think of several useful extensions to the scheduler construction algorithm. As case studies suggest that reachability can be ensured even when the hypothesis of theorem 1 is not satisfied, we are examining how partial reachability can be guaranteed, and appropriate diagnostics be provided in that case.

Since the algorithm is essentially based on light-weight decentralized schedulers, it seems interesting to study the application of our results to distributed systems and in particular to sensor networks, consisting of a large number of nodes with limited resources. Closely related is the question how the scheduler construction algorithm can be made incremental so as to allow for reconfigurable systems in which the architecture can be changed (e.g. by dropping or adding components) at run-time.

As our algorithm constructs schedulers to ensure reachability properties for an already existing system, they can be seen as a cross-cutting property that is added to the system, in the sense of aspect-oriented programming [11]. This relation deserves to be studied more closely; a combination of both techniques could be used to ensure new behaviors, such as reinitialization, by construction.

References

[1] K. Altisen, G. Gössler, and J. Sifakis. Scheduler modeling based on the controller synthesis paradigm. *Journal of Real-Time Systems, special issue on "control-theoretical approaches to real-time computing"*, 23(1/2):55–84, 2002.

[2] V. Bertin, E. Closse, M. Poize, J. Pulou, J. Sifakis, P. Venier, D. Weil, and S. Yovine. Taxys = Esterel + Kronos — a tool for verifying real-time properties of embedded systems. In *Proc. CDC'01*, 2001.

[3] E. Closse, M. Poize, J. Pulou, J. Sifakis, P. Venier, D. Weil, and S. Yovine. TAXYS = ESTEREL + KRONOS. a tool for the development and verification of real-time embedded systems. In *Proc. CAV'01*, volume 2102 of *LNCS*, pages 391–395. Springer-Verlag, 2001.

[4] J. Cortadella, A. Kondratyev, L. Lavagno, and Y. Watanabe. Quasi-static scheduling for concurrent architectures. In *proc. ACSD'03*, 2003.

[5] F. Gaucher, E. Jahier, B. Jeannet, and F. Maraninchi. Automatic state reaching for debugging reactive programs. In *proc. AADEBUG'03*, 2003.

[6] G. Gössler and J. Sifakis. Priority systems. In F. de Boer, M. Bonsangue, S. Graf, and W.-P. de Roever, editors, *proc. FMCO'03*, volume 3188 of *LNCS*, pages 314–329. Springer-Verlag, 2004.

[7] G. Gössler and J. Sifakis. Composition for component-based modeling. *Science of Computer Programming*, 55(1-3):161–183, 2005.

[8] T. Grötker, S. Liao, G. Martin, and S. Swan. *System Design with SystemC*. Kluwer, 2002.

[9] P.-A. Hsiung and F-S. Su. Synthesis of real-time embedded software by timed quasi-static scheduling. In *proc. VLSI'03*, pages 579–584. IEEE CS Press, 2003.

[10] Open SystemC Initiative. SystemC. `http://www.systemc.org`.

[11] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Videira Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proc. ECOOP '97*, volume 1241 of *LNCS*, page 220ff. Springer-Verlag, 1997.

[12] M. Moy, F. Maraninchi, and L. Maillet-Contoz. LusSy: A toolbox for the analysis of system-on-a-chip at the transactional level. In *proc. ACSD'05*, 2005.

[13] M. Sgroi, L. Lavagno, Y. Watanabe, and A. Sangiovanni-Vincentelli. Synthesis of embedded software using free-choice Petri nets. In *Proc. DAC'99*, pages 805–810, 1999.

[14] W.M. Wonham and P.J. Ramadge. Modular supervisory control of discrete event systems. *Mathematics of Control Signals and Systems*, 1(1):13–30, 1988.

Unité de recherche INRIA Rhône-Alpes
655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)

Unité de recherche INRIA Futurs : Parc Club Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex (France)
Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

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