Assume-Guarantee Reasoning for Hierarchical Hybrid Systems*

Thomas A. Henzinger Marius Minea Vinayak Prabhu

Dept. of EECS, University of California, Berkeley, CA 94720, USA
{tah,marius,vinayak}@eecs.berkeley.edu
www.eecs.berkeley.edu/~{tah,marius,vinayak}

Abstract. The assume-guarantee paradigm is a powerful divide-and-conquer mechanism for decomposing a verification task about a system into subtasks about the individual components of the system. The key to assume-guarantee reasoning is to consider each component not in isolation, but in conjunction with assumptions about the context of the component. Assume-guarantee principles are known for purely concurrent contexts, which constrain the input data of a component, as well as for purely sequential contexts, which constrain the entry configurations of a component. We present a model for hierarchical system design which permits the arbitrary nesting of parallel as well as serial composition, and which supports an assume-guarantee principle for mixed parallel-serial contexts. Our model also supports both discrete and continuous processes, and is therefore well-suited for the modeling and analysis of embedded software systems which interact with real-world environments. Using an example of two cooperating robots, we show refinement between a high-level model which specifies continuous timing constraints and an implementation which relies on discrete sampling.

1 Introduction

In the automatic verification of systems with very large state spaces, the model-checking task needs to be decomposed into subtasks of manageable complexity. It is natural to decompose the verification task following the component structure of the design. However, an individual component often does not satisfy its requirements unless the component is put into the right context. Thus, in order to verify each component individually, we need to make assumptions about its context, namely, about the other components of the design. This reasoning is circular: component A is verified under the assumption that context B behaves correctly, and symmetrically, B is verified assuming the correctness of A. The assume-guarantee paradigm provides a systematic theory and methodology for ensuring the soundness of the circular style of postulating and discharging assumptions in component-based reasoning.

* Support for this research was provided in part by the AFOSR MURI grant F49620-00-1-0327, and the DARPA SEC grant F33615-C-98-3614, the MARCO GSRC grant 98-DT-660, the NSF ITR grant CCR-0085949.

When components are composed in parallel, context assumptions constrain the inputs to a component. Assume-guarantee principles for parallel composition are advocated, among others, by [MC81,AL95,McM97,AH99], and by [TAKB96,AH97] in a real-time setting. If components are composed in series, context assumptions constrain the entry configurations of a component. An assume-guarantee principle for serial composition is given in [AG00]. In hierarchical design, it is often useful to nest parallel and serial composition. This is especially true for embedded software, where serial composition occurs at multiple levels of granularity (e.g., software procedures; modes of operation; exception handling), and so does parallel composition (e.g., hardware modules; software threads; environment interaction). We provide an assume-guarantee principle for the case where a context can contain both parallel and serial components, arbitrarily nested.

For this purpose, we use a formal model which is called Masaccio, in honor of the Italian fresco painter who is credited with inventing perspective. The Masaccio language was defined in [Hen00]; we modify it slightly in order to obtain a general assume-guarantee principle. Masaccio is a formal model for hybrid dynamical systems which are built from atomic discrete components (difference equations) and atomic continuous components (differential equations) by parallel and serial composition, arbitrarily nested. Data is represented by variables; control by locations. The syntax of components includes six operations: besides parallel and sequential composition, data connections are built by variable renaming, control connections by location renaming, data abstractions by variable hiding, and control abstractions by location hiding. The formal semantics of each component consists of an interface, which determines the possible ways of using the component, and a set of executions, which define the possible behaviors of the component in real time. The intended use of Masaccio is to provide a formal, structured model for software and hardware that interacts with a physical environment in real time. Parallel composition is conjunctive: it typically combines actors (software threads, sensors, actuators, etc.); serial composition is disjunctive: it typically combines modes of operation (time-triggered and event-triggered mode switching, degraded and fault modes, etc.). Masaccio conservatively extends Reactive Modules [AH99,AH97], which provide parallel but no serial composition, and it inherits the mixing of discrete and continuous behavior from Hybrid Automata [ACH$^+$95,Hen96], which are not hierarchical. The parallel composition of Masaccio is synchronous; asynchronicity can be modeled as in [AH99].

We demonstrate that Masaccio supports hierarchical, component-based design and analysis. In particular, we prove the soundness of (noncircular) compositional proof rules for both parallel and serial composition, and the soundness of a (circular) assume-guarantee proof rule, which permits assumptions about mixed parallel-serial contexts. Several key insights are necessary to enable the assume-guarantee principle. First, assume-guarantee reasoning is sound only for components that cannot deadlock internally. We therefore equip the interface of a component with entry conditions and insist that a location can be hidden

only if the corresponding entry condition is valid. Second, if two components A and B are composed in series, the assume-guarantee principle is sound only if each trace of the composite system $A + B$ can be assigned uniquely to either A or B. This can be achieved by requiring that for all locations common to A and B, the entry conditions are disjoint. Third, if A and B are composed in parallel, we wish to model the fact that either component may preempt the other on termination, causing $A||B$ to terminate. Therefore, in refinement, B is more specific than C not only if every trace of B is a trace of C, but also if every trace of B has a prefix (possibly generated if B is preempted) which is a trace of C. This novel notion of refinement is consistent with sequential composition: a trace may terminate at an exit location of a component, and the serial addition of another component can then provide it with a continuation. Thus, a prefix of a trace is more general than the trace itself, since it potentially allows several different continuations. It will follow that both parallel and serial composition are congruences with respect to refinement.

We illustrate our formalism by modeling at different levels of detail a system of two cooperating robots, one of which is always following the other. The specification requires that a request by one robot to lead is honored within a certain time bound by the other robot starting to follow. We give an implementation that relies on periodic sampling of the robot states, and show how assume-guarantee reasoning simplifies the task of refinement checking between implementation and specification.

Related work Concurrent and sequential hierarchies have long been nested in informal and semiformal ways (for instance, Statecharts [Har87], UML [BRJ98], Ptolemy [DGH$^+$99]). While these languages enjoy considerable acceptance as good engineering practice, the most widely used versions of these languages do not support compositional formal analysis. For Statecharts, variants with compositional semantics have been defined (see, e.g., [US94]), but an assume-guarantee paradigm is not known. Hierarchic Modules [AG00] provide an assume-guarantee principle for serial composition, and parallel composition is reduced to serial composition. No continuous behaviors are considered. The languages Shift [DGV97] and Charon [AGH$^+$00] support the hierarchical design of hybrid systems, but its emphasis is on simulation, and serial and parallel composition cannot be nested arbitrarily. The model of Hybrid I/O Automata [LSVW96] offers compositionality in a setting without serial composition.

2 The Masaccio Model for Embedded Components

In Masaccio, a system model is built out of *components*. We illustrate Masaccio by modeling parts of a system with two communicating robots, which will be used in Section 4; the formal definition of Masaccio is given in the appendix. The semantics of a component is defined by its interface ("structure") and its set of executions ("behavior"). The executions are *hybrid*: the state of a component may evolve by any sequence of discrete transitions (so-called *jumps*) and continuous evolutions (*flows*).

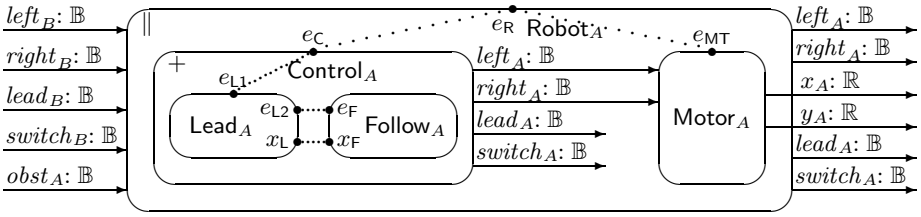

Fig. 1. Robot specification

The interface of a component The interface of a component determines how the component can be composed (i.e., can interact) with other components. In Masaccio, control and data are handled separately. The interface of a component A contains a set V_A of variables partitioned into *input variables* and *output variables*, and a set L_A of *interface locations*, through which control can enter and/or exit the component. All variables are typed, with domains such as the booleans $\mathbb{B}$, the natural numbers $\mathbb{N}$, and the reals $\mathbb{R}$. While control resides inside a component, the input variables are updated by the environment (such as another component put in parallel), and the output variables are updated by the component. The component interface specifies a *dependency relation* $\prec_A$ between I/O variables and output variables. If $x \prec_A y$, then the value of y can depend without delay on the value of x. Specifically, with each jump, the new value of output y can depend on the new value of (say) input x, and during a flow, the derivative of output y can depend on the simultaneous derivative of input x. The dependency relation must be acyclic, in order to guarantee the existence of suitable output values and output curves.

An I/O state of the component is a value assignment to the variables in V_A. The component interface specifies for each location $a \in L_A$ a *jump entry condition* $\psi_A^{jump}(a)$ and a *flow entry condition* $\psi_A^{flow}(a)$. The component can be entered by a jump iff the jump entry condition is satisfied by the current I/O state, and by the new values of the input variables; the component can be entered by a flow iff the flow entry condition is satisfied by the current I/O state. The length of a flow may be constrained by the component, but whenever the flow entry condition is satisfied, at least a flow of duration 0 is possible. Control can exit the component at every location. In typical designs, the exit points are the locations with unsatisfiable entry conditions.

As an example, we portray a scenario in which two similar robots, structured as in Figure 1, move around in an environment with obstacles. The robots jointly choose the strategy of one leading and the other following, and their roles can switch. The interface of robot A consists of five input and six output variables. It contains a unique location e_R, with jump entry condition true (not represented). Once entered, the robot will react and execute forever, without control exiting. The inputs $lead_B$ and $switch_B$ indicate whether robot B is in the lead mode, or about to switch from follow to lead. The input $obst_A$ indicates if an obstacle is encountered. The component Motor_A, shown in Figure 2, controls the motion of the two wheels based on the signals $left_A$ and $right_A$, which allow the robot to

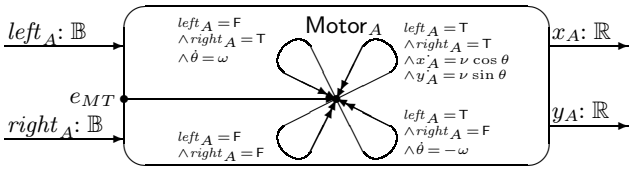

Fig. 2. Motor specification

go straight, halt, or turn in either direction. The outputs x_A and y_A give the position of the robot.

The executions of a component The behavior of a component A is described by a set E_A of finite executions; the treatment of infinite behaviors for the study of liveness issues, such as nonzenoness [Hen96], is deferred for now. An *execution* is either a triple $(a, \boldsymbol{w}, b)$ or a pair $(a, \boldsymbol{w})$ defined by an origin location $a \in L_A$, a nonempty finite sequence $\boldsymbol{w}$ of execution steps and, possibly, a destination location $b \in L_A$. An execution step is either a jump or a flow. A *jump* consists of a source I/O state and a sink I/O state; a *flow* consists of a real duration $\delta \geq 0$ together with a differentiable curve f that maps every real time in the compact interval $[0, \delta]$ to an I/O state. For types other than $\mathbb{R}$, we assume that only constant functions are differentiable. The source of the flow is the I/O state $f(0)$, and the sink is $f(\delta)$. For any two successive execution steps, the sink of the first must coincide with the source of the second. In figures, arrows with double tips denote flows, whereas normal arrows represent jumps.

The set E_A of executions is *prefix-closed*. Indeed, if a component permits a flow of a certain duration, then all restrictions of the flow to shorter durations, including the restriction to duration 0, are also permitted. Every component is *deadlock-free*, in the sense that (1) if the jump entry condition of a location a is satisfiable at an I/O state q, then there is an execution with origin a which starts with a jump with source q, (2) if the flow entry condition of location a is true at q, then there is an execution with origin a which starts with a flow with source q, and (3) every execution that does not end in a destination location can be prolonged by either a destination or a jump. Indeed, the stronger condition of *input-permissiveness* holds, which asserts that a component cannot deadlock no matter how the environment decides to change the inputs, by either jumping or flowing. Prefix-closure, deadlock-freedom, and input-permissiveness are formally defined and proved in the full version of this paper. They are essential properties of every component, because the environment (another component) may decide to interrupt a flow at any time to perform a jump, in which case the component must be prepared to match the environment jump by a local jump.

Atomic components Every component in Masaccio is built from two kinds of atomic components, with discrete and continuous behavior, respectively. An atomic component has an arbitrary number of input and output variables, but only two locations, which serve as origin and destination, respectively, for its executions, all of which contain a single step. For an atomic discrete component, that step is a jump; for an atomic continuous component, a flow. The legal

jumps of an *atomic discrete component* are defined by a jump predicate, which constrains the output values of the sink depending on the source I/O state and on input values of the sink. Such a predicate is typically specified by a difference equation. The legal flows of an *atomic continuous component* are defined by a flow predicate, which constrains the time derivatives of output variables depending on the current I/O state and on the current time derivatives of input variables. Such a predicate is typically specified by a differential equation, as in Figure 2. A flow predicate may also constrain the values of output variables, so that a flow must not go on for any duration that would violate this "invariant" condition. Both jump predicates and flow predicates may allow nondeterminism.

Operations on components *Discrete components* are built from atomic discrete components using the six operations of parallel and serial composition, variable and location renaming, and variable and location hiding, arbitrarily nested. The discrete components conservatively extend Reactive Modules [AH99] by serial composition. *Hybrid components* are built from both discrete and continuous atomic components using the same six operations.

Parallel composition is defined synchronously, as conjunction, with static await dependencies between outputs and inputs preventing circularity. For two components A and B, an execution of the parallel composition $A||B$ starts at a common location in $L_A \cap L_B$. The execution is synchronous in both components: each jump of A must be matched by a concurrent jump of B, and each flow of A must be matched by a concurrent flow of B with the same duration. Control exits the parallel composition when it exits any one of the two components. If the execution of A reaches a destination location, then the concurrent execution of B is preempted and terminated; if B reaches a destination location, then the concurrent execution of A is terminated; if both A and B simultaneously reach destination locations, then the result is nondeterministic. When constructing a parallel composition $A||B$, inputs of A can be identified with outputs of B, and vice versa, by renaming variables. Such identifications are depicted by solid lines in the figures. Similarly, locations of A can be identified with locations of B by renaming locations; these identifications are depicted by dotted lines. We write $A[x := y]$ for the component that results from renaming the variable x in A to y, and $A[a := b]$ for the component that results from renaming the location a in A to b.

In Figure 1, the component Robot_A is the parallel composition of the components $\mathsf{Control}_A$ and Motor_A. Before composition, the two entry locations e_C and e_{MT} are renamed to a common location e_R.

Serial composition and location hiding can be used to achieve the sequencing of components. Serial composition represents disjunctive choice between the executions of two components. For two components A and B, an execution of the serial composition $A+B$ is either an execution of A or an execution of B. Hiding renders a location internal to a component, and inaccessible (invisible) from the outside. The executions of the resulting component are obtained by stringing together at that location any finite number of executions of the original com-

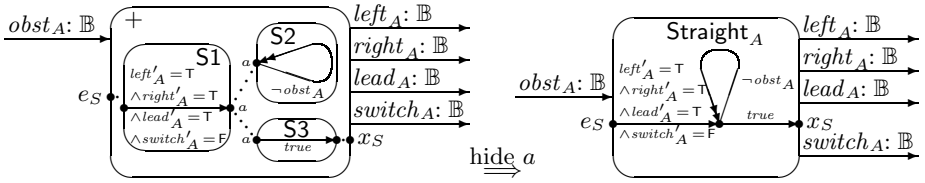

Fig. 3. Serial composition and location hiding

ponent. To avoid internal deadlock, a location a can be hidden only if its jump entry condition is valid, so that it can always take another jump at a. We write $A\backslash a$ for the component that results from hiding a in A.

Figure 3 shows how a sequential component (representing the straight movement of the robot in the lead mode) is obtained by the serial composition of several components, followed by location hiding. Let $\mathsf{Straight}_A = (\mathsf{S1} + \mathsf{S2} + \mathsf{S3})\backslash a$, where $\mathsf{S1}$ and $\mathsf{S3}$ are atomic discrete components, and $\mathsf{S2}$ is obtained from an atomic continuous component by renaming destination location to origin location. The resulting component initializes its output variables by a jump, flows (without output changes) for any amount of time as long as the input $obst_A$ remains false, and nondeterministically exits with a jump. In the same way, any "automaton structure" can be built from individual "edges" (i.e., atomic components) using serial composition, location renaming, and location hiding.

Variable hiding builds an abstract component by turning some outputs of a component into internal state. Hidden variables, however, do not maintain their values from one exit of a component to a subsequent entry, but they are nondeterministically reinitialized upon every entry to the component so as to satisfy the applicable entry condition. We write $A\backslash x$ for the component that results from hiding the output variable x of the component A.

3 Assume-Guarantee Refinement between Components

If component A refines component B, then B can be viewed as a more abstract (permissive) version of A, with some details (constraints) left out in B which are spelled out in A. In particular, in the trace-based semantics of concurrent systems, refinement is taken to be the containment relation on trace sets. If A refines B, then A is a more specific description of system behavior than B in the sense that A may be equivalent to $B||C$ for some parallel context C which constrains the inputs to B. In analogy, in the trace-based semantics of sequential systems, refinement ought to be interpreted as prefix relation on trace sets. If A refines B, then A is a more specific description of system behavior than B in the sense that A may be equivalent to $B + C$ for some serial context C which constrains the continuations of B. Consequently, in Masaccio, if A refines B, then A may specify fewer traces and longer traces than B.

The refinement relation Component A *refines* component B if the following two conditions are satisfied:

1. Every output variable of B is an output variable of A, every input variable of B is an I/O variable of A, and the dependency relation of B is a subset of the dependency relation of A.
2. For every execution $(a, \boldsymbol{w})$ (or $(a, \boldsymbol{w}, b)$, respectively) of A, either $(a, \boldsymbol{w}[V_B])$ (or $(a, \boldsymbol{w}[V_B], b)$, respectively, where $\boldsymbol{w}[V_B]$ is the projection of $\boldsymbol{w}$ to the variables of B) is an execution of B, or there exist a proper, nonempty prefix $\boldsymbol{w'}$ of $\boldsymbol{w}$ and an interface location $c \in L_B$ such that $(a, \boldsymbol{w'}[V_B], c)$ is an execution of B.

Note that the second condition implies that every interface location of A is an interface location of B. Furthermore, by input-permissiveness, if A refines B, then for every location a of A, the jump entry condition of a in A implies the jump entry condition of a in B, and the flow entry condition of a in A implies the flow entry condition of a in B.

Compositionality All six operations on components are compositional.

Theorem 1. *Let A and B be components, let x and y be variables, and let a and b be locations so that the following expressions are all well-defined. If A refines B, then $A||C$ refines $B||C$; and $A + C$ refines $B + C$; and $A[x := y]$ refines $B[x := y]$; and $A[a := b]$ refines $B[a := b]$; and $A\backslash x$ refines $B\backslash x$; and $A\backslash a$ refines $B\backslash a$.*

More generally, define a *context* to be a component expression that can take a component as a parameter. For instance, if $(A + B)||D$ is well-defined, we can regard $C[\cdot] = ([\cdot] + B)||D$ as a context for component A.

Corollary 1. *Let $C[\cdot]$ be a context for both A_1 and A_2. If A_1 refines A_2, then $C[A_1]$, refines $C[A_2]$.*

Assume-guarantee reasoning Our assume-guarantee rule states that for discrete components, if two components can be individually replaced in a context while maintaining refinement, then both can be replaced simultaneously. Therefore, in order to show that a complex component $C[A_1, B_1]$ (the "implementation") refines a simpler component $C[A_2, B_2]$ (the "specification"), it suffices to look at simplified versions of the implementation one at a time. First, we prove that A_1 refines its specification B_1, under the "assumption" B_2; then, we prove that A_2 refines its specification B_2, under the "assumption" B_1. This reasoning is inherently circular. A special case is the assume-guarantee rule for the parallel composition of Reactive Modules [AH99]: take the context $C[\circ, \bullet]$ in the following theorem to be $\circ||\bullet$. The proof relies on the deadlock-freedom and input-permissiveness of components. It also requires that each execution of a serial composition can be uniquely assigned to one of the components. This can be achieved by disjoint entry conditions. We say that the serial composition $A + B$ is *jump-deterministic* if for all common interface locations $a \in L_A \cap L_B$, the conjunction $\psi_A^{jump}(a) \wedge \psi_B^{jump}(a)$ is unsatisfiable, and *flow-deterministic* if $\psi_A^{flow}(a) \wedge \psi_B^{flow}(a)$ is unsatisfiable for all $a \in L_A \cap L_B$. The serial composition $A + B$ is *deterministic* if it is both jump-deterministic and flow-deterministic.

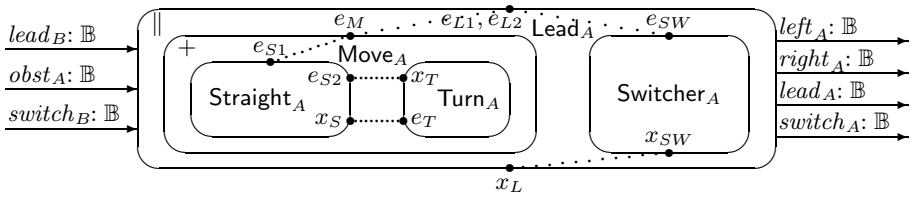

Fig. 4. Component Lead_A

For hybrid modules, we need to break the circularity of the rule, by relaxing one assumption, say, B_2, to allow arbitrary flows at all hidden locations. We write $rlax(B_2)$ for the component that results from B_2 by (1) replacing every flow predicate in B_2 by *true*, and (2) serially composing every hidden location a of B_2 which is not the origin location of any flow, with an atomic continuous component that permits all flows from origin a to destination a.

Theorem 2. *Let $C[\circ, \bullet]$ be a context whose arguments are not in the scope of any variable or location hiding. Suppose that all input variables of $C[A_2, B_2]$ are variables of $C[A_1, B_1]$, and that within $C[A_2, B_2]$ the context arguments are not within the scope of any nondeterministic serial composition. If $C[A_1, rlax(B_2)]$ refines $C[A_2, rlax(B_2)]$, and $C[A_2, B_1]$ refines $C[A_2, B_2]$, then $C[A_1, B_1]$ refines $C[A_2, B_2]$.*

Linear components If all flows are specified by linear differential equations, and no degenerate flows of 0 duration can be enforced, then the existence of unique solutions allows us to strengthen the assume-guarantee rule. In this case, we can make circular assumptions about the flows. An *open linear condition* on a set V of real-valued variables is a conjunction of boolean variables and strict ($<$ or $>$) comparisons between linear combinations of the variables in V. Consider a flow action F (consult the appendix for a definition). The atomic continuous component $A(F)$ is *linear* if (1) all variables in $V_{A(F)}$ have the type $\mathbb{R}$, and (2) the flow predicate φ_F^{flow} has the form $\alpha(X_F) \wedge (\dot{Z}_F = \beta(X_F, \dot{Y}_F))$, where α is an open linear condition, called *invariant*, on the source variables X_F, and β is a set of linear combinations, one for the derivative $\dot{z} \in \dot{Z}_F$ of each controlled flow variable, of the source variables X_F and the derivatives $\dot{Y}_F$ of the uncontrolled flow variables. A component is *linear* if (1) all its atomic continuous components are linear, and (2) all its serial compositions are flow-deterministic. Let $rlax'$ be defined like $rlax$, with the difference that only the invariants rather than the flow predicates are replaced with *true*.

Theorem 3. *Let $C[\circ, \bullet]$ be a context whose arguments are not in the scope of any variable or location hiding. Suppose that $C[A_1, B_1]$ and $C[A_2, B_2]$ are linear components, that all input variables of $C[A_2, B_2]$ are variables of $C[A_1, B_1]$, and that within $C[A_2, B_2]$ the context arguments are not within the scope of any nondeterministic serial composition. If $C[A_1, rlax'(B_2)]$ refines $C[A_2, rlax'(B_2)]$, and $C[A_2, B_1]$ refines $C[A_2, B_2]$, then $C[A_1, B_1]$ refines $C[A_2, B_2]$.*

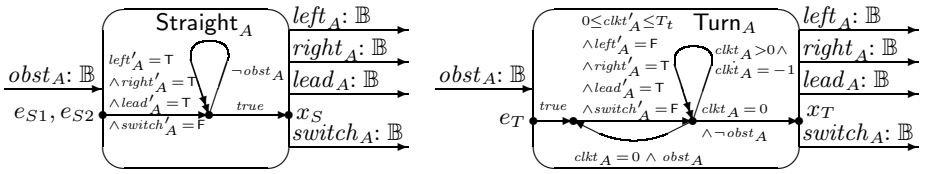

Fig. 5. Components $\mathsf{Straight}_A$ and Turn_A

4 A Two-Robot Example

We continue the presentation of the two-robot system whose overall view was given in Section 2. Robot A (Figure 1) starts out as the leader. After a while it may move from Lead_A to Follow_A, as indicated by the dotted line connecting location x_L (with an unsatisfiable entry condition, which is not shown) and location e_F. It may then move back to lead mode (line x_F–e_{L2}). Robot B has the same structure, except that it starts out in follow mode. Within the subcomponent Move_A (Figure 4), the robot can execute in $\mathsf{Straight}_A$ arbitrarily long while there is no obstacle. Upon sensing an obstacle, control is passed to the component Turn_A, which commands the robot to rotate for an amount of time given by timer variable $clkt_A$. Control then returns to the component $\mathsf{Straight}_A$. The sequence of straight moves and turns continues until robot B switches to leading status. This event is modeled by the boolean signal $switch_B$, which is monitored by the component $\mathsf{Switcher}_A$. We require the switcher unit to preempt execution of the lead mode within a specified amount of time T_{sw} after the other robot has signaled its intention to lead. Once Lead_A is exited, control enters the component Follow_A, which samples the values of $left_B$ and $right_B$ and drives its own motor signals $left_A$ and $right_A$. The robot may stay in the follow mode arbitrarily long, provided that $obst_A$ is false. At any time it may also issue the signal $switch_A$, exit the component Follow_A and switch back to lead mode.

We now present a robot implementation that contains a modified component Lead^I_A, which does not continuously observe the *switch* signal (Figure 7). Instead, the implementation samples the leading indicators of both robots with a period T_{ed}, as measured by the global clock clk. If both robots are leading, a correction is made by the component $\mathsf{Errordetect}_A$. The new state depends on the last sampled values of the leading signals: the robot that had been leading before now switches to follow mode.

We wish to show that when composed together, two robot implementations refine the parallel composition of two robot specifications, provided that $T_{ed} < T_{sw}$. The specification of robot A is $\mathsf{Control}_A||\mathsf{Motor}_A$, and the implementation of robot A is $\mathsf{Control}^I_A||\mathsf{Motor}_A$, where $\mathsf{Control}_A = (\mathsf{Lead}_A + \mathsf{Follow}_A)\backslash e_{L2}\backslash e_F$ and $\mathsf{Control}^I_A = (\mathsf{Lead}^I_A + \mathsf{Follow}_A)\backslash e_{L2}\backslash e_F$. Robot B is specified and implemented symmetrically. Denoting the parallel composition with the motor by the context $C_A[\cdot] = \cdot||\mathsf{Motor}_A$, and similarly for C_B, we wish to prove that

$$C_A[\mathsf{Control}^I_A]||C_B[\mathsf{Control}^I_B] \;\text{ refines }\; C_A[\mathsf{Control}_A]||C_B[\mathsf{Control}_B].$$

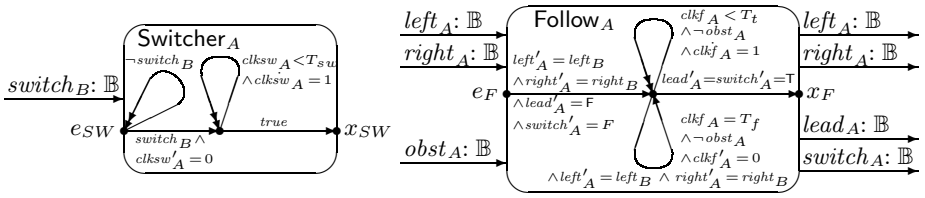

Fig. 6. Components $\mathsf{Switcher}_A$ and Follow_A

Note that $C_A[\mathsf{Control}^I_A]$ does not refine $C_A[\mathsf{Control}_A]$, because a robot implementation meets the specification only when composed with a symmetric robot. This is where assume-guarantee reasoning helps. All continuous components in the system are linear. Hence by Theorem 3, it suffices to discharge the simpler assertions

$$C_A[\mathsf{Control}^I_A]||C_B[\mathsf{Control}'_B] \text{ refines } C_A[\mathsf{Control}_A]||C_B[\mathsf{Control}'_B]$$
$$C_A[\mathsf{Control}_A]||C_B[\mathsf{Control}^I_B] \text{ refines } C_A[\mathsf{Control}_A]||C_B[\mathsf{Control}_B],$$

where $\mathsf{Control}'_B = rlax'(\mathsf{Control}_B)$. We simplify further using compositionality (Theorem 1), and are left to prove that

$$\mathsf{Control}^I_A \parallel \mathsf{Control}'_B \text{ refines } \mathsf{Control}_A \parallel \mathsf{Control}'_B$$
$$\mathsf{Control}_A \parallel \mathsf{Control}^I_B \text{ refines } \mathsf{Control}_A \parallel \mathsf{Control}_B,$$

two proof obligations that involve simpler components than the original one. The power of the assume-guarantee rules of Theorems 2 and 3 stems from the fact that they can be applied to components arbitrarily deep in the design hierarchy, creating proof obligations which have smaller differences between the two components which are supposed to refine each other.

Acknowledgments We thank Rajeev Alur, Radu Grosu, and Edward Lee for many stimulating discussions.

References

[ACH+95] R. Alur, C. Courcoubetis, N. Halbwachs, T.A. Henzinger, P.-H. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. The algorithmic analysis of hybrid systems. *Theoretical Computer Science*, 138:3–34, 1995.

[AG00] R. Alur and R. Grosu. Modular refinement of hierarchic reactive machines. In *Principles of Programming Languages*, pp. 390–402, ACM Press, 2000.

[AGH+00] R. Alur, R. Grosu, Y. Hur, V. Kumar, and I. Lee. Modular specification of hybrid systems in Charon. In *Hybrid Systems: Computation and Control*, LNCS 1790, pp. 130–144, Springer-Verlag, 2000.

[AH97] R. Alur and T.A. Henzinger. Modularity for timed and hybrid systems. In *Concurrency Theory*, LNCS 1243, pp. 74–88, Springer-Verlag, 1997.

[AH99] R. Alur and T.A. Henzinger. Reactive modules. *Formal Methods in System Design*, 15:7–48, 1999.

[AL95] M. Abadi and L. Lamport. Conjoining specifications. *ACM Transactions on Programming Languages and Systems*, 17:507–534, 1995.

[BRJ98] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, 1998.

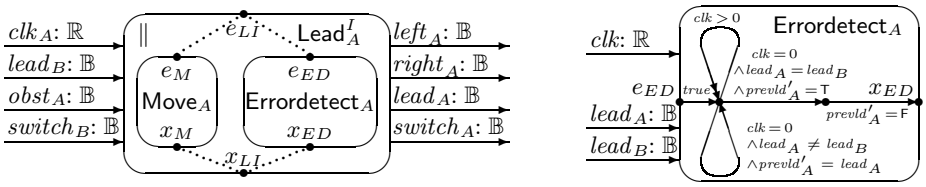

Fig. 7. Components Lead_A^I and $\mathsf{Errordetect}_A$

[DGH+99] J. Davis, M. Goel, C. Hylands, B. Kienhuis, E.A. Lee, J. Liu, X. Liu, L. Muliadi, S. Neuendorffer, J. Reekie, N. Smyth, J. Tsay, and Y. Xiong. Overview of the Ptolemy project. Tech. Rep. UCB/ERL M99/37, University of California, Berkeley, 1999.

[DGV97] A. Deshpande, A. Göllü, and P. Varaiya. Shift: A formalism and a programming language for dynamic networks of hybrid automata. In *Hybrid Systems*, LNCS 1273, pp. 113–134, Springer-Verlag, 1997.

[Har87] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.

[Hen96] T.A. Henzinger, The theory of hybrid automata. In *Logic in Computer Science*, pp. 278–292, IEEE Computer Society Press, 1996.

[Hen00] T.A. Henzinger. Masaccio: A formal model for embedded components. In *Theoretical Computer Science*, LNCS 1872, pp. 549–563, Springer Verlag, 2000.

[LSVW96] N.A. Lynch, R. Segala, F. Vaandrager, and H.B. Weinberg. Hybrid I/O Automata. In *Hybrid Systems*, LNCS 1066, pp. 496–510, Springer-Verlag, 1996.

[McM97] K.L. McMillan. A compositional rule for hardware design refinement. In *Computer-aided Verification*, LNCS 1254, pp. 24–35, Springer-Verlag, 1997.

[MC81] J. Misra and K.M. Chandy. Proofs of networks of processes. *IEEE Transactions on Software Engineering*, 7:417–426, 1981.

[TAKB96] S. Tasiran, R. Alur, R.P. Kurshan, and R.K. Brayton. Verifying abstractions of timed systems. In *Concurrency Theory*, LNCS 1119, pp. 546–562, Springer-Verlag, 1996.

[US94] A.C. Uselton and S.A. Smolka. A compositional semantics for Statecharts using labeled transition systems. In *Concurrency Theory*, LNCS 836, pp. 2–17, Springer-Verlag, 1994.

Appendix: Formal Definition of Masaccio

Let V be a set of typed variables. For a variable $x \in V$, denote by x' its primed version, and denote by $\dot{x}$ its dotted version. The type of x' is the same as the type of x. The type of $\dot{x}$ is $\mathbb{R}$ if the type of x is $\mathbb{R}$, and $\{0\}$ otherwise. This is because on types other than $\mathbb{R}$, we assume that only the constant functions are differentiable. Let $V' = \{x' \mid x \in V\}$ be the set of primed versions of the variables in V, and let $\dot{V} = \{\dot{x} \mid x \in V\}$ be the set of dotted versions of the variables in V. Let $[V]$ be the set of type-conforming value assignments to the variables in V: if $x \in V$ and $q \in [V]$, let $q(x)$ be the value assigned by q to x.

The interface of a component The *interface of a component* A consists of:

- A finite set V_A^i of typed *input variables*.

- A finite set V_A^o of typed *output variables*, such that $V_A^i \cap V_A^o = \emptyset$. Let $V_A = V_A^i \cup V_A^o$ be the set of *I/O variables*. The value assignments in $[V_A]$ are called *I/O states*.
- An *dependency relation* $\prec_A \subseteq V_A \times V_A^o$ between I/O variables and output variables, such that the transitive closure $\prec_A^*$ is asymmetric. A set $U \subseteq V_A$ of I/O variables is *dependency-closed* if $x \prec_A y$ and $y \in U$ implies $x \in U$.
- A finite set L_A of *interface locations*.
- For each location $a \in L_A$, a predicate $\psi_A^{jump}(a)$ on the variables in $V_A \cup V_A^{i'}$, called *jump entry condition*, and a predicate $\psi_A^{flow}(a)$ on the variables in V_A, called *flow entry condition*.

The executions of a component A *jump* of a component A is a pair $(p, q) \in [V_A]^2$ of I/O states. The I/O state p is the *source* of the jump, and q is the *sink*. A *flow* of A is a pair (δ, f) consisting of a nonnegative real $\delta \in \mathbb{R}_{\geq 0}$, and a function $f\colon \mathbb{R} \to [V_A]$ from the reals to I/O states which is differentiable, with time derivative f', on the compact interval $[0, \delta] \subset \mathbb{R}$. The real δ is the *duration* of the flow, the I/O state $f(0)$ is the *source*, and $f(\delta)$ is the *sink*. A *step* of A is either a jump or a flow of A. The step w is *successive* to the step v if the sink of v is equal to the source of w. An *execution* of A is either a pair $(a, \boldsymbol{w})$ or a triple $(a, \boldsymbol{w}, b)$, where $a, b \in L_A$ are interface locations, and $\boldsymbol{w} = w_0 \cdots w_n$ is a finite, nonempty sequence of steps of A such that (1) every step w_i, for $1 \leq i \leq n$, is successive to the preceding step w_{i-1}, and (2) the first step w_0 satisfies the entry conditions of location a: if $w_0 = (p, q)$ is a jump, then $\psi_A^{jump}(a)$ is true if each I/O variable $x \in V_A$ is assigned the value $p(x)$, and each primed input variable $y' \in V_A^{i'}$ is assigned the value $q(y)$; if $w_0 = (\delta, f)$ is a flow, then $\psi_A^{flow}(a)$ is true if each I/O variable $x \in V_A$ is assigned the value $f(0)(x)$. The location a is the *origin* of the execution, the sequence $\boldsymbol{w}$ is the *trace*, and the location b (when present) is the *destination*. Given a trace $\boldsymbol{w}$ and a set $U \subseteq V_A$ of I/O variables, we write $\boldsymbol{w}[U]$ for the projection of $\boldsymbol{w}$ to the variables in U,

Atomic discrete components An *atomic discrete component* is specified by a jump action. A *jump action* J consists of a finite set X_J of *source variables*, a finite set Y_J of *uncontrolled sink variables*, a finite set Z_J of *controlled sink variables* disjoint from Y_J, and a predicate φ_J^{jump} on the variables in $X_J \cup Y_J' \cup Z_J'$, called *jump predicate*. The jump action J specifies the component $A(J)$. The *interface of the component* $A(J)$ is defined as follows:

- $V_{A(J)}^i = (X_J \setminus Z_J) \cup Y_J$.
- $V_{A(J)}^o = Z_J$.
- $y \prec_{A(J)} z$ iff $y \in Y_J$ and $z \in Z_J$.
- $L_{A(J)} = \{\mathit{from}, \mathit{to}\}$.
- $\psi_{A(J)}^{jump}(\mathit{from}) = (\exists Z_J')\, \varphi_J^{jump}$ and $\psi_{A(J)}^{flow}(\mathit{from}) = \mathit{false}$.
- $\psi_{A(J)}^{jump}(\mathit{to}) = \psi_{A(J)}^{flow}(\mathit{to}) = \mathit{false}$.

The *executions of the component* $A(J)$ are defined as follows. The pair $(a, \boldsymbol{w})$ is an execution of $A(J)$ iff $a = \mathit{from}$ and the trace $\boldsymbol{w}$ consists of a single jump (p, q) such that φ_J^{jump} is true if each source variable $x \in X_J$ is assigned the value $p(x)$, and each primed sink variable $y' \in Y_J' \cup Z_J'$ is assigned the value $q(y)$. The triple $(a, \boldsymbol{w}, b)$ is an execution of $A(J)$ iff the pair $(a, \boldsymbol{w})$ is an execution of $A(J)$, and $b = \mathit{to}$.

Atomic continuous components An *atomic continuous component* is specified by a flow action. A *flow action* F consists of a finite set X_F of *source variables*, a finite set Y_F of *uncontrolled flow variables*, a finite set Z_F of *controlled flow variables* disjoint

from Y_F, and a predicate φ_F^{flow} on the variables in $X_F \cup \dot{Y}_F \cup \dot{Z}_F$, called *flow predicate*. The flow action F specifies the component $A(F)$. The *interface of the component* $A(F)$ is defined as follows:

- $V^i_{A(F)} = (X_F \setminus Z_F) \cup Y_F$.
- $V^o_{A(F)} = Z_F$.
- $y \prec_{A(F)} z$ iff $y \in Y_F$ and $z \in Z_F$.
- $L_{A(F)} = \{from, to\}$.
- $\psi^{jump}_{A(F)}(from) = false$ and $\psi^{flow}_{A(F)}(from) = (\exists \dot{Y}_F, \dot{Z}_F)\, \varphi_F^{flow}$.
- $\psi^{jump}_{A(F)}(to) = \psi^{flow}_{A(F)}(to) = false$.

The *executions of the component* $A(F)$ are defined as follows. The pair $(a, \boldsymbol{w})$ is an execution of $A(F)$ iff $a = from$ and the trace w consists of a single flow (δ, f) such that the following holds: if $\delta = 0$, then $(\exists \dot{Y}_F, \dot{Z}_F)\, \varphi_F^{flow}$ is true if each source variable $x \in X_F$ is assigned the value $f(0)(x)$; if $\delta > 0$, then for all $\varepsilon \in [0, \delta]$, the flow predicate φ_F^{flow} is true if each source variable $x \in X_F$ is assigned the value $f(\varepsilon)(x)$, and each dotted flow variable $\dot{y} \in \dot{Y}_F \cup \dot{Z}_F$ is assigned the value $f'(\varepsilon)(y)$. The triple $(a, \boldsymbol{w}, b)$ is an execution of $A(F)$ iff the pair $(a, \boldsymbol{w})$ is an execution of $A(F)$, and $b = to$.

Parallel composition Two components A and B *can be composed in parallel* if their interfaces satisfy the following conditions:

- $V^o_A \cap V^o_B = \emptyset$.
- There are no two variables $x \in V^o_A$ and $y \in V^o_B$ such that both $x \prec^*_B y$ and $y \prec^*_A x$.
- For all $a \in L_A$, if $\psi^{jump}_A(a)$ or $\psi^{flow}_A(a)$ is satisfiable, then $a \in L_B$. For all $a \in L_B$, if $\psi^{jump}_B(a)$ or $\psi^{flow}_B(a)$ is satisfiable, then $a \in L_A$. For all $a \in L_A \cap L_B$, the projections of the entry conditions of a in A and B to the common variables are equivalent: $(\exists V_A \setminus V_B)(\exists V^{i'}_A \setminus V^{i'}_B)\, \psi^{jump}_A(a)$ is equivalent to $(\exists V_B \setminus V_A)(\exists V^{i'}_B \setminus V^{i'}_A)\, \psi^{jump}_B(a)$, and $(\exists V_A \setminus V_B)\, \psi^{flow}_A(a)$ is equivalent to $(\exists V_B \setminus V_A)\, \psi^{flow}_B(a)$.

The *interface* of $A||B$ is defined from the interfaces of A and B:

- $V^i_{A||B} = (V^i_A \setminus V^o_B) \cup (V^i_B \setminus V^o_A)$.
- $V^o_{A||B} = V^o_A \cup V^o_B$.
- $\prec_{A||B} = \prec_A \cup \prec_B$.
- $L_{A||B} = L_A \cup L_B$.
- If $a \in L_A \cap L_B$, then $\psi^{jump}_{A||B}(a) = \psi^{jump}_A(a) \wedge \psi^{jump}_B(a)$ and $\psi^{flow}_{A||B}(a) = \psi^{flow}_A(a) \wedge \psi^{flow}_B(a)$. If $a \in L_A \setminus L_B$ or $a \in L_B \setminus L_A$, then $\psi^{jump}_{A||B}(a) = \psi^{flow}_{A||B}(a) = false$.

The *executions* of $A||B$ are defined from the executions of A and B. The pair $(a, \boldsymbol{w})$ is an execution of $A||B$ iff $(a, \boldsymbol{w}[V_A])$ is an execution of A and $(a, \boldsymbol{w}[V_B])$ is an execution of B. The triple $(a, \boldsymbol{w}, b)$ is an execution of $A||B$ iff either $(a, \boldsymbol{w}[V_A], b)$ is an execution of A and $(a, \boldsymbol{w}[V_B])$ is an execution of B, or $(a, \boldsymbol{w}[V_B], b)$ is an execution of B and $(a, \boldsymbol{w}[V_A])$ is an execution of A.

Serial composition Two components A and B *can be composed in series* if $V^o_A = V^o_B$. The *interface* of $A + B$ is defined from the interfaces of A and B:

- $V^i_{A+B} = V^i_A \cup V^i_B$.
- $V^o_{A+B} = V^o_A = V^o_B$.
- $\prec_{A+B} = \prec_A \cup \prec_B$.
- $L_{A+B} = L_A \cup L_B$.

– If $a \in L_A \cap L_B$, then $\psi^{jump}_{A+B}(a) = \psi^{jump}_A(a) \vee \psi^{jump}_B(a)$ and $\psi^{flow}_{A+B}(a) = \psi^{flow}_A(a) \vee \psi^{flow}_B(a)$. If $a \in L_A \setminus L_B$, then $\psi^{jump}_{A+B}(a) = \psi^{jump}_A(a)$ and $\psi^{flow}_{A+B}(a) = \psi^{flow}_A(a)$. If $a \in L_B \setminus L_A$, then $\psi^{jump}_{A+B}(a) = \psi^{jump}_B(a)$ and $\psi^{flow}_{A+B}(a) = \psi^{flow}_B(a)$.

The *executions* of $A + B$ are defined from the executions of A and B. The pair $(a, \boldsymbol{w})$ is an execution of $A + B$ iff either $(a, \boldsymbol{w}[V_A])$ is an execution of A, or $(a, \boldsymbol{w}[V_B])$ is an execution of B. The triple $(a, \boldsymbol{w}, b)$ is an execution of $A + B$ iff either $(a, \boldsymbol{w}[V_A], b)$ is an execution of A, or $(a, \boldsymbol{w}[V_B], b)$ is an execution of B.

Variable renaming The variable $x \in V_A$ *can be renamed to* y in component A if y has the same type as x, and either y is not an I/O variable of A, or x and y are both input variables; that is, if $y \in V_A$, then $x, y \in V^i_A$. The *interface of the component* $A[x := y]$ is defined from the interface of A. If $x \in V^i_A$, then $V^i_{A[x:=y]} = (V^i_A \setminus \{x\}) \cup \{y\}$ and $V^o_{A[x:=y]} = V^o_A$; if $x \in V^o_A$, then $V^i_{A[x:=y]} = V^i_A$ and $V^o_{A[x:=y]} = (V^o_A \setminus \{x\}) \cup \{y\}$. In either case, let $L_{A[x:=y]} = L_A$, and let $\prec_{A[x:=y]}$ and $\psi^{jump}_{A[x:=y]}$ and $\psi^{flow}_{A[x:=y]}$ result from renaming x to y, and x' to y', in $\prec_A$ and ψ^{jump}_A and ψ^{flow}_A, respectively. The *executions of the component* $A[x := y]$ result from renaming x to y in the traces of the executions of A.

Location renaming The interface location $a \in L_A$ *can be renamed to* b in component A if either b is not an interface location of A, or the entry conditions of a and b are disjoint; that is, if $b \in L_A$, then both $\psi^{jump}_A(a) \wedge \psi^{jump}_A(b)$ and $\psi^{flow}_A(a) \wedge \psi^{flow}_A(b)$ are unsatisfiable. The *interface of the component* $A[a := b]$ is defined from the interface of A: let $V^i_{A[a:=b]} = V^i_A$; let $V^o_{A[a:=b]} = V^o_A$; let $\prec_{A[a:=b]} = \prec_A$; let $L_{A[a:=b]} = (L_A \setminus \{a\}) \cup \{b\}$; let $\psi^{jump}_{A[a:=b]}(b) = \psi^{jump}_A(a) \vee \psi^{jump}_A(b)$ and $\psi^{flow}_{A[a:=b]}(b) = \psi^{flow}_A(a) \vee \psi^{flow}_A(b)$ if $b \in L_A$, let $\psi^{jump}_{A[a:=b]}(b) = \psi^{jump}_A(a)$ and $\psi^{flow}_{A[a:=b]}(b) = \psi^{flow}_A(a)$ if $b \notin L_A$, and let $\psi^{jump}_{A[a:=b]}(c) = \psi^{jump}_A(c)$ and $\psi^{flow}_{A[a:=b]}(c) = \psi^{flow}_A(c)$ for all locations $c \in L_A \setminus \{a, b\}$. The *executions of the component* $A[a := b]$ result from renaming a to b in the origins and destinations of the executions of A.

Variable hiding The variable $x \in V_A$ *can be hidden* in the component A if $x \in V^o_A$. The *interface of the component* $A\backslash x$ is defined from the interface of A: let $V^i_{A\backslash x} = V^i_A$; let $V^o_{A\backslash x} = V^o_A \setminus \{x\}$; let $\prec_{A\backslash x}$ be the intersection of the transitive closure $\prec^*_A$ with $V_{A\backslash x} \times V^o_{A\backslash x}$; let $L_{A\backslash x} = L_A$; let $\psi^{jump}_{A\backslash x}(a) = (\exists x)\, \psi^{jump}_A(a)$ and $\psi^{flow}_{A\backslash x}(a) = (\exists x)\, \psi^{flow}_A(a)$ for all locations $a \in L_A$. The *executions of the component* $A\backslash x$ are defined from the executions of A. The pair $(a, \boldsymbol{w})$ is an execution of $A\backslash x$ iff $(a, \boldsymbol{w}[V_{A\backslash x}])$ is an execution of A. The triple $(a, \boldsymbol{w}, b)$ is an execution of $A\backslash x$ iff $(a, \boldsymbol{w}[V_{A\backslash x}], b)$ is an execution of A.

Location hiding The interface location $c \in L_A$ *can be hidden* in the component A if the jump entry condition $\psi^{jump}_A(c)$ is equivalent to *true*. The *interface of the component* $A\backslash c$ is defined from the interface of A: let $V^i_{A\backslash c} = V^i_A$; let $V^o_{A\backslash c} = V^o_A$; let $\prec_{A\backslash c} = \prec_A$; let $L_{A\backslash c} = L_A \setminus \{c\}$; let $\psi^{jump}_{A\backslash c}(a) = \psi^{jump}_A(a)$ and $\psi^{flow}_{A\backslash c}(a) = \psi^{flow}_A(a)$ for all locations $a \in L_{A\backslash c}$. The *executions of the component* $A\backslash c$ are defined from the executions of A. The pair $(a, \boldsymbol{w})$ is an execution of $A\backslash c$ iff $c \neq a$ and either $(a, \boldsymbol{w})$ is an execution of A, or there is a finite sequence $\boldsymbol{w}_1, \ldots, \boldsymbol{w}_n$ of traces, $n \geq 2$, such that $\boldsymbol{w} = \boldsymbol{w}_1 \cdots \boldsymbol{w}_n$ and the following are executions of A: the triple $(a, \boldsymbol{w}_1, c)$, the triples $(c, \boldsymbol{w}_i, c)$ for $1 < i < n$, and the pair $(c, \boldsymbol{w}_n)$. The triple $(a, \boldsymbol{w}, b)$ is an execution of $A\backslash c$ iff $c \notin \{a, b\}$ and $(a, \boldsymbol{w}, b)$ is an execution of A, or there is a finite sequence $\boldsymbol{w}_1, \ldots, \boldsymbol{w}_n$ of traces, $n \geq 2$, such that $\boldsymbol{w} = \boldsymbol{w}_1 \cdots \boldsymbol{w}_n$ and the following are executions of A: the triple $(a, \boldsymbol{w}_1, c)$, the triples $(c, \boldsymbol{w}_i, c)$ for $1 < i < n$, and the triple $(c, \boldsymbol{w}_n, b)$.