Compositional Verification of Real-Time Applications

Jozef Hooman

Dept. of Computing Science
Eindhoven University of Technology
P.O. Box 513, 5600 MB Eindhoven, The Netherlands
e-mail: hooman@win.tue.nl
http://www.win.tue.nl/win/cs/tt/hooman/

Abstract. To support top-down design of distributed real-time systems, a framework of mixed terms has been incorporated in the verification system PVS. Programs and assertional specifications are treated in a uniform way. We focus on the timed behaviour of parallel composition and hiding, presenting several alternatives for the definition of a denotational semantics. This forms the basis of compositional proof rules for parallel composition and hiding. The formalism is applied to an example of a hybrid system, which also serves to illustrate our ideas on platform-independent programming.

1 Introduction

The aim of this work is to devise a formal framework for the top-down design of distributed real-time systems. Important ingredient of such a framework is a set of compositional proof rules for the programming constructs. That is, it should be possible to reason with the specifications of components without knowing their implementation [dR85,HdR86].

In Sect. 1.1 we introduce the framework of mixed terms. To obtain mechanized support we use the verification system PVS, presented in Sect. 1.2. Section 1.3 addresses applications and the topic of this paper. The structure of the rest of the paper can be found in Sect. 1.4.

1.1 Mixed Terms

Our specifications are based on assertions, i.e., logical formulae that express desired properties of (part of) a system. To be able to formalize intermediate stages during the top-down design of a system, we aim at a framework where specifications and programming constructs can be mixed freely. This is inspired by similar work on untimed systems [Old85,Old91,Zwi89] and related to recent work on timed systems [SO98].

Example 1. As a simple example, consider a top-level specification $TLSpec$, including timing constraints. Suppose we decide to implement this as the parallel composition of two components specified by $SpecComp_1$ and $SpecComp_2$. This refinement step is denoted by $(SpecComp_1 \parallel SpecComp_2) \Rrightarrow TLSpec$. It should be justified by a compositional rule for parallel composition. Often a set of internal events, say $IntEvents$, is introduced to synchronize or communicate between the two components. These internal events can be encapsulated by a hiding construct, denoted by "$-$". By means of a compositional hiding rule we then show $((SpecComp_1 \parallel SpecComp_2) - IntEvents) \Rrightarrow TLSpec$.
Next each component can be developed in isolation. For instance,
$SpecComp_{11}$; $SpecComp_{12} \Rrightarrow SpecComp_1$ and
$SpecComp_{21} \parallel SpecComp_{22} \Rrightarrow SpecComp_2$.
Then, by means of transitivity of $\Rrightarrow$ and so-called monotonicity rules for parallel composition and hiding, this leads to
$(((SpecComp_{11} \; ; \; SpecComp_{12}) \parallel$
$(SpecComp_{21} \parallel SpecComp_{22})) - IntEvents) \Rrightarrow TLSpec$.
Again the specifications can be refined further, leading to a (real-time) program for each software component of the system.

Traditionally, real-time programs are based on constructs such as delays, time-outs, periods, and priorities. Such a program usually depends heavily on the underlying platform, the scheduling policy, and also on the other components of the system (e.g., also the priority of totally unrelated parallel components is important). Consequently, program verification requires many assumptions about the execution platform and compositionality is difficult to achieve.

Our approach aims at a platform-independent program activity, postponing platform considerations as long as possible. This is achieved by extending conventional (untimed) programming languages with so-called timing annotations that only specify the relevant timing constraints [HvR97]. As a separate activity, these platform-independent programs are then scheduled on a particular execution platform.

1.2 Mechanized Proof Support

To obtain mechanized support for our formal framework, we use the verification system PVS (Prototype Verification System) [ORS92,ORSvH95]. Our mixed framework has been formulated in the language of PVS, a typed higher-order logic. Type-checking might generate proof obligations (Type Check Conditions), requiring a proof that expressions indeed have the proper type. In general, properties can be verified by means of the interactive proof checker of PVS.

PVS provides not only convenient support during the specification and verification of applications, but it is equally useful for the development of formal theory. The use of so-called putative theorems, expressing properties that ought to hold, also frequently reveals errors during theoretical studies. For instance, in the work described here, several errors have been detected by the formulation of alternative definitions and attempts to prove equivalence.

1.3 Applications and Topic of this Paper

In previous work, our assertional framework has been used to verify several protocols with PVS, such as part of the ACCESS.bus protocol [Hoo95] and a membership protocol with local clocks and a dynamically changing network [Hoo97]. Another class of applications concerns the design of hybrid systems, containing both discrete and continuous components. For instance, a steam boiler control system has been specified and verified [VH96].

These applications are based on a formalization of the framework in PVS as described in [Hoo94]. There the proof rules for sequential programming constructs have been proved sound, but the rule for parallel composition was stated as an axiom, based on a manual soundness proof. Main topic of the current paper is to formalize this soundness proof and to investigate several alternatives for the formalization of parallel composition. Additionally we investigate the hiding construct and prove soundness of the required monotonicity rules.

We aim at a framework which allows reasoning about local clocks and hybrid systems; it should be possible to deal with both discrete and continuous time. Hence the framework developed here will be parameterized by a time domain and can be instantiated with various notions of time.

Focusing on parallel composition and hiding, we only describe the externally visible behaviour of a component in terms of the events that occur at any point of time. This clarifies the exposition, leaving extensions of the approach to deal with a local state along the lines of [Hoo94] to future work.

1.4 Overview

The basic semantic primitives are defined in Sect. 2. Several equivalent formulations for the semantics of parallel composition can be found in Sect. 3. Moreover, we prove the soundness of a compositional rule for parallel composition. An alternative framework, aiming at the definition of parallel composition by intersection is presented in Section 4. We show that the two approaches are isomorphic. Semantics and proof rules for the hiding construct are given in Sect. 5, again investigating both approaches. To illustrate the use of the framework and our ideas on platform-independent design, we consider in Sect. 6 a simple example of a hybrid system. Section 7 contains concluding remarks.

2 Semantic Primitives

The PVS specification language allows us to structure the framework into a number of parameterized theories. Theory TimePrim contains a few simple notations to express timing properties. The time domain `Time` is a parameter of this theory, which makes it possible to instantiate it for concrete examples with, e.g., a discrete or a continuous notion of time. The semantic framework and the proof rules presented here are independent of this choice. We only assume given a strict order $<$ and a partial order $\leq$ on this domain, assuming the usual relation between these two, as expressed by the ASSUMING clause below. Importing theory TimePrim with a particular time domain then leads to a so-called Type Check Condition which requires a proof that the particular orders indeed satisfy the formula named leq_less. Of course, we could use only one of the orders as a parameter and define the other in terms of it, but the version presented here allows us to import the theory with, for instance, the real numbers and then make optimal use of the built-in decision procedures of PVS for the reals and the corresponding relations.

We use three dots (...) to indicate that some obvious details are omitted. Observe that TimePrim imports theory Connectives which is presented below.

$$
\begin{array}{l}
\text{TimePrim}[\text{Time}: \text{TYPE}, <\ : (\text{strict\_order?}[\text{Time}]),\\
\qquad\qquad\qquad\qquad\quad \leq\ : (\text{partial\_order?}[\text{Time}])] : \text{THEORY}\\
\quad \text{BEGIN}\\
\quad \text{ASSUMING}\\
\quad\ \ \text{leq\_less}: \text{ASSUMPTION}\ \forall\ (t_1, t_2: \text{Time}): t_1 \leq t_2 \Leftrightarrow t_1 < t_2 \lor t_1 = t_2\\
\quad \text{ENDASSUMING}\\
\quad t, t_0, t_1 : \text{VAR Time}\\
\\
\quad [t_0, t_1] : \text{setof}[\text{Time}] = \{t \mid t_0 \leq t \land t \leq t_1\}\\
\quad [t_0, t_1) : \text{setof}[\text{Time}] = \{t \mid t_0 \leq t \land t < t_1\}\\
\quad \ldots\\
\quad P : \text{VAR pred}[\text{Time}] \qquad \%\ \text{pred}[\text{Time}] = [\text{Time} \to \text{bool}]\\
\quad I : \text{VAR setof}[\text{Time}] \qquad \%\ \text{setof}[\text{Time}] = [\text{Time} \to \text{bool}]\\
\\
\quad P\ \mathbf{in}\ I : \text{bool} = \exists\ t : t \in I \land P(t)\\
\\
\quad P\ \mathbf{during}\ I : \text{bool} = \forall\ t : t \in I \Rightarrow P(t)\\
\\
\quad \text{IMPORTING Connectives}[\text{Time}]\\
\text{END TimePrim}
\end{array}
$$

Theory Connectives is copied from [Sha98]; it lifts the boolean connectives to the domain of predicates over a given type.

$$
\begin{array}{l}
\text{Connectives}[T : \text{TYPE}] : \text{THEORY}\\
\quad \text{BEGIN}\\
\quad P, Q : \text{VAR pred}[T]\\
\quad t : \text{VAR}\ T\\
\\
\quad \neg(P)(t) : \text{bool} = \neg P(t);\\
\quad \ldots\\
\quad \text{TRUE}(t) : \text{bool} = \text{TRUE}\\
\quad \text{FALSE}(t) : \text{bool} = \text{FALSE}\\
\text{END Connectives}
\end{array}
$$

The basic semantic primitives are defined in theory SemPrim. Since theory TimePrim is imported and no more restrictions are imposed on the time domain, we use the same parameters and also copy the assuming clause to be able to prove the generated Type Check Condition. This construction is used in all subsequent theories, except for concrete examples, and is omitted henceforth.

We declare a nonempty type of events and define the notion of an *observation function* which is a function from the time domain to a set of events. The intention is that such a function represents an externally observable timing behaviour of a component by describing the set of events that occur at each point of time. Events can be shared by several components, e.g., for synchronization or to observe the same physical event.

The basic semantic structure is defined by type `CompInfo` which is a record with two fields:

- Field α represents the *alphabet* of the component, that is, the set of events that might occur during the behaviour of the component.
- Field *obs* describes the observable behaviour of the component. Since we allow nondeterministic components, it is represented by a set of observation functions.

Note that components are not restricted to (parts of) computer programs; also physical components can be represented in this way, as will be illustrated by an example of a hybrid system in Sect. 6.

In subsequent sections, additional restrictions will be imposed on the type CompInfo, to represent choices concerning the semantic representation. Further note that the exposition here only concerns the externally observable behaviour and a representation of an internal state is omitted.

$$
\begin{array}{l}
\mathrm{SemPrim}[\mathrm{Time}:\ \mathrm{TYPE}, <:\ (\mathrm{strict\_order?}[\mathrm{Time}]),\\
\qquad\qquad\qquad\quad \leq:\ (\mathrm{partial\_order?}[\mathrm{Time}])]:\ \mathrm{THEORY}\\
\quad \mathrm{BEGIN}\\
\quad \mathrm{ASSUMING}\\
\quad\ \ \mathrm{leq\_less}:\ \mathrm{ASSUMPTION}\ \forall\ (t_1, t_2:\ \mathrm{Time}):\ t_1 \leq t_2 \Leftrightarrow t_1 < t_2 \vee t_1 = t_2\\
\quad \mathrm{ENDASSUMING}\\
\quad \mathrm{IMPORTING}\ \mathrm{TimePrim}[\mathrm{Time}, <, \leq]\\
\\
\quad \mathrm{Events}:\ \mathrm{NONEMPTY\_TYPE}\\
\\
\quad \mathrm{ObsFuncts}:\ \mathrm{TYPE}\ \ = [\mathrm{Time} \rightarrow\ \mathrm{setof}[\mathrm{Events}]]\\
\\
\quad \mathrm{CompInfo}:\ \mathrm{TYPE}\ \ = [\#\ \alpha:\ \mathrm{setof}[\mathrm{Events}], \mathrm{obs}:\ \mathrm{setof}[\mathrm{ObsFuncts}]\ \#]
\end{array}
$$

The usual operations on sets (of events) are lifted to observation functions.

$$
\begin{array}{l}
\quad o, o_1, o_2, o_3:\ \mathrm{VAR}\ \mathrm{ObsFuncts}\\
\quad t, t_0, t_1, t_2, t_3, t_4:\ \mathrm{VAR}\ \mathrm{Time}
\end{array}
$$

$\text{ci}: \ \text{VAR CompInfo}$
$\text{Eset}, \text{Eset0}, \text{Eset1}, \text{Eset2}: \ \text{VAR setof[Events]}$
$e: \ \text{VAR Events}$

$o_1 \cup o_2: \ \text{ObsFuncts} \ = \ \lambda\ t: \ o_1(t) \cup o_2(t)$
$o \cap \text{Eset}: \ \text{ObsFuncts} \ = \ \lambda\ t: \ o(t) \cap \text{Eset}$
$o \setminus \text{Eset}: \ \text{ObsFuncts} \ = \ \lambda\ t: \ o(t) \setminus \text{Eset}$
$o \subseteq \text{Eset}: \ \text{bool} \ = \ \forall\ t: \ o(t) \subseteq \text{Eset}$

$\text{obs\_union\_comm}: \ \text{LEMMA } o_1 \cup o_2 = o_2 \cup o_1$
$\text{obs\_union\_assoc}: \ \text{LEMMA } (o_1 \cup o_2) \cup o_3 = o_1 \cup (o_2 \cup o_3)$
$\text{intersection\_difference}: \ \text{LEMMA } o \cap (\text{Eset1} \setminus \text{Eset2}) = (o \setminus \text{Eset2}) \cap \text{Eset1}$

With the current definitions, $o(t)(\text{read})$ expresses that a event read occurs at time t (note that a set of events is represented as a predicate on events). To be able to use the notations from theory TimePrim, one should be able to write $o(\text{read})(t)$ and then, for instance, read **in** $[3, 7)$. This can be achieved by defining a conversion At which makes it possible to interpret o as $\text{At}(o)$ when convenient.

$\text{At}(o)(e)(t): \ \text{bool} \ = \ o(t)(e)$

CONVERSION At
END SemPrim

Henceforth, declarations of variables are not repeated, so below o, o_1, t, t_0, etc., are used without declaration.

3 Parallel Composition

The parallel composition of real-time components is studied, where we consider in this section a representation where events outside the alphabet of a process, i.e. events of the environment, are ignored. In Sect. 3.1, a denotational semantics is presented. To increase the confidence in the definition, several equivalent formulations are studied. Section 3.2 contains a general framework to denote assertional specifications. A compositional rule for parallel composition can be found in Sect. 3.3.

3.1 Denotational Semantics of Parallel Composition

Theory Sem defines the type Comps of components by imposing the restriction that any observation function of a component contains only events of its alphabet. So an observation function of a component describes only the behaviour that is relevant for the component itself.

$\text{Sem}[\text{Time}: \textsc{type} \ldots]: \textsc{theory}$
$\quad\textsc{begin}$
$\quad\textsc{assuming}\ldots$
$\quad\textsc{importing}\ \text{SemPrim}[\text{Time}, <, \leq]$

$$\text{ObsInAlpha}(\text{ci}): \text{bool} = \forall\ o: \text{obs}(\text{ci})(o) \Rightarrow o \subseteq \alpha(\text{ci})$$

$$\text{Comps}: \textsc{type} = \{\text{ci} \mid \text{ObsInAlpha}(\text{ci})\}$$

$\textsc{end}\ \text{Sem}$

Parallel composition of two components can be defined by taking the union of the alphabets and the (pointwise) union of the observation functions, with the additional restriction that these observation functions agree (i.e., are equal) on the events that are in both alphabets.

$\text{SemPar}[\text{Time}: \textsc{type} \ldots]: \textsc{theory}$
$\quad\textsc{begin}$
$\quad\textsc{assuming}\ldots$
$\quad\textsc{importing}\ \text{Sem}[\text{Time}, <, \leq]$

$$\text{comp}, \text{comp0}, \text{comp1}, \text{comp2}, \text{comp3}, \text{comp4}: \textsc{var}\ \text{Comps}$$

$$\begin{aligned}&\text{par}(\text{comp1}, \text{comp2}): \text{Comps} = \\ &\quad (\#\ \alpha := \alpha(\text{comp1}) \cup \alpha(\text{comp2}), \\ &\qquad \text{obs} := \lambda\ o: (\exists\ o_1, o_2: \text{obs}(\text{comp1})(o_1) \wedge \text{obs}(\text{comp2})(o_2) \wedge o = o_1 \cup o_2 \wedge \\ &\qquad\qquad o_1 \cap \alpha(\text{comp1}) \cap \alpha(\text{comp2}) = o_2 \cap \alpha(\text{comp1}) \cap \alpha(\text{comp2}))\ \#)\end{aligned}$$

Since par(comp1, comp2) should be of type Comps, type-checking leads to a Type Check Condition requiring a proof that the definition indeed satisfies ObsInAlpha. The proof uses the fact that this property already holds for the components.

Moreover, ObsInAlpha has been used to show that an equivalent definition is obtained if $o_1 \cap \alpha(\text{comp1}) \cap \alpha(\text{comp2}) = o_2 \cap \alpha(\text{comp1}) \cap \alpha(\text{comp2})$ is replaced by $o_1 \cap \alpha(\text{comp2}) = o_2 \cap \alpha(\text{comp1})$.

Example 2. Consider a component P_1 which performs a *read* event at time 3 and a $comm(1)$ event between 7 and 9. Component P_2 has the following behaviour; if a $comm(v)$ event occurs at time t, for some value v, then it performs a $write(v+1)$ between $t+5$ and $t+10$. Note that this behaviour is represented by a possibly infinite set of observation functions; for all values of t and v and any time point between $t+5$ and $t+10$ there exists an observation function.

To obtain a behaviour of $\text{par}(P_1, P_2)$ we take the union of a behaviour of P_1 and one of P_2, provided they agree on the joint *comm* event. Hence we can only use observation functions of P_2 that contain a $comm(1)$ event between 7 and 9, and thus have a $write(2)$ event between 12 and 19. Hence the observation functions of $\text{par}(P_1, P_2)$ contain a *read* at 3, a $comm(1)$ event between 7 and 9, and a $write(2)$ between 12 and 19.

To increase the confidence in the definition of parallel composition, an alternative definition is formulated and shown to be equivalent with the previous one. This alternative definition is similar to the trace-based semantics of parallel composition [Hoa85], which turned out to be useful for the formulation of a compositional rule for (untimed) parallel composition [Zwi89]. Basic idea is that the projection of an observation of the parallel construct onto the alphabet of one of the components leads to an observation of this component. Projection is here represented by intersection. Additionally, any observation function of the composition should only contain events of its alphabet. Note that // can be used as an infix operator in PVS.

$$
\begin{array}{l}
//(\mathrm{comp1}, \mathrm{comp2}) : \ \mathrm{Comps} \ = \\
\quad (\#\ \alpha := \ \alpha(\mathrm{comp1}) \cup \alpha(\mathrm{comp2}), \\
\qquad \mathrm{obs} := \lambda\ o : (\exists\ o_1, o_2 : \mathrm{obs}(\mathrm{comp1})(o_1) \wedge \mathrm{obs}(\mathrm{comp2})(o_2) \wedge \\
\qquad\qquad\qquad\qquad\qquad o \cap \alpha(\mathrm{comp1}) = o_1 \wedge o \cap \alpha(\mathrm{comp2}) = o_2 \wedge \\
\qquad\qquad\qquad\qquad\qquad o \subseteq \alpha(\mathrm{comp1}) \cup \alpha(\mathrm{comp2}))\ \#)
\end{array}
$$

$\mathrm{AltSemEquiv} : \ \textsc{theorem}\ \mathrm{comp1}\ //\ \mathrm{comp2} = \mathrm{par}(\mathrm{comp1}, \mathrm{comp2})$

Example 3. Consider again the processes P_1 and P_2 of Example 2. Then any observation function of $P_1//P_2$ should lead to an observation function of P_1 if the intersection (projection) is taken with the events of P_1, viz. *read* and *comm*. Hence any observation function of $P_1//P_2$ should have a *read* event at 3 and a $comm(1)$ event between 7 and 9. Since the projection onto events of P_2, viz. *comm* and *write* should lead to an observation function of P_2, this implies that there is a $write(2)$ between 12 and 19.

As a further test of the definitions, it can be shown that parallel composition boils down to a simple union of observation functions if there are no shared events. Moreover, we show that // is idempotent.

$$
\begin{array}{l}
\mathrm{NoSharedEvents} : \ \textsc{lemma}\ \alpha(\mathrm{comp1}) \cap \alpha(\mathrm{comp2}) = \emptyset \Rightarrow \\
\quad \mathrm{comp1}\ //\ \mathrm{comp2} = \\
\qquad (\#\ \alpha := \ \alpha(\mathrm{comp1}) \cup \alpha(\mathrm{comp2}), \\
\qquad\quad \mathrm{obs} := \lambda\ o : (\exists\ o_1, o_2 : \mathrm{obs}(\mathrm{comp1})(o_1) \wedge \mathrm{obs}(\mathrm{comp2})(o_2) \wedge o = o_1 \cup o_2)\ \#)
\end{array}
$$

$\mathrm{Idempotence} : \ \textsc{fact}\ \mathrm{comp}\ //\ \mathrm{comp} = \mathrm{comp}$

END SemPar

3.2 Specifications

Common to the two versions of the semantic framework presented here, is a notion of refinement between components. Component ci1 refines (implements) components ci2, denoted by ci1 $\Rrightarrow$ ci2, if the alphabet of "implementation" ci1 is an extension of that of "specification" ci2 and the observable behaviour of ci1 is included in that of ci2.

$\mathrm{SpecsPrim}[\mathrm{Time}: \text{TYPE} \ldots]: \text{THEORY}$
BEGIN
ASSUMING...
IMPORTING SemPrim[Time, <, ≤]

$$\mathrm{ci1} \Rrightarrow \mathrm{ci2}: \ \mathrm{bool} \ = \alpha(\mathrm{ci2}) \subseteq \alpha(\mathrm{ci1}) \wedge \mathrm{obs}(\mathrm{ci1}) \subseteq \mathrm{obs}(\mathrm{ci2})$$

$$\mathrm{RefRefl}: \ \text{THEOREM} \ \mathrm{ci} \Rrightarrow \mathrm{ci}$$
$$\mathrm{RefTrans}: \ \text{THEOREM} \ (\mathrm{ci0} \Rrightarrow \mathrm{ci2}) \Leftrightarrow (\exists \ \mathrm{ci1}: \ (\mathrm{ci0} \Rrightarrow \mathrm{ci1}) \wedge (\mathrm{ci1} \Rrightarrow \mathrm{ci2}))$$

Instead of a semantic description it is often convenient to specify components using assertions, which are simply predicates over observation functions.

$$\mathrm{Assertion}: \ \text{TYPE} \ = \mathrm{pred}[\mathrm{ObsFuncts}]$$
IMPORTING Connectives[ObsFuncts]
$$A, A_0, A_1, A_2: \ \text{VAR Assertion}$$
$$\mathrm{Valid}(A): \ \mathrm{bool} \ = \ \forall \ o: \ A(o)$$
END SpecsPrim

In general, assertional specifications can have a certain structure with, for instance pre and post conditions or rely/guarantee pairs. To illustrate the basic concepts, here a specification simply consists of an alphabet and a single assertion.

To allow mixed terms, combining specifications and programming constructs in a uniform way, specifications also have type Comps. Since this requires a proof of ObsInAlpha, an observation function satisfying the specification should only contain events of the alphabet.

$\mathrm{Specs}[\mathrm{Time}: \text{TYPE} \ldots]: \text{THEORY}$
BEGIN
ASSUMING...
IMPORTING Sem[Time, <, ≤], SpecsPrim[Time, <, ≤]

$$\mathrm{spec}(\mathrm{Eset}, A): \ \mathrm{Comps} \ = \ (\# \ \alpha := \ \mathrm{Eset}, \ \mathrm{obs} := \ \lambda \ o: \ o \subseteq \mathrm{Eset} \wedge A(o) \ \#)$$

END Specs

Example 4. Assume given read and write events.

$$\mathrm{read}, \mathrm{write}: \ \mathrm{Events}$$
$$\mathrm{ReadWriteDiff}: \ \text{AXIOM} \ \mathrm{read} \neq \mathrm{write}$$

Observe that component C_1

$$C_1: \ \mathrm{Comps} \ = \ \mathrm{spec}(\{e \mid \ e = \mathrm{write}\}, \ \lambda \ o: \ o(7)(\mathrm{write}))$$

specifies observation functions that contain only write events. Component C_2

$C_2 : \text{ Comps } = \text{ spec}(\{e \mid e = \text{read} \vee e = \text{write}\},\ \lambda\, O :\ O(7)(\text{write}))$

additionally allows arbitrary read events. Lemma ActionFalse below expresses that we obtain an empty set of observation functions is we restrict the alphabet to read events. On the other hand, assertion $\lambda\ o :\ \neg o(7)(\text{write})$ does not impose any additonal restriction.

$$\begin{aligned} \text{ActionFalse} : \ \text{LEMMA}\ & \text{spec}(\{e \mid e = \text{read}\},\ \lambda\, o :\ o(7)(\text{write})) = \\ & \text{spec}(\{e \mid e = \text{read}\}, \text{FALSE}) \end{aligned}$$

$$\begin{aligned} \text{NoActionTrue} : \ \text{LEMMA}\ & \text{spec}(\{e \mid e = \text{read}\},\ \lambda\, o :\ \neg o(7)(\text{write})) = \\ & \text{spec}(\{e \mid e = \text{read}\}, \text{TRUE}) \end{aligned}$$

3.3 Compositional Proof Rule for Parallel Composition

Main topic of this chapter is the formulation of a compositional rule for parallel composition. First, however, we prove the soundness of a consequence rule, which allows weakening of assertions.

RulePar[Time : TYPE ...] : THEORY
  BEGIN
  ASSUMING...
  IMPORTING SemPar[Time, $<, \leq$], Specs[Time, $<, \leq$]

$$\begin{aligned} \text{ConsRule} : \ \text{THEOREM}\ & \text{Eset0} = \text{Eset} \wedge \text{Valid}(A_0 \Rightarrow A) \Rightarrow \\ & (\text{spec}(\text{Eset0}, A_0) \Rrightarrow \text{spec}(\text{Eset}, A)) \end{aligned}$$

The next example shows that the condition Eset0 = Eset of the consequence rule cannot be weakened to Eset $\subseteq$ Eset0.

Example 5. Component C_2 is not a refinement of C_1, because C_1 specifies, by its alphabet, that there are no read events, whereas C_2 allows arbitrary read events.

$$\text{C2notrefC1} : \ \text{FACT}\ \neg(C_2 \Rrightarrow C_1)$$

The monotonicity rule is important for the composition of refinement steps.

$$\begin{aligned} \text{MonoPar} : \ \text{THEOREM}\ & (\text{comp1} \Rrightarrow \text{comp3}) \wedge (\text{comp2} \Rrightarrow \text{comp4}) \Rightarrow \\ & (\text{comp1} \mathbin{//} \text{comp2} \Rrightarrow \text{comp3} \mathbin{//} \text{comp4}) \end{aligned}$$

Next, the aim is to formulate a rule where the parallel composition of specifications of components corresponds to the conjunction of the assertions:

$$\text{spec}(\text{Eset1}, A_1) \mathbin{//} \text{spec}(\text{Eset2}, A_2) \Rrightarrow \text{spec}(\text{Eset1} \cup \text{Eset2}, A_1 \wedge A_2). \qquad (*)$$

The following example shows, however, that this is not valid if, e.g., assertion A_1 refers to events of Eset2 that are not in Eset1.

Example 6. Formula (*) would lead to

$$\begin{aligned}&\mathrm{spec}(\{e \mid\ e = \mathrm{read}\},\ \lambda\, o:\ \neg o(7)(\mathrm{write}))\\ &\ //\ \mathrm{spec}(\{e \mid\ e = \mathrm{write}\},\ \lambda\, o:\ o(7)(\mathrm{write})) \Rrightarrow\\ &\qquad \mathrm{spec}(\{e \mid\ e = \mathrm{read} \vee e = \mathrm{write}\}, \mathrm{FALSE})\end{aligned}$$

and hence, using Example 4,

$$\begin{aligned}&\mathrm{spec}(\{e \mid\ e = \mathrm{read}\}, \mathrm{TRUE})\\ &\ //\ \mathrm{spec}(\{e \mid\ e = \mathrm{write}\},\ \lambda\, o:\ o(7)(\mathrm{write})) \Rrightarrow\\ &\qquad \mathrm{spec}(\{e \mid\ e = \mathrm{read} \vee e = \mathrm{write}\}, \mathrm{FALSE})\end{aligned}$$

To rule out such counter examples, it is required that validity of the assertion of a component only depends on the events of its alphabet. Hence we define OnlyDepEve(A, Eset) to express that validity of A only depends on the events in Eset. Again, to increase the confidence, we prove the equivalence with an alternative formulation which also turns out to be convenient in proofs.

$$\mathrm{OnlyDepEve}(A, \mathrm{Eset}):\ \mathrm{bool}\ =\ \forall\, o_1, o_2 : A(o_1) \wedge o_1 \cap \mathrm{Eset} = o_2 \cap \mathrm{Eset} \Rightarrow A(o_2)$$

$$\mathrm{OnlyDepEveEquiv}:\ \textsc{lemma}\ \mathrm{OnlyDepEve}(A, \mathrm{Eset}) \Leftrightarrow (\forall\, o:\ A(o) \Leftrightarrow A(o \cap \mathrm{Eset}))$$

$$\begin{aligned}&\mathrm{ParCompRule}:\ \textsc{theorem}\ \mathrm{OnlyDepEve}(A_1, \mathrm{Eset1}) \wedge \mathrm{OnlyDepEve}(A_2, \mathrm{Eset2}) \Rightarrow\\ &\quad \mathrm{spec}(\mathrm{Eset1}, A_1)\ //\ \mathrm{spec}(\mathrm{Eset2}, A_2) \Rrightarrow \mathrm{spec}(\mathrm{Eset1} \cup \mathrm{Eset2}, A_1 \wedge A_2)\end{aligned}$$

Theorem ParCompRule, expressing soundness of the parallel composition rule, has been proved as follows. Assume that we have OnlyDepEve(A_1, Eset1) and OnlyDepEve(A_2, Eset2). Let $o \in \mathrm{obs}(\mathrm{spec}(\mathrm{Eset1}, A_1)\ //\ \mathrm{spec}(\mathrm{Eset2}, A_2))$. Then there exist, e.g., an o_1 such that $o \cap \mathrm{Eset1} = o_1$ and $A_1(o_1)$, and hence we have $A_1(o \cap \mathrm{Eset1})$. By OnlyDepEveEquiv this leads to $A_1(o)$. By symmetry, we obtain $o \in \mathrm{obs}(\mathrm{spec}(\mathrm{Eset1} \cup \mathrm{Eset2}, A_1 \wedge A_2))$.

4 Semantics of Parallel Composition by Intersection

In this section we aim at a framework where the semantics of parallel composition can be formulated as the intersection of the sets of behaviours of the two components. To achieve this, here the semantic representation of a component contains arbitrary events of the environment. The relation with the semantics of the previous section is studied in Sect. 4.1.

First define a type of components, called CompsEnv, where we require that any arbitrary behaviour outside the alphabet is included.

$$\begin{aligned}&\mathrm{ArbOutAlpha}(\mathrm{ci}):\ \mathrm{bool}\ = \forall\, o_1, o_2 :\\ &\quad \mathrm{obs}(\mathrm{ci})(o_1) \wedge o_1 \cap \alpha(\mathrm{ci}) = o_2 \cap \alpha(\mathrm{ci}) \Rightarrow \mathrm{obs}(\mathrm{ci})(o_2)\end{aligned}$$

$$\mathrm{CompsEnv}:\ \textsc{type}\ =\ \{\mathrm{ci} \mid\ \mathrm{ArbOutAlpha}(\mathrm{ci})\}$$

Parallel composition is defined by the intersection of the sets of behaviours.

$\mathrm{ce}, \mathrm{ce0}, \mathrm{ce1}, \mathrm{ce2}, \mathrm{ce3}$: VAR CompsEnv

$$//(\mathrm{ce1}, \mathrm{ce2}) : \mathrm{CompsEnv} = \\ (\# \ \alpha := \alpha(\mathrm{ce1}) \cup \alpha(\mathrm{ce2}),\ \mathrm{obs} := \mathrm{obs}(\mathrm{ce1}) \cap \mathrm{obs}(\mathrm{ce2}) \ \#)$$

Example 7. Consider again the two components of Example 2: P_1 which does a *read* at time 3 and a $comm(1)$ between 7 and 9, P_2 responds to a $comm(v)$ at time t with a $write(v+1)$ between $t+5$ and $t+10$. Now the observation functions of P_1 contain arbitrary events outside its alphabet (*read* and *comm*), including arbitrary *write* events. Similarly, the observable behaviour of P_2 contains observation functions with arbitrary *read* events. Taking the intersection of these sets of functions, we obtain all observation functions that contain a *read* at 3, a $comm(1)$ event between 7 and 9, and a $write(2)$ between 12 and 19.

In our mixed framework, specifications are also components, and hence the definition of specifications has to be adapted to obtain arbitrary behaviour outside the alphabet.

$$\mathrm{spec}(\mathrm{Eset}, A) : \mathrm{CompsEnv} = \\ (\# \ \alpha := \mathrm{Eset},\ \mathrm{obs} := \lambda\ o : (\exists\ o_1 : A(o_1) \wedge o \cap \mathrm{Eset} = o_1 \cap \mathrm{Eset}) \ \#)$$

Example 8. Observe that for the components C_1 and C_2 of Example 4, which are now of type CompsEnv, we have

C2refC1 : FACT $C_2 \Rrightarrow C_1$

since C_1 allows more arbitrary behaviour (all non-write events) than C_2 (all non-write and non-read events).

As before we have NoActionTrue, i.e. a specification expressing no activity outside the alphabet is equivalent to true. But now also assertions expressing the occurrence of events outside the alphabet are ignored, as expressed by ActionTrue

$$\mathrm{ActionTrue} : \text{LEMMA } \mathrm{spec}(\{e \mid e = \mathrm{read}\},\ \lambda\ o :\ o(7)(\mathrm{write})) = \\ \mathrm{spec}(\{e \mid e = \mathrm{read}\}, \mathrm{TRUE})$$

$$\mathrm{NoActionTrue} : \text{LEMMA } \mathrm{spec}(\{e \mid e = \mathrm{read}\},\ \lambda\ o :\ \neg o(7)(\mathrm{write})) = \\ \mathrm{spec}(\{e \mid e = \mathrm{read}\}, \mathrm{TRUE})$$

Nice is that in this framework the consequence rule can be strengthened, since the condition about the alphabets can be weakened to a subset relation.

$$\mathrm{ConsRule} : \text{THEOREM } \mathrm{Eset} \subseteq \mathrm{Eset0} \wedge \mathrm{Valid}(A_0 \Rightarrow A) \Rightarrow \\ (\mathrm{spec}(\mathrm{Eset0}, A_0) \Rrightarrow \mathrm{spec}(\mathrm{Eset}, A))$$

The rules for monotonicity and parallel composition remain unchanged. The proof of ParCompRule now proceeds as follows. Again assume that we have OnlyDepEve(A_1, Eset1) and OnlyDepEve(A_2, Eset2). Consider an observation $o \in \text{obs}(\text{spec}(\text{Eset1}, A_1) \;//\; \text{spec}(\text{Eset2}, A_2))$. Then, e.g., $o \in \text{obs}(\text{spec}(\text{Eset1}, A_1))$ and hence there exists an o_1 such that $A_1(o_1)$ and $o \cap \text{Eset1} = o_1 \cap \text{Eset1}$. Hence OnlyDepEve leads to $A_1(o)$. By symmetry, $o \in \text{obs}(\text{spec}(\text{Eset1} \cup \text{Eset2}, A_1 \wedge A_2))$.

Finally observe that Example 6 can also be used here to show that the OnlyDepEve conditions are needed for the soundness of the rule.

4.1 Relating the Frameworks

To relate the two approaches and the corresponding definitions of parallel composition, two functions are defined to transform one representation into the other. They are each others inverse.

$$
\begin{array}{l}
\text{AddEnv(comp)} : \ \text{CompsEnv} \ = \\
\quad (\#\ \alpha := \ \alpha(\text{comp}), \\
\qquad \text{obs} := \lambda\ o_1 : (\exists\ o_2 : \text{obs(comp)}(o_2) \wedge o_1 \cap \alpha(\text{comp}) = o_2 \cap \alpha(\text{comp}))\ \#)
\end{array}
$$

$$
\begin{array}{l}
\text{RemEnv(ce)} : \ \text{Comps} \ = \\
\quad (\#\ \alpha := \ \alpha(\text{ce}), \\
\qquad \text{obs} := \ \lambda\ o_1 : \ (\exists\ o_2 : \ \text{obs(ce)}(o_2) \wedge o_1 = o_2 \cap \alpha(\text{ce}))\ \#)
\end{array}
$$

$$
\begin{array}{l}
\text{AddRemProp} : \ \textsc{lemma}\ \text{AddEnv(RemEnv(ce))} = \text{ce} \\
\text{RemAddProp} : \ \textsc{lemma}\ \text{RemEnv(AddEnv(comp))} = \text{comp}
\end{array}
$$

Theorems RemRel and AddRel relate the two versions of parallel composition. Note that AddRel can be proved easily from RemRel and the properties AddRemProp and RemAddProp.

$$
\begin{array}{l}
\text{RemRel} : \ \textsc{theorem}\ \text{RemEnv(ce1 // ce2)} = \text{RemEnv(ce1) // RemEnv(ce2)} \\
\text{AddRel} : \ \textsc{theorem} \\
\qquad\qquad \text{AddEnv(comp1 // comp2)} = \text{AddEnv(comp1) // AddEnv(comp2)}
\end{array}
$$

We have not yet addressed commutativity and associativity of parallel composition in the framework of Sect. 3. The reason is that it is much easier to prove these properties first in the framework with an arbitrary environment, where we can simply use the properties of union and intersection from the PVS prelude.

$$
\begin{array}{l}
\text{ParCommEnv} : \ \textsc{lemma}\ \text{ce1 // ce2} = \text{ce2 // ce1} \\
\text{ParAssocEnv} : \ \textsc{lemma}\ \text{(ce1 // ce2) // ce3} = \text{ce1 // (ce2 // ce3)}
\end{array}
$$

By RemAddProp and theorem AddRel these results can be tranformed easily to the framework without environment events.

$$
\begin{array}{l}
\text{ParComm} : \ \textsc{lemma}\ \text{comp1 // comp2} = \text{comp2 // comp1} \\
\text{ParAssoc} : \ \textsc{lemma}\ \text{(comp1 // comp2) // comp3} = \text{comp1 // (comp2 // comp3)}
\end{array}
$$

5 Hiding of Internal Events

Besides parallel composition, also the possibility to encapsulate internal events is important during top-down design. In Sect. 5.1 a hiding operator is defined in the framework of Sect. 3. The other framework, with an arbitrary enironment, is considered in Sect. 5.2.

5.1 Hiding in the Framework without Environment Events

To hide a set of events Eset from component comp, denoted by "comp - Eset", the elements of Eset are removed from the alphabet and the observation functions.

$\mathrm{Hiding}[\mathrm{Time}: \text{TYPE} \ldots]: \text{THEORY}$
$\quad \text{BEGIN}$
$\quad \text{ASSUMING} \ldots$
$\quad \text{IMPORTING } \mathrm{RulePar}[\mathrm{Time}, <, \leq]$

$$
\begin{aligned}
&-(\mathrm{comp}, \mathrm{Eset}): \mathrm{Comps} \ = \\
&\qquad (\# \ \alpha := \ \alpha(\mathrm{comp}) \setminus \mathrm{Eset}, \\
&\qquad\quad \mathrm{obs} := \ \lambda \ o : \ (\exists \ o_1 : \ \mathrm{obs}(\mathrm{comp})(o_1) \wedge o = o_1 \setminus \mathrm{Eset}) \ \#)
\end{aligned}
$$

To show that the definition indeed leads to an element of type Comps, type-checking leads to the requirement to prove that it satisfies ObsInAlpha.

We also prove a monotonicity rule and a hiding rule, which requires that the assertion of the specification only depends on the remaining, not hidden, events.

$$\mathrm{HideMono}: \ \text{THEOREM} \ (\mathrm{comp1} \Rrightarrow \mathrm{comp2}) \Rightarrow (\mathrm{comp1} \ - \ \mathrm{Eset} \Rrightarrow \mathrm{comp2} \ - \ \mathrm{Eset})$$

$$
\begin{aligned}
&\mathrm{HideRule}: \ \text{THEOREM} \ \mathrm{OnlyDepEve}(A, \mathrm{Eset} \setminus \mathrm{Eset0}) \Rightarrow \\
&\qquad\qquad \mathrm{spec}(\mathrm{Eset}, A) \ - \ \mathrm{Eset0} \Rrightarrow \mathrm{spec}(\mathrm{Eset} \setminus \mathrm{Eset0}, A)
\end{aligned}
$$

END Hiding

Example 9. For the components C_1 and C_2 of Example 4 we can prove

$$\mathrm{C1hideC2}: \ \text{LEMMA} \ C_1 = C_2 \ - \ \{e \mid \ e = \mathrm{read}\}$$

5.2 Hiding in the Framework with an Arbitrary Environment

In the alternative framework we cannot simply remove internal events from the observation function. On the contrary, hiding is achieved by including any arbitrary behaviour concerning the events to be hidden.

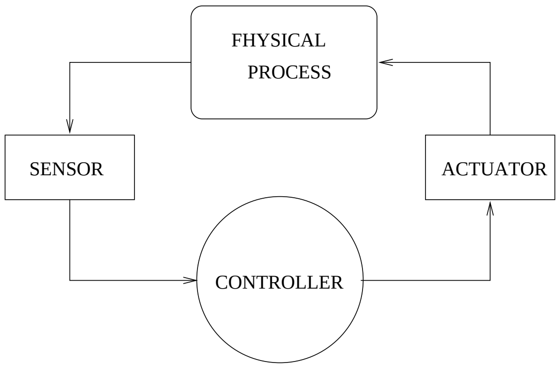
$$
\begin{aligned}
&-(\mathrm{ce}, \mathrm{Eset}): \ \mathrm{CompsEnv} \ = \\
&\qquad (\# \ \alpha := \ \alpha(\mathrm{ce}) \setminus \mathrm{Eset}, \\
&\qquad\quad \mathrm{obs} := \lambda \ o : (\exists \ o_1 : \mathrm{obs}(\mathrm{ce})(o_1) \wedge o \setminus \mathrm{Eset} = o_1 \setminus \mathrm{Eset}) \ \#)
\end{aligned}
$$

The rules are identical to the previous section. Similar to parallel composition (Sect. 4.1), it is reassuring that the two versions are isomorphic.

$$
\begin{aligned}
&\mathrm{HideRemRel}: \ \text{LEMMA} \ \mathrm{RemEnv}(\mathrm{ce} \ - \ \mathrm{Eset}) = \mathrm{RemEnv}(\mathrm{ce}) \ - \ \mathrm{Eset} \\
&\mathrm{HideAddRel}: \ \text{LEMMA} \ \mathrm{AddEnv}(\mathrm{comp} \ - \ \mathrm{Eset}) = \mathrm{AddEnv}(\mathrm{comp}) \ - \ \mathrm{Eset}
\end{aligned}
$$

6 Hybrid Systems

As an application of the theory above, we present a simple example of a hybrid system, i.e. a system with discrete and continuous components. As a typical example, we consider in Sect. 6.3 a simple process control application with the following structure.



The general outline of our approach for such systems is described in Sect. 6.2. We will use the framework of Sect. 3 and Sect. 5.1 (i.e., without environment events), to show a few steps towards the design of a discrete controller. Further, the example serves to illustrate recent ideas on platform-independent development of real-time systems [HvR97], as indicated in the next section.

6.1 Platform-Independence

The goal is to postpone platform-dependent design decisions as long as possible. To achieve this, we distinguish two activities:

– A platform-independent programming activity, where a program is developed independent of any particular architecture on which it has to be executed. To achieve this, a notation has been devised to annotate programs with timing specifications. By programming the functional behaviour and only specifying the timing behaviour, we obtain a context and platform independent way of describing algorithms, similar to untimed system design.
– An activity where the program, including timing annotations, is realized on a particular platform. This involves the transformation of the annotated program into scheduling blocks plus timed precedence constraints, and the design of a schedule for the execution platform. Different from the usual compilation phase for untimed programs is that schedulability analysis might indicate that it is not possible to find a schedule and either the program or the platform has to be adapted.

This apprach should be contrasted with the traditional practice where in an early phase of the design internal deadlines, periods, and priorities are chosen. Thus making the design both context dependent (e.g., even depending on priorities of unrelated tasks) and implementation dependent (e.g., depending on the duration of basic statements and the scheduling policy).

6.2 Design of Process Control Systems

To design a real-time computer system which controls physical processes, we propose the following approach.

1. Formulate the requirements specification of the complete system, including continuous components.
2. Formalize the assumptions about the physical processes in the system.
3. Specify the control component, in general using continuous quantities (as in the previous steps).
4. Verify the control algorithm of the previous step, i.e. show that the specifications of 2. and 3. lead to the properties specified in step 1.
5. Step-wise transformation of the continuous control strategy (of step 3) into a specification in terms of a discrete interface. Usually this is done by means of sensors and actuators, assuming formal specifications of these components.
6. Design a program satisfying the discrete specification, obtained in the previous step, of the control component.

6.3 Example Hybrid System

The example presented here is a simplified version of a mine pump example [MJ96]. Consider a mine with a certain engine (e.g., a pump to evacuate water) which should not be operating when a certain amount of gas is present, because that might lead to an explosion. Following the steps above, first the requirements specification of the mine system is formulated.

Requirements Specification As a first specification of the system, we simply require that no explosion should occur at any point of time, where we choose the real numbers as our (continuous) time domain.

ExplEx : THEORY
  BEGIN
  Time : TYPE $=$ real
  NonNegTime : TYPE $= \{t : \text{Time} \mid t \geq 0\}$
  IMPORTING Hiding[Time, $<, \leq$]

  expl : Events

$$\mathrm{EMS} : \ \mathrm{setof[Events]} \ = \ \{e \mid \ e = \mathrm{expl}\}$$

$$\mathrm{AMS} : \ \mathrm{Assertion} \ = \ \lambda\ o : \ \forall\ t : \ \neg o(t)(\mathrm{expl})$$

$$\mathrm{MineSystem} : \ \mathrm{Comps} \ = \ \mathrm{spec(EMS, AMS)}$$

Specify Physical Environment Next we specify the assumption that an explosion can only occur if the engine is operating and there has been gas for at least, say, CritGasPeriod time units. For simplicity, suppose there is an event *engine* which occurs iff the engine is operating. An alternative is to use start and stop events for the engine, and to define a predicate expressing that the engine is on if the last event was a start event.

$$\mathrm{gas, engine} : \ \mathrm{Events}$$

$$\mathrm{ContEventsDiffer} : \ \textsc{axiom}\ \mathrm{gas} \neq \mathrm{expl} \wedge \mathrm{gas} \neq \mathrm{engine} \wedge \mathrm{expl} \neq \mathrm{engine}$$

$$\mathrm{EM} : \ \mathrm{setof[Events]} \ = \{e \mid \ e = \mathrm{expl} \vee e = \mathrm{gas} \vee e = \mathrm{engine}\}$$

$$\mathrm{CritGasPeriod} : \ \mathrm{NonNegTime}$$

$$\mathrm{AM} : \ \mathrm{Assertion} \ = \lambda\ o : \\ \forall\ t : \ o(\mathrm{expl})(t) \Rightarrow o(\mathrm{engine})(t) \wedge o(\mathrm{gas})\ \mathbf{during}\ [t - \mathrm{CritGasPeriod}, t]$$

$$\mathrm{Mine} : \ \mathrm{Comps} \ = \ \mathrm{spec(EM, AM)}$$

Specify Control Component The aim is to specify a control strategy Control, in terms of the physical events gas and engine, such that together with specification Mine we obtain the required specification MineSystem. Hence the alphabet is defined by EC.

$$\mathrm{EC} : \ \mathrm{setof[Events]} \ = \ \{e \mid \ e = \mathrm{gas} \vee e = \mathrm{engine}\}$$

As a first attempt, we specify that if there is gas at some point in time, then the engine is switched off within, say, StopDelay time units;

$\lambda\ o : \ \forall\ t : \ o(t)(\mathrm{gas}) \Rightarrow \neg o(\mathrm{engine})\ \mathbf{in}\ [t, t + \mathrm{StopDelay}]$

Assuming StopDelay $\leq$ CritGasPeriod this indeed leads to MineSystem. But the specification is stronger than necessary and imposes an unrealistic condition on the implementation; gas needs to be observed at any point in time, since a response is needed for any single point in time where gas occurs. Therefore, as a next attempt, we weaken the assertion, requiring that if there is gas during a certain period then the engine should be off after StopDelay.

$\lambda\ o : \forall\ t_1, t_2 : o(\mathrm{gas})\ \mathbf{during}\ [t_1, t_2] \Rightarrow \neg o(\mathrm{engine})\ \mathbf{during}\ [t_1 + \mathrm{StopDelay}, t_2]$

Observe that the assertion is trivially true if $t_2 < t_1 + \mathrm{StopDelay}$, and hence a response is only needed if there is gas during StopDelay time units. Again we can derive MineSystem, provided StopDelay $\leq$ CritGasPeriod.

However, aiming at platform-independence, we try to avoid internal reaction times, such as StopDelay, and corresponding conditions on them. This leads to the following specification, which only refer to the given constant CritGasPeriod.

$$\begin{array}{l} \text{AC}: \text{ Assertion } = \lambda\, o : \\ \quad \forall\, t : o(\text{gas}) \textbf{ during } [t, t + \text{CritGasPeriod}] \Rightarrow \neg o(\text{engine})(t + \text{CritGasPeriod}) \end{array}$$

$$\text{Control}: \text{ Comps } = \text{ spec}(\text{EC}, \text{AC})$$

Verification of the Control Algorithm Main part of the verification is the proof that the assertions of the mine and the control algorithm lead to the assertion of the system, as expressed by lemma MSAssert.

$$\text{MSAssert}: \text{ LEMMA Valid}(\text{AM} \wedge \text{AC} \Rightarrow \text{AMS})$$

To apply the parallel composition rule we first prove OnlyDepEve(AM, EM) and OnlyDepEve(AC, EC). Then the rules ParCompRule and ConsRule, together with transitivity of refinement (theorem RefTrans) lead essentially to the required specification, except that the alphabet still contains the events gas and engine.

$$\text{MSIntEve}: \text{ setof[Events] } = \{e \mid e = \text{gas} \vee e = \text{engine}\}$$

$$\text{MSPar}: \text{ THEOREM Mine // Control} \Rrightarrow \text{spec}(\text{EMS} \cup \text{MSIntEve}, \text{AMS})$$

To remove the gas and engine events, we apply monotonicity of hiding (HideMono), transitivity of refinement (RefTrans), and hiding (HideRule).

$$\text{MSRef}: \text{ THEOREM (Mine // Control)} - \text{MSIntEve} \Rrightarrow \text{MineSystem}$$

END ExplEx

6.4 Transformation into a Discrete Interface

To obtain a discrete interface, we assume here that the control program communicates with sensors and actuators by means of shared registers. This communication mechanism is axiomatized in theory RegisterComm, expressing that reading a register yields the last written value.

$\text{RegisterComm}[\text{Registers}: \textsc{type}, \text{Values}: \textsc{nonempty\_type}, \text{Time}: \textsc{type} \ldots]: \textsc{theory}$
$\quad \textsc{begin}$
$\quad \textsc{assuming}\ldots$
$\quad \textsc{importing}\ \text{SemPrim}[\text{Time}, <, \leq]$

$\quad \text{reg}: \textsc{var}\ \text{Registers}$
$\quad \text{val}, \text{val0}, \text{val1}, \text{val2}: \textsc{var}\ \text{Values}$

$\quad \text{read}(\text{reg}, \text{val}): \text{Events}$
$\quad \text{write}(\text{reg}, \text{val}): \text{Events}$

$$\text{LastWrite}(\text{reg}, \text{val}, o, t): \text{bool} = \exists\, t_0 : t_0 \leq t \wedge o(\text{write}(\text{reg}, \text{val}))(t_0) \wedge$$
$$(\forall\ \text{val1} : \text{val1} \neq \text{val} \Rightarrow \neg o(\text{write}(\text{reg}, \text{val1}))\ \textbf{during}\ (t_0, t])$$

$$\text{NoWrite}(\text{reg}, o, t): \text{bool} = \forall\, t_0, \text{val}: t_0 \leq t \Rightarrow \neg o(\text{write}(\text{reg}, \text{val}))(t_0)$$

$\quad \text{ReadWriteAx}: \textsc{axiom}$
$$o(\text{read}(\text{reg}, \text{val}))(t) \Rightarrow \text{LastWrite}(\text{reg}, \text{val}, o, t) \vee \text{NoWrite}(\text{reg}, o, t)$$

Observe that in case of simultaneous writes a non-deterministic choice is made. Any arbitrary value is allowed if there are no preceding writes. Finally we introduce a convenient abstraction from the values read or written.

$\quad \text{read}(\text{reg}): \text{Events}$
$\quad \text{write}(\text{reg}): \text{Events}$
$\quad \text{AbstrReadAx}: \textsc{axiom}\ o(\text{read}(\text{reg}))(t) \Leftrightarrow (\exists\ \text{val}: o(\text{read}(\text{reg}, \text{val}))(t))$
$\quad \text{AbstrWriteAx}: \textsc{axiom}\ o(\text{write}(\text{reg}))(t) \Leftrightarrow (\exists\ \text{val}: o(\text{write}(\text{reg}, \text{val}))(t))$
$\textsc{end}\ \text{RegisterComm}$

Gas Sensor Assume given a gas sensor which updates a boolean register GasPresent at least once every WritePeriod time units. Moreover, assume each update reflects the current state of the gas.

$\text{ExplReg}: \textsc{theory}$
$\quad \textsc{begin}$
$\quad \textsc{importing}\ \text{ExplEx}$
$\quad \text{Registers}: \textsc{nonempty\_type}$
$\quad \text{RegValues}: \textsc{type} = \text{bool}$
$\quad \textsc{importing}\ \text{RegisterComm}[\text{Registers}, \text{RegValues}, \text{Time}, <, \leq]$

$\quad \text{GasPresent}: \text{Registers}$

$$\text{EGS}: \text{setof[Events]} = \{e \mid e = \text{gas} \vee e = \text{write}(\text{GasPresent}) \vee$$
$$\exists\ \text{val}: e = \text{write}(\text{GasPresent}, \text{val})\}$$

$\quad \text{WritePeriod}: \text{NonNegTime}$

$$\text{AGS1}: \text{Assertion} = \lambda\, o: \forall\, t: o(\text{write}(\text{GasPresent}))\ \textbf{in}\ [t, t + \text{WritePeriod}]$$

$$\text{AGS2} : \text{Assertion} =$$
$$\lambda\ o : \ \forall\ t, \text{val} : \ o(\text{write}(\text{GasPresent}, \text{val}))(t) \Rightarrow \text{val} = o(\text{gas})(t)$$

$$\text{AGS} : \text{Assertion} = \text{AGS1} \wedge \text{AGS2}$$

$$\text{GasSensor} : \text{Comps} = \text{spec}(\text{EGS}, \text{AGS})$$

Gas Sensor Control The goal is to specify a component which reads register GasPresent and controls the engine. Hence it has the following alphabet.

$$\text{ESC} : \text{setof[Events]} = \{e \mid e = \text{engine} \vee e = \text{read}(\text{GasPresent}) \vee \exists\ \text{val} : \ e = \text{read}(\text{GasPresent}, \text{val})\}$$

A possible specification of the control component could express that register GasPresent is read at least once every ReadPeriod time units.

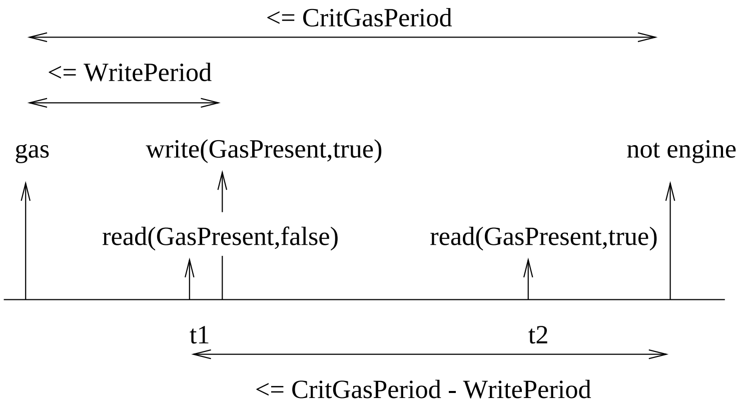
$\lambda\ o : \ \forall\ t : \ o(\text{read}(\text{GasPresent}))\ \mathbf{in}\ [t, t + \text{ReadPeriod}]$

When reading a value TRUE from this register, the engine is stopped within SensorControlDelay time units, i.e., given o, for all t,

$o(\text{read}(\text{GasPresent}, \text{TRUE}))(t) \Rightarrow \neg o(\text{engine})\ \mathbf{in}\ [t, t + \text{SensorControlDelay}]$

Then we can prove that together with GasSensor this implies Control, assuming a number of constraints on timing constants ReadPeriod and SensorControlDelay.

Again, however, we prefer to avoid the choice of internal reading periods and assumptions about internal response times. Here it is indeed possible to give a specification without introducing new timing constants. The key idea is that the response deadline should not depend on the time when GasPresent was found to be true, but on the previous read event. Observe that when gas becomes present, it might take at most WritePeriod before GasPresent is changed accordingly, and this might happen just after the previous read. Hence we have to switch the engine off within CritGasPeriod − WritePeriod after the previous read event.



In the specification below, the previous read is not mentioned explicitly, but only the preceding non-read period. To be able to stop the engine in time, the register has to be read at least once every CritGasPeriod − WritePeriod time units.

$$\text{reg}: \ \textsc{var}\ \text{Registers}$$

$$\begin{aligned}&\text{OnceReadSince}(o, \text{reg}, \text{val})(t_1, t_2): \ \text{bool} \ = \\ &\qquad t_1 < t_2 \wedge o(\text{read}(\text{reg}, \text{val}))(t_2) \wedge \neg o(\text{read}(\text{reg}))\ \textbf{during}\ (t_1, t_2)\end{aligned}$$

$$\text{OnceReadSince}(o, \text{reg})(t_1, t_2): \ \text{bool} \ = \ \exists\ \text{val}: \ \text{OnceReadSince}(o, \text{reg}, \text{val})(t_1, t_2)$$

$$\begin{aligned}\text{ASC1}: \ \text{Assertion} \ = \lambda\ o : \forall\ t : (\exists\ t_1, t_2 : t_1 < t \wedge \\ t_2 \in [t, t + \text{CritGasPeriod} - \text{WritePeriod}) \wedge \\ \text{OnceReadSince}(o, \text{GasPresent})(t_1, t_2))\end{aligned}$$

We specify that if a value TRUE is read at t_2 and any previous read event occured before t_1, then the engine is switched off before $t_1 + \text{CritGasPeriod} - \text{WritePeriod}$ and it remains off as long as no value FALSE has been read.

$$\begin{aligned}&\text{ASC2}: \ \text{Assertion} \ = \lambda\ o : \forall\ t_1, t_2, t_3 : \\ &\quad \text{OnceReadSince}(o, \text{GasPresent}, \text{TRUE})(t_1, t_2) \wedge \\ &\qquad \neg o(\text{read}(\text{GasPresent}, \text{FALSE}))\ \textbf{in}\ (t_2, t_3] \Rightarrow \\ &\qquad\quad (\exists\ t : t \in [t_2, t_1 + \text{CritGasPeriod} - \text{WritePeriod}] \wedge \\ &\qquad\qquad \neg o(\text{engine})\ \textbf{during}\ [t, t_3])\end{aligned}$$

$$\text{ASC}: \ \text{Assertion} \ = \ \text{ASC1} \wedge \text{ASC2}$$

$$\text{SensorControl}: \ \text{Comps} \ = \ \text{spec}(\text{ESC}, \text{ASC})$$

Verification of this Design Step To verify the design step, the most difficult part is to show that the assertions of the gas sensor and the sensor control lead to control assertion AC. This is expresed by lemma GSAssert, assuming WritePeriod $\leq$ CritGasPeriod. After proving OnlyDepEve(AGS, EGS) and OnlyDepEve(ASC, ESC), the parallel composition rule leads to the control specification with the exception of a number of internal events.

$$\text{GSAssert}: \ \textsc{lemma}\ \text{WritePeriod} \leq \text{CritGasPeriod} \Rightarrow \text{Valid}(\text{AGS} \wedge \text{ASC} \Rightarrow \text{AC})$$

$$\begin{aligned}&\text{GSIntEve}: \ \text{setof[Events]} \ = \\ &\ \{e \mid e = \text{write}(\text{GasPresent}) \vee (\exists\ \text{val}: \ e = \text{write}(\text{GasPresent}, \text{val})) \vee \\ &\qquad e = \text{read}(\text{GasPresent}) \vee (\exists\ \text{val}: \ e = \text{read}(\text{GasPresent}, \text{val}))\}\end{aligned}$$

$$\begin{aligned}&\text{GSPar}: \ \textsc{theorem}\ \text{WritePeriod} \leq \text{CritGasPeriod} \Rightarrow \\ &\ (\text{GasSensor}\ //\ \text{SensorControl} \Rrightarrow \text{spec}(\text{EC} \cup \text{GSIntEve}, \text{AC}))\end{aligned}$$

By the hiding rule the internal events can be removed and finally we can combine this design step with the previous one, viz. MSRef, leading to TLGSRef.

GSRef : THEOREM WritePeriod $\leq$ CritGasPeriod $\Rightarrow$
((GasSensor // SensorControl $-$ GSIntEve $\Rrightarrow$ Control))

TLGSRef : THEOREM WritePeriod $\leq$ CritGasPeriod $\Rightarrow$
((Mine // ((GasSensor // SensorControl) $-$ GSIntEve)) $-$ MSIntEve $\Rrightarrow$
MineSystem)

END ExplReg

Similarly, we could specify an actuator for the engine and develop a discrete specification of a control component. Here we only show the basic ideas of the last step where a program satisfying this specification is developed.

Platform-Independent Program Design To illustrate the basic ideas of our approach to platform-independent program design, we extend a simple imperative programming language with so-called *timing annotations*, written between brackets "[...]". These annotations contain expressions with special timing variables $m, m_1, m_2, \ldots$ which are used to record and restrict the *execution moment* of statements. The execution moment of a statement is a relevant point in time during the execution of the statement.

Example 10. Consider, as a simple example, the program

$$read(r_1, x)[\, m? \,] \,;\ \ldots \quad y := f(x) \quad \ldots \,;\ write(r_2, y)[\ > m + L \,;\ < m + U \,].$$

Annotation $m?$ expresses that the execution moment of $read(r_1, x)$ is assigned to m. The annotation of $write(r_2, y)$ expresses that its execution moment should be after $m + L$ and before $m + U$. Hence the timing annotations express the timing relations between relevant events, without further assumptions about the timing of intermediate parts.

In our simplified mine pump example, this leads to the following program.

Let $d = CritGasPeriod - WritePeriod$.

$[\, m_1 := 0 \,]$;

while *true*

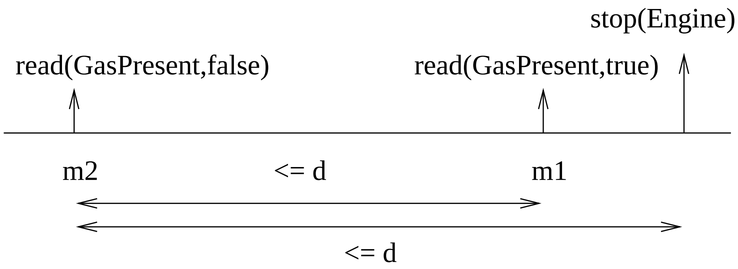
do $[\, m_2 := m_1 \,]$;

$read(GasPresent, b)[\ \leq m_2 + d, ?m_1 \,]$;

if b **then** $stop(Engine)[\ \leq m_2 + d \,]$ **fi**

od

Observe that m_2 records the previous read event (0 if there is no such event). The timing annotations require that a read event occurs within at most d time units after the previous one, and a possible stop action also takes place within this deadline, as the next picture shows.



With timing annotations there is no need to choose a reading period and, correspondingly, to determine an internal response time. Note that such a choice should depend on the platform; a short reading period gives a heavy periodic load, but a long period implies a short remaining response time and hence imposes strong requirements on the timing of the response actions.

In general, correctness of a program with timing annotations does not require any assumption about the execution platform such as the duration of statements, the number of processors, the mapping of processes to processors, and the particular scheduling policy. Extracting a timed precedence graph from the program and scheduling it on a particular platform is considered as a separate activity.

7 Concluding Remarks

We have presented two approaches for a timed semantics of parallel components:

- In the first framework (Sect. 3) an observation function of a component contains only events of its alphabet. The semantics of parallel composition can either be formulated using the union of observations plus some synchronisation constraint, or by requiring that an observation function of the parallel construct can be projected onto behaviours of the components.
- In the second framework (Sect. 4) any arbitrary behaviour outside the alphabet is included in the semantics. Then the semantics of parallel composition is defined using a simple intersection of the sets of behaviours.

A straightforward correspondence has been established between the two semantics approaches, using functions that add or remove the arbitrary events of the environment.

Observe that these approaches also have their consequences for the allowed refinements, although we used the same refinement relation for both approaches (a simple set inclusion, similar to the trace inclusion in e.g. [Old85]). For instance, in the second framework the condition in the consequence rule can be weakened.

Interesting is comparison with the refinement relation defined in [AH97], which requires that if an observation of the implementation is projected onto the alphabet of the specification, an observation of the specification is obtained.

In our framework this could be defined as

$$\text{ci1} \preceq \text{ci2} : \text{ bool } = \alpha(\text{ci2}) \subseteq \alpha(\text{ci1}) \wedge (\forall\ o : \text{ obs}(\text{ci1})(o) \Rightarrow \text{obs}(\text{ci2})(o \cap \alpha(\text{ci2})))$$

Observe that for reflexivity of $\preceq$ it is required that an observation function contains only events of the afphabet, i.e. property ObsInAlpha should hold.

As mentioned in [AH97], this relation implies P // Q $\preceq$ P. This does not hold in our first framework without environment events, but there we can hide the external events of the second component and get

$$((\text{comp1} \ // \ \text{comp2}) \ - \ (\alpha(\text{comp2}) \setminus \alpha(\text{comp1}))) \Rightarrow \text{comp1}$$

In the second framework, which contains arbitrary environment events, we have

$$\text{ce1} \ // \ \text{ce2} \Rightarrow \text{ce1}$$

since adding a parallel component simply restricts the set of behaviours.

Clearly more work is needed to investigate various notions of refinement in combination with the real-time frameworks developed here and to study their mutual relations. Also relevant is a comparison of extensions to sequential programs with a local state, similar to [Hoo94]. Another topic of future research concerns the specifications which are kept simple here, but might be more structured with, for instance, pre/post conditions or rely/guarantee (assumption/commitment) pairs.

Current work includes research on the ideas of platform-independent design of real-time systems. Although an axiomatic semantics has been given for a simple language with timing annotations [HvR97], more work is needed to obtain a formal semantics of a more realistic programming language and to incorporate this into PVS. Also the design process, supported by compositional proof rules, requires further study.

Acknowledgement

The design of the hybrid system example is strongly influenced by joint work with Onno van Roosmalen on platform-independent design.

References

[AH97] R. Alur and T. Henzinger. Modularity for timed and hybrid systems. In *Proc. Conf. on Concurrency Theory (CONCUR '97)*, pages 74–88. LNCS 1243, Springer-Verlag, 1997.

[dR85] W.P. de Roever. The quest for compositionality - a survey of assertion-based proof systems for concurrent programs, Part I: Concurrency based on shared variables. In *Proc. IFIP Working Conference 1985: The role of abstract models in computer science*, pages 181–207. North-Holland, 1985.

[HdR86] J. Hooman and W.P. de Roever. The quest goes on: a survey of proof systems for partial correctness of CSP. In *Current Trends in Concurrency*, pages 343–395. LNCS 224, Springer-Verlag, 1986.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice Hall, 1985.

[Hoo94] J. Hooman. Correctness of real time systems by construction. In *Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 19–40. LNCS 863, Springer-Verlag, 1994.

[Hoo95] J. Hooman. Verifying part of the ACCESS.bus protocol using PVS. In *Proceedings 15th Conference on the Foundations of Software Technology and Theoretical Computer Science*, pages 96–110. LNCS 1026, Springer-Verlag, 1995.

[Hoo97] J. Hooman. Verification of distributed real-time and fault-tolerant protocols. In *Algebraic Methodology and Software Technology (AMAST'97)*, to appear. LNCS 1349, Springer-Verlag, 1997.

[HvR97] J. Hooman and O. van Roosmalen. Platform-independent verification of real-time programs. In *Proceedings of the Joint Workshop on Parallel and Distributed Real-Time Systems*, pages 183–192. IEEE Computer Society Press, 1997.

[MJ96] M. Joseph, editor. *Real-time Systems: Specification, Verification and Analysis.* Prentice Hall, 1996.

[Old85] E.-R. Olderog. Process theory: Semantics, specification and verification. In *Current Trends in Concurrency*, pages 442–509. LNCS 224, Springer-Verlag, 1985.

[Old91] E.-R. Olderog. *Nets, Terms and Formulas*, volume 23 of *Camebridge Tracts in Theoretical Computer Science.* Camebridge University Press, 1991.

[ORS92] S. Owre, J. Rushby, and N. Shankar. PVS: A prototype verification system. In *11th Conference on Automated Deduction*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752. Springer-Verlag, 1992.

[ORSvH95] S. Owre, J. Rushby, N. Shankar, and F. von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, 1995.

[Sha98] N. Shankar. Machine-assisted verification using theorem proving and model checking. In M. Broy, editor, *Mathematical Programming Methodology*, to appear. 1998.

[SO98] M. Schenke and E.-R. Olderog. Transformational design of real-time systems – Part I: From requirements to program specification. *Acta Informatica*, to appear, 1998.

[VH96] J. Vitt and J. Hooman. Assertional specification and verification using PVS of the steam boiler control system. In J.-R. Abrial, E. Börger, and H. Langmaack, editors, *Formal Methods for Industrial Applications: Specifying and Programming the Steam Boiler Control*, pages 453–472. LNCS 1165, Springer-Verlag, 1996.

[Zwi89] J. Zwiers. *Compositionality, Concurrency and Partial Correctness.* LNCS 321, Springer-Verlag, 1989.