Introducing Parallel Composition to the Timed Refinement Calculus

Graeme Smith

Software Verification Research Centre
University of Queensland, Australia
smith@svrc.uq.edu.au

Abstract. The timed refinement calculus is a predicate-transformer-based formalism for the specification and refinement of real-time, reactive systems. Although it has been successfully applied to a number of case studies, its scalability and ability to effectively model concurrent and distributed real-time systems is inhibited by its lack of a suitable parallel composition operator. In particular, previous definitions of parallel composition for the formalism lack associativity or do not behave correctly when one of the components aborts. In this paper, we provide a new definition which is well-behaved under certain restrictions.

1 Introduction

The refinement calculus [3, 14] provides a formal approach to the abstract specification and stepwise development of sequential programs. A program is modelled as a *predicate transformer* which, following the approach of Dijkstra [4], transforms a required property of the final state of a program (referred to as a *postcondition*) into the property that an initial state of the program must satisfy in order to achieve that required property (referred to as the *weakest precondition*).

In order to specify real-time and concurrent systems, however, it is necessary to model *reactive programs*. Although some work has been done in the framework of the refinement calculus [2], in general, such programs cannot be described readily in terms of initial and final states alone. Hence, Mahony and Hayes [13, 10] adapted the predicate transformer approach of the refinement calculus to describe a program in terms of an *assumption* about the environment in which the program acts and the *effect* of the program under that assumption.

The major difference between the refinement calculus and the work of Mahony and Hayes is that, in the former, pre and postconditions describe the initial and final system states respectively, whereas, in the latter, assumptions and effects describe the system over all time. This allows entire system histories to be specified, making Mahony and Hayes's approach, referred to as the *timed refinement calculus*, appropriate for the specification of real-time and embedded systems.

Unfortunately, the scalability and ability to effectively model concurrent and distributed real-time systems is inhibited in the timed refinement calculus by the

lack of a suitable parallel composition operator. Although a definition for parallel composition with one-way communication (usually referred to as *piping*) exists, previous attempts at defining such an operator with bi-directional communication [11, 12, 9] have not been successful: either the operator was not associative or did not cause the composite system to abort when one of its components aborted [9]. This latter property is central to the timed refinement calculus and distinguishes it from approaches of the assumption/commitment paradigm.

In this paper, we present a new definition of parallel composition with bi-directional communication which satisfies the desired properties under certain restrictions. In Section 2, we discuss related work; specifically the assumption/ commitment approaches. In Section 3, we introduce the timed refinement calculus notation through a simple example and examine the properties of the existing piping operator. In Section 4, we build on the definition of the piping operator to present our new definition of parallel composition and illustrate its use. In Section 5, we present a refinement rule for introducing the operator into specifications.

2 Related work

Among other approaches to specifying reactive systems, the approach in this paper is closest to the *assumption/commitment* (or *rely/guarantee*) approaches used by Abadi and Lamport [1], Jones [7, 8] and the Duration Calculus [16] among others. In those approaches, a specification comprises an assumption A and a commitment C. The semantics of such a specification is defined using logical implication as $A \Rightarrow C$. Under this semantics, parallel composition with bi-directional communication can be defined simply as conjunction [1].

Consider, however, composing two components in an environment which satisfies the assumption of one of the components but not the other. Intuitively, the assumption of the composite system is not met. In the assumption/commitment approach, however, the semantics of the component whose assumption is satisfied is of the form $(\text{true} \Rightarrow E_1)$ and that of the component whose assumption is not satisfied is of the form $(\text{false} \Rightarrow E_2)$. Hence, the semantics of the composite system is $(\text{true} \Rightarrow E_1) \wedge (\text{false} \Rightarrow E_2)$ which is equivalent to $(\text{true} \Rightarrow E_1)$, i.e., the assumption of the composite system is 'satisfied'.

This is a result of the fact that the semantics of a component aborting, $(\text{false} \Rightarrow E)$, and that of a component behaving chaotically, $(A \Rightarrow \text{true})$, are not distinguished in the assumption/commitment approach. Both are logically equivalent to true. While this is adequate for many kinds of systems, e.g., distributed systems, it is not suitable for systems where a number of processes are running concurrently on a single processor. An aborting process, in this case, could perhaps overwrite another process (or its workspace) causing it to abort as well.

Adopting a predicate transformer approach in the timed refinement calculus enables us to distinguish processes which abort from those which behave chaotically. The semantics of a specification with assumption A, effect E, and

the ability to modify variables $\vec{x}$, is given by the following predicate transformer [10].

$$\lambda\, R \bullet A \wedge (\forall\, \vec{x} \bullet E \Rightarrow R)$$

That is, the specification will achieve goal R in any environment in which, firstly, the assumption A holds and, secondly, for all possible outputs $\vec{x}$ such that the effect E holds, R also holds. This is the weakest assumption which the environment must satisfy in order that effect R is obtained.

Hence, the semantics of a process aborting (A is false), i.e., $\lambda\, R \bullet$ false, is different to the semantics of a component behaving chaotically (E is true), i.e., $\lambda\, R \bullet A \wedge (\forall\, \vec{x} \bullet R)$.

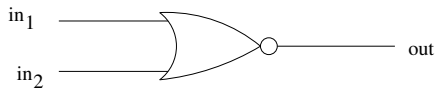
3 Timed refinement calculus

In the timed refinement calculus, predicates are specified using the Z specification language [17]. A *timed-trace* style is employed where all variables in the predicates are represented as functions of time. For example, a Boolean variable *in* (representing a digital input) is declared as follows.

$$in : \mathbb{T} \rightarrow \mathbb{B}$$

Here $\mathbb{T}$ is the set of values of absolute time and $\mathbb{B}$ is the set of Boolean values. We will assume that time is defined as the set of all non-negative real numbers, i.e., $\mathbb{T} == \mathbb{R}^+$.

Consider, for example, specifying the effect of a NOR gate [11], the circuit diagram and truth table of which is given in Figure 1.



in_1	in_2	out
F	F	T
F	T	F
T	F	F
T	T	F

Figure 1: NOR gate

Assuming the NOR gate has an operating delay of *delay* time units, its function from inputs to outputs is captured by the following function *nor*. We arbitrarily specify that the value of the output is initially false.

$$\begin{array}{|l} delay : \mathbb{T} \\ nor : ((\mathbb{T} \rightarrow \mathbb{B}) \times (\mathbb{T} \rightarrow \mathbb{B})) \rightarrow (\mathbb{T} \rightarrow \mathbb{B}) \\ \hline delay > 0 \\ \forall\, i_1, i_2 : \mathbb{T} \rightarrow \mathbb{B} \bullet \\ \quad nor(i_1, i_2) = (\lambda\, t : \mathbb{T} \bullet \textbf{if}\ t < delay\ \textbf{then}\ \text{false} \\ \qquad\qquad\qquad\qquad \textbf{else}\ \neg\, (i_1(t - delay) \vee i_2(t - delay))) \end{array}$$

In order to prove certain properties about the NOR gate specification, it may be useful to restrict the inputs to *finitely variable* signals, i.e., signals which only change a finite number of times in any finite time interval [6]. In other words, each finite interval of time consists of a (finite) sequence of intervals over which the value of the signal is not changed. This is captured by the set *fin_var*. (The details of this definition are not central to understanding this paper and are elided below.)

$$\begin{array}{|l} fin\_var : \mathbb{P}(\mathbb{T} \to \mathbb{B}) \\ \hline \ldots \end{array}$$

The form of a specification in the timed refinement calculus is based on that used by Morgan for the (sequential) refinement calculus [14]. A specification S with an assumption A, effect E and list of output variables $\vec{x}$ is expressed as follows.

$$S \mathrel{\widehat{=}} \vec{x} : [A, E]$$

The specified system S is guaranteed to provide the effect E whenever its environment satisfies assumption A. The assumption cannot constrain the output variables $\vec{x}$, so they may not occur free in A. Furthermore, the specified system must be *feasible*, i.e., it cannot constrain the input variables. Therefore, all specifications must also satisfy the feasibility constraint $A \Rightarrow (\exists \vec{x} \bullet E)$ [10]. These constraints and the absence of a notion of initial and final variable values are the main differences from Morgan's notation.

The complete specification of a NOR gate is therefore given as follows.

$$NOR \mathrel{\widehat{=}} out : [\{in_1, in_2\} \subset fin\_var, out = nor(in_1, in_2)]$$

In other words, provided that in_1 and in_2 are finitely variable then *out* will provide the desired NOR function.

The fundamental rules for sequential system refinement are weakening of the precondition and strengthening of the postcondition [3, 14]. Analogously, the fundamental rules for refinement in the timed refinement calculus are weakening of the assumption and strengthening of the effect [13, 10]. That is, given specifications S_1 and S_2, then S_1 is refined by S_2, denoted $S_1 \sqsubseteq S_2$, if, and only if, $A_1 \Rightarrow A_2$ (i.e., A_2 is at least as weak as A_1) and $A_1 \Rightarrow (\forall \vec{x} \bullet E_2 \Rightarrow E_1)$ (i.e., whenever A_1 is true, E_2 is at least as strong as E_1).

3.1 Piping

The piping operator [10] is illustrated in Figure 3. It allows communication of outputs in one direction (from the left-hand component to the right-hand one).

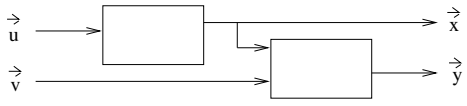


Figure 3: Piping

For generality, the communicated outputs are not hidden in the resulting composition. If hiding is desired it can be achieved using the existing hiding operator of the timed refinement calculus [10] which is defined as follows. (The brackets $[\![$ and $]\!]$ enclose the local environment in which $\vec{x}$ may be referenced.)

$$[\![\vec{x}, \vec{y} : [A, E] \setminus \{\vec{x}\}]\!] \equiv \vec{y} : [A, \exists \vec{x} \bullet E]$$

The definition of the piping operator is given by Mahony [10]. The assumption of the composition requires the assumption of the first component to be true and, for any outputs satisfying the effect of the first component, the assumption of the second component to be true. The effect of the composition is simply the conjunction of the effects of the components. The piping operator '$\gg$' is hence defined as follows.

$$\vec{x} : [A_1, E_1] \gg \vec{y} : [A_2, E_2] \equiv \vec{x}, \vec{y} : [A_1 \wedge (\forall \vec{x} \bullet E_1 \Rightarrow A_2), E_1 \wedge E_2],$$
$$\{\vec{x}\} \cap \{\vec{y}\} = \varnothing, \vec{y} \,\mathrm{nfi}\, A_1, \vec{y} \,\mathrm{nfi}\, E_1$$

In the sidecondition above, nfi means "not free in". These "not free in" conditions are necessary so that the assumption of the composite specification does not refer to its outputs.

As required in the timed refinement calculus, the composite system aborts whenever one of its components does. If the first component aborts, i.e., A_1 evaluates to false, then the assumption of the composite system is false. If A_1 is true then $\exists \vec{x} \bullet E_1$ is also true for the component $\vec{x} : [A, E]$ to be feasible. Hence, if A_2 evaluates to false in some environment due to inputs other than $\vec{x}$, the assumption of the composite system is again false.

The piping operator is also associative and monotonic with respect to refinement.

4 Parallel Composition

Our approach to defining a bi-directional parallel composition operator follows that of Müller and Scholz [15] who show how to compose specifications with bi-directional communication using two operators: piping and feedback. In this section, we first define a feedback operator and examine its properties, and then use this operator, in combination with the existing piping operator, to define our operator for parallel composition.

4.1 Feedback

A feedback operator for the timed refinement calculus is illustrated in Figure 4.

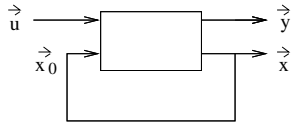


Figure 4: Feedback

Again, for generality, the outputs which form the feedback loop are not hidden by this operator.

The definition of the operator requires a subset of the inputs $\vec{x_0}$ to be identified with a subset of the outputs $\vec{x}$. The assumption of the system after the application of the feedback operator requires that all values of the outputs satisfying the effect also satisfy the assumption of the original component. In addition, it requires that the original assumption on inputs other than $\vec{x_0}$ is satisfied. This ensures that if the original component aborts in some environment due to the values of inputs other than $\vec{x_0}$ then the feedback system will also abort in that environment.

Given that inputs $\vec{x_0}$ have the same types as outputs $\vec{x}$, the operator is defined as follows.

$$[\mu\, \vec{x_0} \backslash \vec{x}] \bullet \vec{x}, \vec{y} : [A, E] \equiv \\ \vec{x}, \vec{y} : [(\exists\, \vec{x_0} \bullet A) \wedge (\forall\, \vec{x_0}, \vec{x}, \vec{y} \bullet E \wedge \vec{x} = \vec{x_0} \Rightarrow A), E \wedge \vec{x_0} = \vec{x}]$$

If the effect E constructs the values of $\vec{x}$ from $\vec{x_0}$ and these values are different from the corresponding values of $\vec{x_0}$ then it must incorporate a delay. Otherwise, conjoining E with the predicate $\vec{x_0} = \vec{x}$ will result in false, i.e., the specification will be unimplementable. This requirement for a delay in feedback systems is consistent with the results of Müller and Scholz [15].

This operator is not monotonic with respect to refinement. For example, given a timed-trace variable y, if $S_1 \mathrel{\widehat{=}} x : [x_0 = y, x = y]$ and $S_2 \mathrel{\widehat{=}} x : [x_0 = y, x = x_0]$ then since $x_0 = y \Rightarrow (\forall\, x \bullet x = x_0 \Rightarrow x = y)$, we know that $S_1 \sqsubseteq S_2$. However,

$$\begin{aligned}[\mu\, x_0 \backslash x] \bullet S_1 &= x : [(\exists\, x_0 \bullet x_0 = y) \wedge (\forall\, x_0 \bullet x_0 = y \Rightarrow x_0 = y), x = y \wedge x_0 = x] \\ &= x : [\text{true}, x = y \wedge x_0 = x]\end{aligned}$$

is not refined by

$$\begin{aligned}[\mu\, x_0 \backslash x] \bullet S_2 &= x : [(\exists\, x_0 \bullet x_0 = y) \wedge (\forall\, x_0 \bullet x_0 = x_0 \Rightarrow x_0 = y), x_0 = x] \\ &= x : [\text{false}, x_0 = x]\end{aligned}$$

It is monotonic, however, when the inputs $\vec{x_0}$ are not free in the assumption A. Fortunately, this is the situation we are most likely to encounter in practice. It would be unusual to want to make assumptions about outputs (feedback variables) that we are going to construct.

In this case $(\exists\, \vec{x_0} \bullet A) \Leftrightarrow A$ and since $\vec{x}$ and $\vec{y}$ do not occur free in A, we can deduce that $A \wedge (\forall\, \vec{x_0}, \vec{x}, \vec{y} \bullet E \wedge \vec{x_0} = \vec{x} \Rightarrow A) \Leftrightarrow A$. Hence, the operator's definition reduces to the following (obviously monotonic) definition.

$$[\mu\, \vec{x_0} \backslash \vec{x}] \bullet \vec{x}, \vec{y} : [A, E] \mathrel{\widehat{=}} \vec{x}, \vec{y} : [A, E \wedge \vec{x_0} = \vec{x}], \quad \vec{x_0}\, \text{nfi}\, A$$

4.2 Bi-directional parallel composition

Our bi-directional parallel composition operator '$\|$' is defined in terms of piping and feedback as illustrated in Figure 5.

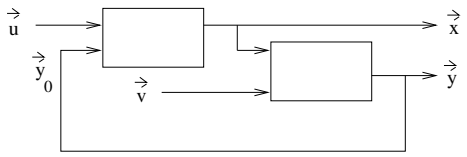


Figure 5: Bi-directional parallel composition

Building on the definition of piping, it could be defined as follows.

$\vec{x} : [A_1, E_1] \parallel \vec{y} : [A_2, E_2] \equiv [\mu\, \vec{y_0} \backslash \vec{y}] \bullet \vec{x}, \vec{y} : [A_1 \wedge (\forall\, \vec{x} \bullet E_1 \Rightarrow A_2), E_1 \wedge E_2],$
$\{\vec{x}\} \cap \{\vec{y}\} = \varnothing$

The body of the definition (occurring after the '$[\mu\, \vec{y_0} \backslash \vec{y}]\ \bullet$') is identical to the definition of piping. If $\vec{y_0}$ occurs free in the assumption of the body, the operator is not monotonic with respect to refinement for the same reason as the feedback operator.

If we place the restriction that $\vec{y_0}$ is not free in the assumption of the body then, as before, we have monotonicity. Furthermore, the definition of parallel reduces to that of piping with the extra constraint $\vec{x_0} = \vec{x}$ in the right-hand process as follows.

$$\begin{aligned}\vec{x} : [A_1, E_1] \parallel \vec{y} : [A_2, E_2] &\equiv [\mu\, \vec{y_0} \backslash \vec{y}] \bullet \vec{x}, \vec{y} : [A_1 \wedge (\forall\, \vec{x} \bullet E_1 \Rightarrow A_2), E_1 \wedge E_2] \\ &\equiv \vec{x}, \vec{y} : [A_1 \wedge (\forall\, \vec{x} \bullet E_1 \Rightarrow A_2), E_1 \wedge E_2 \wedge \vec{y_0} = \vec{y}] \\ &\equiv \vec{x} : [A_1, E_1] \gg \vec{y} : [A_2, E_2 \wedge \vec{y_0} = \vec{y}]\end{aligned}$$

Hence, the operator is also associative and the composite system aborts when one of its components aborts (since these are properties of piping).

This restriction requires that A_1 and E_1 not refer to $\vec{y_0}$ (since these predicates occur in the assumption of the body of the definition). It is too restrictive to be of any use in practice. Consider, however, the following property of the feedback operator. Given a specification $\vec{x}, \vec{y} : [A, E]$, if the effect E and the predicate $\vec{x} = \vec{x_0}$ together with the assumptions other than those on the feedback variables, i.e., $\exists\, \vec{x_0} \bullet A$, imply the assumption A, i.e., $E \wedge \vec{x} = \vec{x_0} \wedge (\exists\, \vec{x_0} \bullet A) \Rightarrow A$ then

$[\mu\, \vec{x_0} \backslash \vec{x}] \bullet \vec{x}, \vec{y} : [A, E] \equiv [\mu\, \vec{x_0} \backslash \vec{x}] \bullet \vec{x}, \vec{y} : [\exists\, \vec{x_0} \bullet A, E]$

This follows since the assumption of $[\mu\, \vec{x_0} \backslash \vec{x}] \bullet \vec{x}, \vec{y} : [A, E]$

$(\exists\, \vec{x_0} \bullet A) \wedge (\forall\, \vec{x_0}, \vec{x}, \vec{y} \bullet E \wedge \vec{x_0} = x \Rightarrow A)$

is equivalent to the assumption of $[\mu\, \vec{x_0} \backslash \vec{x}] \bullet \vec{x}, \vec{y} : [(\exists\, x_0 \bullet A), E]$

$(\exists\, \vec{x_0} \bullet (\exists\, \vec{x_0} \bullet A)) \wedge (\forall\, \vec{x_0}, \vec{x}, \vec{y} \bullet E \wedge \vec{x_0} = x \Rightarrow (\exists\, \vec{x_0} \bullet A))$

since if $E \wedge \vec{x} = \vec{x_0} \wedge (\exists\, \vec{x_0} \bullet A) \Rightarrow A$ then $E \wedge \vec{x} = \vec{x_0} \Rightarrow (\exists\, \vec{x_0} \bullet A)$ implies $E \wedge \vec{x} = \vec{x_0} \Rightarrow A$.

So in cases where $E \wedge \vec{x} = \vec{x_0} \wedge (\exists\, \vec{x_0} \bullet A) \Rightarrow A$, we can effectively remove the references to $\vec{x_0}$ in the assumption of $\vec{x} : [A, E]$ (using existential quantification) before applying the feedback operator and still obtain a specification

semantically equivalent to that which would have been obtained with the original component.

Hence an alternative definition of the parallel operator which is monotonic and associative and for which the composite system aborts when one of its components does is as follows.

$$\vec{x} : [A_1, E_1] \parallel \vec{y} : [A_2, E_2] \equiv$$
$$[\mu\, \vec{y_0} \backslash \vec{y}] \bullet \vec{x}, \vec{y} : [\exists\, \vec{y_0} \bullet A_1 \wedge (\forall\, \vec{x} \bullet E_1 \Rightarrow A_2), E_1 \wedge E_2],$$
$$\{\vec{x}\} \cap \{\vec{y}\} = \varnothing$$
$$E_1 \wedge E_2 \wedge \vec{y_0} = \vec{y} \wedge (\exists\, \vec{y_0} \bullet A_1 \wedge (\forall\, \vec{x} \bullet E_1 \Rightarrow A_2)) \Rightarrow A_1 \wedge (\forall\, \vec{x} \bullet E_1 \Rightarrow A_2)$$

The second sidecondition is simply the property $E \wedge \vec{x} = \vec{x_0} \wedge (\exists\, \vec{x_0} \bullet A) \Rightarrow A$ with A, E and $\vec{x} = \vec{x_0}$ replaced by the corresponding predicates from the operator definition.

4.3 Example

As an example of the use of the parallel composition operator, consider combining two NOR gates to form a flip-flop [11] as in Figure 6.

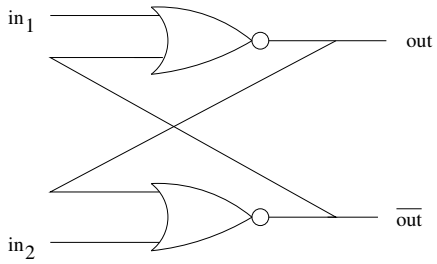


Figure 6: Circuit diagram of flip-flop.

The flip-flop can be specified in terms of piping and feedback. This can be seen more clearly if we restructure Figure 6 as shown in Figure 7.

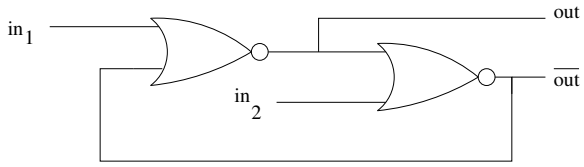


Figure 7: Alternative circuit diagram of flip-flop.

The component NOR gates are specified as follows.

$$NOR_1 \mathrel{\widehat{=}} NOR[in_2 \backslash \overline{out_0}]$$
$$NOR_2 \mathrel{\widehat{=}} NOR[in_1, out \backslash out, \overline{out}]$$

Hence, the flip-flop is specified as $\mathit{FlipFlop} \mathrel{\widehat{=}} NOR_1 \parallel NOR_2$ provided that the sideconditions for the use of the parallel operator hold.

The first sidecondition is trivially true. The antecedent of the second sidecondition

$$
\begin{array}{l}
out = nor(in_1, \overline{out_0}) \wedge \overline{out} = nor(out, in_2) \wedge \overline{out_0} = \overline{out} \\
(\exists\, \overline{out_0} : \mathbb{T} \rightarrow \mathbb{B} \bullet \\
\quad \{in_1, \overline{out_0}\} \subset \mathit{fin\_var} \wedge \\
\quad (\forall\, out : \mathbb{T} \rightarrow \mathbb{B} \bullet out = nor(in_1, \overline{out_0}) \Rightarrow \{in_2, out\} \subset \mathit{fin\_var}))
\end{array}
$$

implies $out = nor(in_1, \overline{out}) \wedge \overline{out} = nor(out, in_2) \wedge \{in_1, in_2\} \subset \mathit{fin\_var}$, since when in_1 and $\overline{out_0}$ are finitely variable then $nor(in_1, \overline{out_0})$ is finitely variable. (This follows directly from the definition of *nor*.)

If in_1 and in_2 are finitely variable, then the NOR gate outputs, out and $\overline{out}$, must also be finitely variable. This is a result of these outputs having a fixed initial value and only being able to change *delay* time units after an input has changed. (Without a delay in the NOR gates, whenever in_1 and in_2 were both false, out and $\overline{out}$ would have opposite values, but could change their values simultaneously at any time. Hence, they would not necessarily be finitely variable.)

Hence the above predicate implies $\{out, \overline{out}, in_1, in_2\} \subset \mathit{fin\_var}$ which trivially implies the consequent of the second sidecondition

$$
\begin{array}{l}
\{in_1, \overline{out}\} \subset \mathit{fin\_var} \wedge \\
(\forall\, out : \mathbb{T} \rightarrow \mathbb{B} \bullet out = nor(in, out) \Rightarrow \{in_2, out_2\} \subset \mathit{fin\_var})
\end{array}
$$

5 Refinement

Given the definition of parallel composition of Section 4, the following rule is useful for introducing it during refinement.

$$
\frac{\vec{x}, \vec{y} : [A, E_1 \wedge E_2]}{\vec{x} : [A, E_1] \parallel \vec{y} : [A, E_2]} \quad [\, \{\vec{x}\} \cap \{\vec{y}\} = \varnothing \,]
$$

This rule is valid since $A \wedge (\forall\, \vec{x} \bullet E_1 \Rightarrow A)$ is equivalent to A and hence the second sidecondition for the parallel composition operator reduces to $E_1 \wedge E_2 \wedge \vec{y_0} = \vec{y} \wedge (\exists\, \vec{y} \bullet A) \Rightarrow A$. Since $\vec{y}$ are not free in A (since outputs cannot be free in assumptions), this reduces to $E_1 \wedge E_2 \wedge \vec{y_0} = \vec{y} \wedge A \Rightarrow A$ which is trivially true.

6 Conclusion

We have defined a bi-directional parallel composition operator for the timed refinement calculus and shown how it can be used to construct specifications from component specifications. We have also presented a refinement rule which allows it to be introduced during specification development. The operator is monotonic with respect to refinement and associative and results in a composite system aborting when any one of its components aborts.

Acknowledgements

Thanks to Colin Fidge and Ian Hayes for many helpful comments on this work. This work is funded by Australian Research Council grant number A49801500: *A Unified Formalism for Concurrent Real-Time Software Development.*

References

1. M. Abadi and L. Lamport. Composing specifications. In J.W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Stepwise Refinement of Distributed Systems*, volume 430 of *LNCS*, pages 1–41. Springer-Verlag, 1990.
2. R.-J. Back. Refinement calculus, part II: Parallel and reactive programs. In J.W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Stepwise Refinement of Distributed Systems*, volume 430 of *LNCS*, pages 67–93. Springer-Verlag, 1990.
3. R.-J. Back and J. von Wright. *Refinement Calculus: A Systematic Introduction.* Graduate Texts in Computer Science. Springer-Verlag, 1998.
4. E.W. Dijkstra. *A Discipline of Programming.* Prentice Hall, 1976.
5. C.J. Fidge, P. Kearney, and A.P. Martin. Applying the Cogito program development environment to real-time system design. In C. McDonald, editor, *Australasian Computer Science Conference (ACSC'98)*, pages 367–378. Springer, 1998.
6. M.R. Hansen and Zhou Chaochen. Duration calculus: Logical foundations. *Formal Aspects of Computing*, 9(3):283–330, 1997.
7. C.B. Jones. Tentative steps toward a development method for interfering programs. *ACM Transactions on Programming Languages and Systems*, 5(4):596–619, October 1983.
8. C.B. Jones. Interference resumed. In P. Bailes, editor, *Engineering Safe Software*, pages 31–56. Australian Computer Society, 1991.
9. K. Lermer. A parallel operator for real-time processes with predicate transformer semantics. In J.-P. Katoen, editor, *AMAST Workshop on Real-Time and Probabilistic Systems (ARTS '99)*, volume 1601 of *LNCS*, pages 152–171. Springer-Verlag, 1999.
10. B.P. Mahony. *The Specification and Refinement of Timed Processes.* PhD thesis, Department of Computer Science, University of Queensland, 1992.
11. B.P. Mahony. Using the refinement calculus for dataflow processes. Technical Report 94-32, Software Verification Research Centre, University of Queensland, October 1994.
12. B.P. Mahony. Networks of predicate transformers. Technical Report 95-5, Software Verification Research Centre, University of Queensland, February 1995.
13. B.P. Mahony and I.J. Hayes. A case-study in timed refinement: A mine pump. *IEEE Transactions on Software Engineering*, 18(9):817–826, September 1992.
14. C.C. Morgan. *Programming from Specifications.* Prentice Hall, 1990.
15. O. Müller and P. Scholz. Functional specification of real-time and hybrid systems. In O. Maler, editor, *Hybrid and Real-Time Systems (HART'97)*, volume 1201 of *LNCS*. Springer-Verlag, 1997.
16. E.-R. Olderog, A.P. Ravn, and J.U. Skakkebaek. Refining system requirements to program specifications. In C. Heitmeyer and D. Mandrioli, editors, *Formal Methods for Real-Time Computing*, volume 5 of *Trends in Software*, chapter 5, pages 107–134. Wiley, 1996.
17. J.M. Spivey. *The Z Notation: A Reference Manual.* Prentice Hall, 2nd edition, 1992.