A Foundation for Formal Reuse of Hardware

Ana C. V. de Melo and Howard Barringer

Department of Computer Science
University of Manchester
Manchester M13 9PL – UK
e–mail: {melo,howard}@cs.man.ac.uk
fax: +44 161 275 6204

Abstract. This paper presents a basis for the formal reuse of hardware components as a strategy to reduce the task of verifying new hardware elements. Assuming the existence of a library of formally verified hardware components, we propose to make effective reuse of these existing elements when creating new ones. The strategy used is to formally create an *interface element* with which an existing hardware component is composed in order to implement a new desired component. In doing so, the verification task of the whole system is reduced to verifying the *interface element.*
Keywords: Reusability, High–Level Synthesis, Formal Methods, Process Algebras, Bisimulation, Interface Equation.

1 Introduction

The complexity of digital systems has grown rapidly over the past decade with the developments in the technologies for their fabrication. The problem of verifying systems in the large has been a challenging task due to the computational problem involved. Although the widespread application of hardware components requires a variety of elements which are sometimes similar to components already designed, many of these similar elements are today redesigned (and re-verified) due to the absence of mechanisms to recognize their similarities. The problem of verification in the large can then be alleviated if *verified components* are formally reused.

The application of formal methods has now produced a number of hardware components that are formally verified. Having such elements stored in a library, they can be reused to build new components that are similar to them in some sense. If the communication between the system elements not found in the library and the reused elements is formally stated, only those non–reused elements are required to be verified. In doing so, the problem of verifying the overall system is reduced to the verification of the non–existing components.

Informally, hardware design has been reused by the time the component is produced in bulk, or from libraries of standard elements when designers know the functionality of certain elements they wish to reuse. A wide, and more formal, form of reuse hardware components has been addressed by techniques to synthesize hardware design. Synthesis of hardware design has now been treated

at the various levels of hardware description, from system down to logic level [20]. Formal methods have been employed to provide verification of hardware such as HOL Proof System [15] and Circal [21], but very little synthesis–related work has been developed [6].

Despite being a kind of reuse, the system–level synthesis methods [13] do not cover all the ways existing components can be reused. The system–level synthesis is very concerned with *partitioning* of systems by adding structure to behavioural objects (in general, the systems are partitioned into standard components). By contrast, reusable components might not fit a natural partitioning of the desired components; a library component could, for example, embed some functionality not required by the specification element and still be useful for reuse. System–level synthesis is indeed an instance of reusability in the sense that the existing component could eventually be a subsystem of the desired one.

Apart from the restricted form of reuse addressed by synthesis, not much attention has been paid to the reuse of complex hardware components. In the current stage of hardware development, high–level specifications of hardware are similar to software specifications. So, the reuse techniques developed for software are also applicable to hardware development. But even in the software domain, where the problem of reuse was first addressed by McIlroy [19], only a few works use a formal approach [11, 14, 18, 35, 28, 36].

This paper presents a foundation for formal reuse of hardware design[1] based on an *interface approach*. That is, an *interface component* is formally created to be composed with the existing one in order to provide the behaviour of the desired element. A process algebra is used for the behavioural representation of hardware, and as a foundation for formally reasoning about the reuse of hardware design. The *interface component* is then constructed via application of a *decomposition operator* created for the particular process algebra. Section 2 introduces the process algebra used for representing hardware components; Sect. 3 presents the interface approach in terms of the process algebra; Sect. 4 presents the *decomposition operator*; and Sect. 5 shows minimization of interface processes by examples.

2 Representing Hardware Components using a Process Algebra

A formal approach for reusing hardware components requires a semantic definition of the representations of these components. EPA [3] is a synchronous process algebra based on SCCS [22] and originally created for describing the formal foundations of ELLA [23]. It is used in the present work as a model for representing hardware components, and as a mathematical foundation for hardware verification. In this section we only present the features of EPA which are used in the current work.

[1] This work is embedded in a project for formal reuse of hardware components using ELLA [23].

Processes and Ports of Communication. EPA processes are essentially reactive elements that transform an input event into an output one. They have a predefined set of input and output ports of communication through which the information is interchanged with the environment. The EPA *process definition* and *call* have then typed parameters to represent these ports of communication. For example, the elementary hardware component boolean–delay,

$$\text{process } DelBool(x\colon bool;\ c_1\colon bool/c_2\colon bool) \ \underline{\triangle}\ c_1(\mathtt{t})/c_2(x) :: DelBool(\mathtt{t};\ c_1/c_2) + c_1(\mathtt{f})/c_2(x) :: DelBool(\mathtt{f};\ c_1/c_2),$$

is a process defined over three groups: x defines the "state" of process *DelBool*; $\{c_1\}$ is the set of input channels; and $\{c_2\}$ is the set of output channels. The process chooses (+) one of the initial actions depending on the input event bound to c_1, and binds the actual "state" to the output channel c_2. By undertaking an initial action, it evolves into a recursive call in which the actual input value is bound to the process "state".

The set of communication ports of a process P is defined as $\mathsf{Chs}(P)$, while the sets of input and output channels are $\mathsf{Chs}_i(P)$ and $\mathsf{Chs}_o(P)$ respectively.

Combinators. Consider P and Q are processes (or agent expressions), C is a set of channel names ($C \subseteq ChanNames$), $\mathbf{x}$ is a set of variables and α is an action ($\alpha \in Act$). The EPA combinators are as follows:

$\alpha :: P$	prefixing operator	$P \backslash C$	hiding operator
$P + Q$	binary summation	$P \upharpoonright_c C$	restriction to channels
$\square\, \mathbf{x} \cdot P$	process choice	$P \mid Q$	composition

The *prefixing* and *summation* operators have the usual semantics. The *process choice* is a generalization of the binary summation in the sense that $\mathbf{x}$ is a set of typed variables and the action to be performed depends on the actual value bound to variables $\mathbf{x}$. The *hiding* operator makes the ports of communication in C no longer observable; it hides the structure of process definitions. By contrast, *restriction to channels* makes only the channels in set C observable. The composition operator will be discussed later.

Structured Actions. EPA actions are represented by a simultaneous activation of an input and an output action. Moreover, they have an enriched structure of channels:

$$\alpha \in Act ::= \epsilon \mid \boxtimes \mid (i/o)$$
$$i, o \in IOAct ::= \epsilon \mid \boxtimes \mid io_1 \oplus io_2 \mid c(v)$$

$\boxtimes$ (*zero action*) is a distinguished action that has "no meaning" in terms of behaviour. It was introduced in EPA to give a uniform treatment for partial

behaviour and hiding [2]. ϵ is an empty action, and i and o are, respectively, input and output actions that simultaneously occur to form a single event. At the ground level, an io–action is a summation (or sequence as a short–hand – $i/o = c_1(v_1), ..., c_n(v_n)/c_{n+1}(v_{n+1}), ..., c_{n+m}(v_{n+m})$) of *primitive actions* ($c(v)$). All actions that represent process behaviour, such as (i/o) and ϵ, are called "proper" actions. $\boxtimes$ is, however, an "improper" action. The sort of actions a process P can perform is given by $\mathsf{Sort}(P)$, its initial actions are in $\mathsf{Sort}_1(P)$, and its sets of input and output actions are $\mathsf{Sort}_i(P)$ and $\mathsf{Sort}_o(P)$, respectively.

For a synchronous calculus the concept of *conflict freedom* emerges [2]:

Definition 1. Suppose action α is defined over the set of channels C_1, β is defined over C_2 and $C = C_1 \cap C_2$. Then, α is conflict free with β (denoted as α CF β) iff $\textit{res-ch}(\alpha, C) = \textit{res-ch}(\beta, C)$[2].

Composition. The EPA parallel composition provides communication of processes in a synchronous manner. Transitions of the composite process are given by a synchronous communication of actions from the component processes. The transitional semantics of such an operator is as follows:

$$\boxed{\mathsf{Product}}\ \frac{P \xrightarrow{\alpha} P', \quad Q \xrightarrow{\beta} Q'}{P \mid Q \xrightarrow{\alpha \star \beta} P' \mid Q'}$$

$$\begin{aligned} &\star : Act^2 \rightarrow Act \\ &\forall\ i_p/o_p, i_q/o_q \in Act \ \bullet \\ &\quad (i_p/o_p) \star (i_q/o_q) = ((i_p \oplus i_q) \ominus (o_p \oplus o_q))/(o_p \oplus o_q) \end{aligned}$$

A meaning of the EPA combinators is given by using a general notion of labeled transition systems [1]. A Transition system for P ($TS(P) = \{\mathsf{Sts}_P, \mathcal{A}_P, T^P, s_0^P\}$) consists of a set of states Sts_P representing the process terms, a set of labels $\mathcal{A}_P$ representing actions, a transition relation T^P over process terms which gives the semantics of the system, and an initial state s_0^P. The transition of composite processes are defined in terms of the transitions of the component elements.

Two other operators are used to define the action product: summation ($\oplus$) and subtraction ($\ominus$). Roughly speaking, summation of actions io_1 and io_2 ($io_1 \oplus io_2$) combines all subactions of io_1 and io_2 if they are *conflict free*, and returns $\boxtimes$ otherwise. On the other hand, subtraction ($io_1 \ominus io_2$) results in all elements in io_1 that have no match with elements in io_2 if they are *conflict free*.

Semantics. A variety of process behavioural equivalences have been defined for deterministic and nondeterministic processes [26, 33, 34]. In the present work, strong bisimulation [3] is adopted to provide substitutivity of components; reuse requires substitution of processes. From now on, we use the term "bisimulation" to mean "strong bisimulation".

[2] $\textit{res-ch}(\alpha, C)$ restricts action α to be observable only for channels in the set C.

Definition 2. Bisimulation $\mathcal{B}$ is a symmetric binary relation on processes such that if $(P, Q) \in \mathcal{B}$ (abbreviated by $P \sim Q$), then for every proper action $\alpha \in Act$:

1. whenever $P \xrightarrow{\alpha} P'$ then $Q \xrightarrow{\alpha} Q'$ for some Q' such that $P' \sim Q'$,
2. whenever $Q \xrightarrow{\alpha} Q'$ then $P \xrightarrow{\alpha} P'$ for some P' such that $P' \sim Q'$.

Transition graphs [1] are used for behavioural representation of processes. The present work will interchange algebraic and graphic representations depending on the problem being presented. EPA actions are graphically represented as input and output actions. The directed graph has the usual meaning: processes are represented by nodes, and arrows represent actions coming from a process to another. The graphical representation of processes showing the action values will drop the channel names for simplification; only the action values are shown as sequences of elements for input and output actions. An action like $(c_1(v_1), c_2(v_2)/c_3(v_3))$ is represented in the graph by $(v_1, v_2/v_3)$ or $(v_1 v_2/v_3)$.

Determinism. Apart from the determinism defined over input actions for EPA processes (as in Automata Theory), determinacy of processes can also be defined over the pairs of input/output actions. In order to simplify future definitions, let us define the concepts of processes deterministically defined for a particular action, and determinism of processes upon the synchronous input/output actions:

Definition 3. A process P is deterministically defined for action α (denoted as $\mathcal{D}!_\alpha(P)$) if whenever $P \xrightarrow{\alpha} P'$ and $P \xrightarrow{\alpha} P''$ then $P' \sim P''$.

Definition 4. Determinism over input/output action is a property defined over processes such that if a process P is deterministic on input/output actions (denoted as $\mathcal{D}!(P)$), then for every proper action $\alpha \in Act$, whenever $P \xrightarrow{\alpha} P'$, $\mathcal{D}!_\alpha(P)$ and $\mathcal{D}!(P')$.

3 The Reuse Interface Approach

The reuse of software/hardware design involves managerial as much as descriptive information of these elements [9, 17]. Due to the formal approach being taken for reuse in the present work, we confine ourselves to reusing the descriptive information of hardware components.

As with software reuse [10, 5, 4, 16, 30], different approaches can be adopted for hardware reuse, such as creating generators of hardware components from patterns, or defining hardware description languages that capture reusability principles. But, what we actually have, in practice, are libraries of components described in a hardware description language. So, we propose reuse of hardware design by identifying similarities between a desired and an existing component, and then automatically generating the context in which the existing component must be embedded to achieve the desired behaviour. Given a desired and an

existing process, we want to be able to "transform" the existing process into the desired behaviour.

Basically, two approaches can be adopted to transform an existing process into a desired one: modifying the design of such a component, or composing the existing process with another one to achieve the desired behaviour. The task involved in the first approach is *redesigning* an existing system into a new one. There, design of the existing process is reused to build the desired one under application of transformation rules. In the second approach, the *Interface Approach*, the existing process is reused without any change in design, but a new process (the *interface process*) must be developed. Hence, the existing process has its behaviour "transformed" by composition with the new process.

From a practical point of view, the interface approach is attractive for reuse hardware design because the hardware components are reused without redesigning. Another aspect which makes this approach attractive is the possibility of having an automatic procedure to find the interface process. But, what does "creation of an interface process" mean when the desired and existing processes are defined in a process algebra?

As we have stated above, the interface process must be composed with an existing component to provide the behaviour of the desired process. Considering that (strong) bisimulation is adopted as the behavioural equivalence relation in EPA, the resulting composition must be bisimilar to the desired process. The *Interface Equation* for this particular reuse problem (synchronous composition and hiding channels) can be defined as:

$$(P \mid X) \restriction_c \mathsf{Chs}(S) \sim S \tag{1}$$

In this equation, S is the desired process while P is the existing component, and X is the interface process to be found. Notice that the observable channels of the compound process are restricted to the set of channels of the desired process; any other communication port is internalized in this compound process.

4 Interface Processes – Correct by Construction

Since reuse has as one of the main goals to minimize the effort of developing new processes by making use of existing ones, these processes must indicate some similarity in behaviour. So, a solution for the reuse problem must provide the interface process and make the best reuse of similarities between the desired and existing processes. The strategy to yield so is find the similarities between the desired and existing processes, and then createn an interface process based on their disimilarities.

In this section we present certain similarities between synchronous processes[3] which are useful for reuse when the interface approach is adopted, and then show the construction of the interface processes via a decomposition operator.

[3] A number of similarity relations useful for reuse has been defined in [7], but we confine ourselves to showing only one of them in the present paper.

4.1 Similarities between Synchronous Processes

To compare the behaviour of processes in order to enable effective *reuse* of an existing component, various notions of similarity can be defined. To realize potential reusable processes, mechanisms other than a conventional *equality* of actions must be provided. So, a relation considering partial identification of actions is required for effective reuse of synchronous processes.

In a synchronous calculus like EPA, compound actions (actions performed by compound processes) are realized by all component processes running synchronously. Therefore, similarity of processes in a synchronous interface approach is useful only if each action of the specification can be identified with an action of the given component. Otherwise, synchronous composition of the component with the interface process would never yield the behaviour of the specification.

For EPA, in particular, actions can be "identical" in a subset of channels. For instance, comparing action $(c_1(0), c_2(1))/(c_3(0), c_4(0))$ with $(c_5(1), c_2(1))/(c_3(0), c_4(0))$, it can be noticed that they coincide in channels c_2, c_3 and c_4. These actions are then identical when restricted to the set of channels $\{c_2, c_3, c_4\}$. Scaling this comparison up to processes, we may also find processes that are identical when restricted to a set of channels. A relation for comparing processes restricted to the set of their coincident channel names is defined as follows:

Definition 5. Channel Restricted Bisimulation $\mathcal{B}_c$ (CR–Bisimulation) is a symmetric binary relation on processes such that if $(P, Q) \in \mathcal{B}_c$ (abbreviated by $P \overset{c}{\sim} Q$), then there exists a non–empty set of channels $C = (\mathsf{Chs}(P) \cap \mathsf{Chs}(Q))$ such that $P \upharpoonright_c C \sim Q \upharpoonright_c C$.

Since this relation is indexed by the set of channels of both processes that have identical names, a family of bisimulations emerges with the definition of CR–bisimulation.

Note that it is only possible to have P CR–bisimilar to Q if all input channels of P are distinct from the output channels of Q and vice–versa. If an input channel name of P coincides with an output channel name of Q, actions of both processes could never be identified with each other:

Lemma 6. *Suppose $P \overset{c}{\sim} Q$, then Q is CR–bisimilar to P only if* $\mathsf{Chs}_i(P) \cap \mathsf{Chs}_o(Q) = \emptyset$ *and* $\mathsf{Chs}_o(P) \cap \mathsf{Chs}_i(Q) = \emptyset$.

Example. Our example ranges over variants of traffic light controllers. Suppose a Major/Minor roads traffic light controller is needed when a particular component has already been designed. In this section we first present a library traffic light component, the specification of the major/minor roads process, and then show the behavioural similarity between these elements.

A Library Component. At an abstract level, a traffic light controller is a process that cycles through a trivial sequence of three light colours: green, amber and red,

and then restarts the sequence[4]. Some variations on sequences of light colours can be obtained if a particular colour is given priority. Moreover, this kind of system must receive a signal to make it change the light colour. So, at a more detailed level, the traffic light controller must be defined with at least one port of communication through which a "change colour" event is input.

Our library traffic light controller gives priority to green, and it depends on two events to change the light colour: a signal for time units, and a signal to communicate the presence or absence of cars. It has some predefined intervals during which a colour must remain and after which such a colour can be changed. A time counter synchronized with a master clock is necessary for signaling these time units. Additionally, a sensor is wired to signal the other incoming event. Thus, this traffic light controller, TLC_1, is an agent that interacts with both a sensor ($Sens$) and a time counter ($Tcnt$) processes – Fig. 1.

The sensor communicates, via channel s, the signal car when a car is detected and $\overline{car}$ when this event does not occur. TLC_1 communicates with the time counter through two channels: $setc$ to initialize such a counter when a light colour is changed, and t to receive a time unit from the counter. The interval t_f is given to control the *traffic flow*, and t_s to control the *safety* time of the system (the light amber, for example, is given for safety, then t_s is concerned with the time interval during which this kind of light must be signaled). If the time unit signaled by the time counter is within these intervals, they are signaled positive (t_s, t_f); they are negated $(\overline{t_s}, \overline{t_f})$ otherwise. And finally, the light colour is communicated to the environment through channel l. The sorts of these incoming and outcoming events are as follows:

$$\text{type } colour \triangleq \{G, A, R\} \quad \text{type } intervs \triangleq \{t_s, \overline{t_s}, t_f, \overline{t_f}\}$$
$$\text{type } scount \triangleq \{st, \overline{st}\} \quad \text{type } sensor \triangleq \{car, \overline{car}\}$$

The light must remain green for a minimum interval t_f and should only cycle to amber with the presence of cars waiting to cross the road. Furthermore, whenever a car is no longer detected, the system must choose the shortest sequence of cycles to reach the green light. The transition graph for this traffic light controller[5] is shown in Fig. 1.

Note that the light remains red for both intervals. A traffic light controller for a single light, like this one, is supposed to have an opposite traffic light running in parallel, such that their light colours do not overlap. By the time the opposite traffic light is green, this light must be red. In order to allow the opposite light to cycle through amber to red, this traffic light must remain red for a safety period.

A Major/Minor Roads Traffic Light Controller. Now, suppose we want to develop a hardware component to control a traffic light at a junction of a major

[4] This example is a variation of a traffic light controller presented in [27].

[5] The transition graphs of processes are shown in this paper to facilitate the discussion on comparison of behaviour of processes.

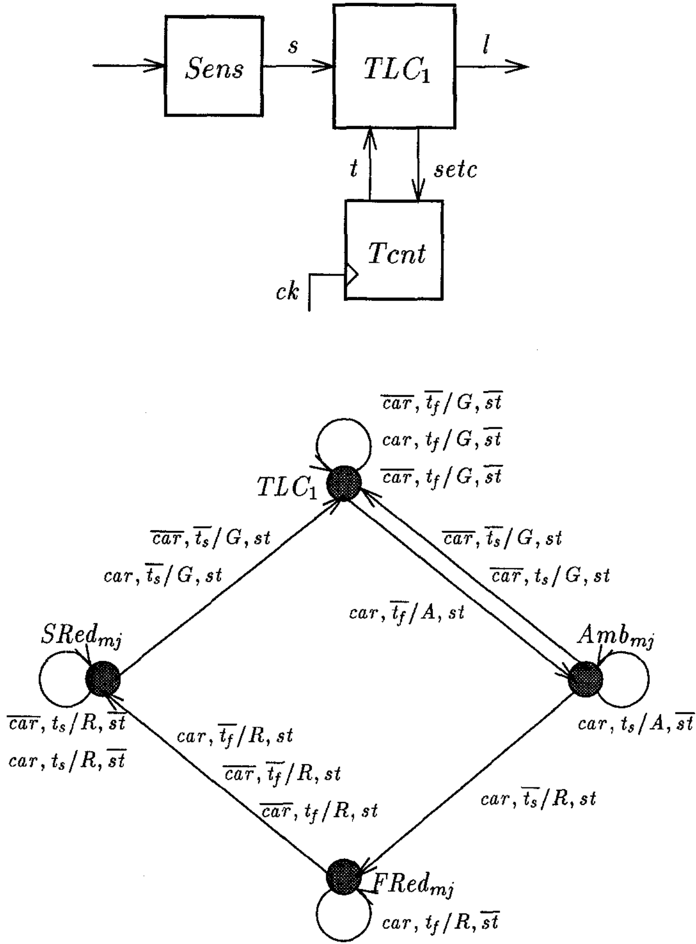

Fig. 1. Structural Definition and Transition Graph for $TLC_1(s, t/l, setc)$

and minor road. The major road is assumed to be busy and so the traffic light gives opportunity to cars coming through the minor road to cross the major road. The minor road has sensors that activate the traffic light only when a car is detected. The major road does not need any sensor because, by default, its light is green.

Thus, the major road traffic light is given priority, but not to the extent that the minor road traffic can be stalled indefinitely. The major road light can only cycle through amber to red if it has been green for a pre–established minimal period of time. Also, the minor road light can only remain green for a maximum period while the sensors indicate the presence of cars. In order to control the time cars are waiting on the minor road and also how long a red light is signaled

to the major road, a time counter must be introduced.

As with the previously defined traffic light controller, the time interval to maintain both lights amber is the safety interval $(t_s, \overline{t_s})$, and the minimum or the maximum intervals for green are specified by the traffic flow interval $(t_f, \overline{t_f})$. This major/minor roads traffic light controller has its structural and behavioural specifications shown in Fig. 2.

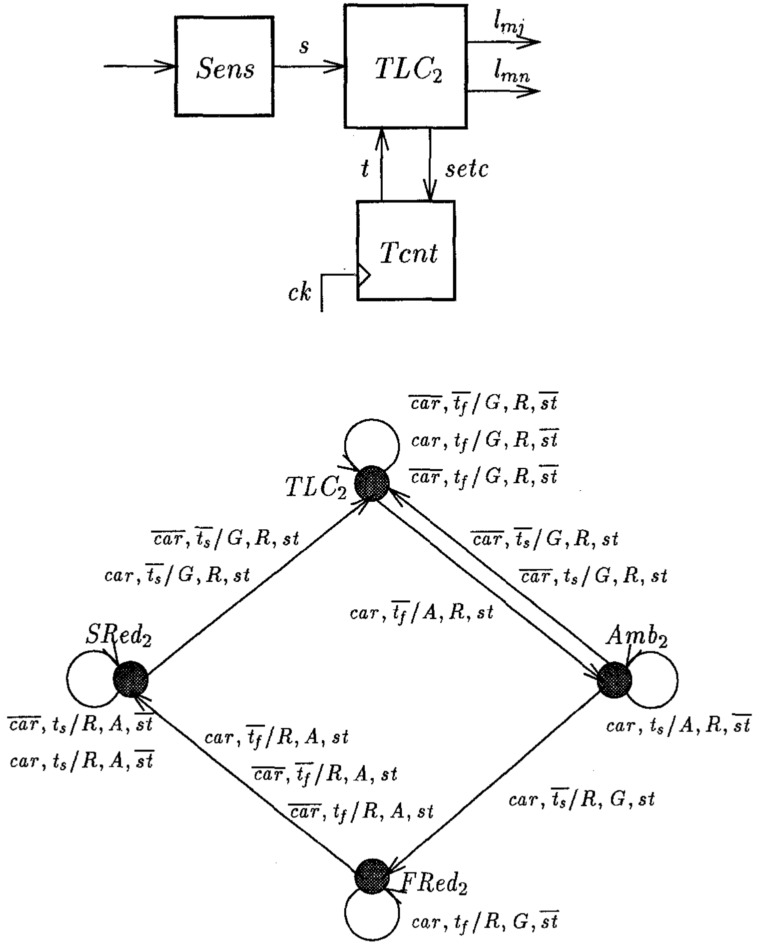

Fig. 2. Structural Definition and Transition Graph for $TLC_2(s, t / l_{mj}, l_{mn}, setc)$

Clearly, the top partitioning of this system can be defined as two processes running in parallel: a major and a minor road processes. The major road process is a controller for a single traffic light which gives priority to green, and is

activated by a sensor and a time counter. It is indeed an instance of the library component previously defined,

$$\text{process } TLC_{mj}(s\colon \mathit{sensor}, t\colon \mathit{intervs}/l_{mj}\colon \mathit{colour}, \mathit{setc}\colon \mathit{stcount}) \triangleq TLC_1(s, t/l_{mj}, \mathit{setc}),$$

which can be reused to implement this new system.

Adopting the interface approach as proposed above, an interface process must be found instead of creating the minor road process separately. But to create such an interface element, the behaviour of the whole system and the major road process must be previously compared.

The processes major/minor and major road can only be found bisimilar for a restricted set of channels: $\{s, t, l_{mj}, setc\}$. Their input channels coincide but the traffic light controller has an output channel not defined for the major road process. Restricting these processes to the set of channels above, we can realize that process TLC_{mj}, for example, can perform all the restricted actions for which TLC_2 is able to move. Transition $TLC_2 \xrightarrow{\overline{t_f},\, car/A,R,st} Amb_2$, for example, is channel restricted identical to $TLC_{mj} \xrightarrow{\overline{t_f},\, car/A,st} Amb_{mj}$. TLC_2 and TLC_{mj} have identical sets of channel restricted initial actions and, besides that, they reach processes that are also CR–bisimilar. These processes are then CR–bisimilar: $TLC_2 \overset{c}{\sim} TLC_{mj}$, $Amb_2 \overset{c}{\sim} Amb_{mj}$, $FRed_2 \overset{c}{\sim} FRed_{mj}$ and $SRed_2 \overset{c}{\sim} SRed_{mj}$.

4.2 Building Interface Processes

We are particularly interested in solving the interface equation by providing an algebraic combinator. Such solution requires the creation of a unique interface process. We must then check the theoretical limitations of reusing components using the synchronous interface approach, and define the operator.

In this section we state the constraints, the necessary and sufficient conditions to solve Equation 1, and show that the application of our decomposition operator is a solution for such an equation. An example is given to illustrate the construction of an interface process. To facilitate the discussion, we assume S as the specification process, P is the existing component, and X is the interface process[6].

Constraints. In a synchronous approach, each specification action (action from the specification component) is simulated by a synchronous execution of an existing and an interface action that are conflict free. So, for each specification action, there exists an action in the existing component which partially simulates it. Also, such an existing action must be conflict free only with actions

[6] As an abuse of notation, we use the variable X to represent a generic process that satisfies Equation 1.

from the interface process whose composition is bisimilar with the required specification action. To hold uniqueness on combination of actions, the component processes must have the following property:

Proposition 7. *Suppose* $C = \mathsf{Chs}(P) \cap \mathsf{Chs}(Q)$. *Process* P *has its initial actions uniquely combined with actions from* Q *only if, for all action* $\alpha \in \mathsf{Sort}_1(P)$, *there exists a unique* $\beta \in \mathsf{Sort}_1(Q)$ *such that* $\textit{res-ch}(\alpha, C) = \textit{res-ch}(\beta, C)$.

This property defines what is necessary for P to achieve uniqueness of conflict freedom of its initial actions with Q initial actions. For a general case, consider processes P and Q, and the set of channels $C = \mathsf{Chs}(P) \cap \mathsf{Chs}(Q)$. If $(Q \upharpoonright_c C)$ is not deterministically defined for its input/output actions that are conflict free with P, then uniqueness of combination of P with Q actions cannot be achieved since some of the Q actions could be combined with more than one action from P. Determinism of a process for its set of actions that are conflict free with another process is defined as follows:

Definition 8. Suppose $C = \mathsf{Chs}(P) \cap \mathsf{Chs}(Q)$. Q is deterministically defined for its input/output actions that are conflict free with P (denoted as $\mathcal{D}!_P(Q)$), if for every proper actions $\alpha, \beta \in Act$, whenever $Q \xrightarrow{\alpha} Q'$ and $P \xrightarrow{\beta} P'$ such that $\alpha \; \mathsf{CF} \; \beta$, then $\mathcal{D}!_\alpha(Q)$ and $\mathcal{D}!_{P'}(Q')$.

Then, the necessary condition to have P actions combined with exactly one action from Q is as follows:

Proposition 9. *Suppose* $C = \mathsf{Chs}(P) \cap \mathsf{Chs}(Q)$. P *actions are uniquely combined with* Q *actions only if* $\mathcal{D}!_P(Q \upharpoonright_c C)$.

As a consequence, a solution for the interface equation can only be found if the property defined in Proposition 9 holds for the existing and the interface processes.

Considering that the interface process X must be constructed from information of S and P corresponding actions, the P actions that are conflict free with the interface actions are also conflict free with the corresponding S action. So, the constraint $\mathcal{D}!_X(P)$ is reduced to $\mathcal{D}!_S(P)$. The property required for uniqueness of combination of X with P in order to reuse P is defined as follows:

Lemma 10. *Suppose* $S \overset{c}{\sim} P$, *then* P *can be reused to simulate* S *only if* $\mathcal{D}!_S(P)$.

In fact, if the properties in Lemma 10 hold, then the interface equation can be solved:

Corollary 11. *Suppose* $S \overset{c}{\sim} P$ *and* $\mathcal{D}!_S(P)$. *Then,* $(P \mid X) \upharpoonright_c \mathsf{Chs}(S) \sim S$ *(Equation 1) can be solved.*

Since these constraints are necessary to allow reuse, we assume the given and the desired processes satisfy them whenever an interface process is required.

The Necessary and Sufficient Conditions to Solve the Interface Equation. Bisimilarity between processes depends on the ability of both processes performing identical initial actions and then reaching bisimilar processes. Therefore, a solution for Equation 1 depends upon these premises. Let us name process Q as a possible solution for this equation. The following theorem gives the necessary and sufficient conditions to have Q as a solution for such an equation:

Theorem 12. *Assume* $S \overset{c}{\sim} P$ *and* $\mathcal{D}!_S(P)$. *Suppose* i_s/o_s, i_p/o_p *and* i_q/o_q *are proper actions. If*

1. $\mathsf{Sort}_1((P \mid Q) \restriction_c \mathsf{Chs}(S)) = \mathsf{Sort}_1(S)$, *and*
2. Q' *is a solution for* $(P' \mid X) \restriction_c \mathsf{Chs}(S) \sim S'$

whenever $Q \xrightarrow{i_q/o_q} Q'$, $S \xrightarrow{i_s/o_s} S'$, $P \xrightarrow{i_p/o_p} P'$ *and res-ch*$(i_p/o_p \star i_q/o_q, \mathsf{Chs}(S)) = i_s/o_s$, *then* Q *is a solution for* $(P \mid X) \restriction_c \mathsf{Chs}(S) \sim S$.

A proof for this theorem is included in [7].

A Decomposition Operator. Here we present our decomposition operator and claim that its use yields a particular solution for the interface equation in which minimal solutions are included.

CR–bisimulation abstract on a set of channels to compare processes. As a result, CR–bisimilar processes may coincide only in part of their behaviour. This means that neither the whole behaviour of S is substituted, nor the entire behaviour of P can be reused. The interface process must then provide the unmatched behaviour and, at the same time, enable P to be run in synchrony.

To solve the interface equation, each S action must be simulated by composition of its CR–bisimilar P action with an interface action (an action from the interface process X). We first discuss the input and output information that needs to be provided by X, and then present the decomposition operator.

The Input and Output Behaviour. To achieve bisimilarity with S, each X action must comprise sufficient information to ensure that its combination with the range of P actions exclusively provides the corresponding S action; considering $C = \mathsf{Chs}(P) \cap \mathsf{Chs}(X)$, then property $\mathcal{D}!_X(P \restriction_c C)$ must hold (Proposition 9). Such a property is accomplished, in particular, if the interface process comprises all information of P which simulates S. As a matter of creating a generic interface process, it must include the set of channels of the existing process. To do so, we must check the minimal information that has to be supplied as input and output of the interface process.

The coincident output channels of P and S denote which output behaviour of P can substitute for the S's output behaviour. But the output behaviour of S not matched to P must be supplied by the interface process. Hence, to provide the entire output behaviour of S, only those output channels of S that do not coincide with the P's need to be provided: $\mathsf{Chs}_o(S) - \mathsf{Chs}_o(P)$.

On the other hand, the external environment is only prepared to generate/accept events that interact with S – it is only able to communicate through

S input and output channels. But, to reuse P, all of its input channels must be provided in order to make it run, in synchrony, with the interface process. Those input channels of P which coincide with the S ones can be obtained directly from the environment. Those other non–coincident input channels, however, must be provided by the interface process. Thus, the output channels of the interface process must comprise at least the following elements:

$$\mathsf{Chs}_o(X) = (\mathsf{Chs}_o(S) - \mathsf{Chs}_o(P)) \cup (\mathsf{Chs}_i(P) - \mathsf{Chs}_i(S))$$

A similar analysis can be made for the input information that must be supplied by the interface process. In doing so, we find that all the S input channels that do not coincide with P's must be given as input channels of the interface process ($\mathsf{Chs}_i(S) - \mathsf{Chs}_i(P)$). This is indeed the minimal set of input channels that must be provided as interface behaviour. However, the input and output behaviour of X must comprise the whole set of P channels to hold uniqueness of conflict freedom. So, apart from the minimal set of channels, we "overconnect" the input actions of the interface process with all the coincident channels of both processes:

$$\mathsf{Chs}_i(X) = \mathsf{Chs}_i(S) \cup \mathsf{Chs}_o(P).$$

Consider $S \xrightarrow{i_s/o_s} S'$ and $P \xrightarrow{i_p/o_p} P'$ such that $S \overset{c}{\sim} P$, $S' \overset{c}{\sim} P'$, and i_s/o_s and i_p/o_p are channel restricted identical. The corresponding X action which interfaces the above S and P actions is as follows:

$$i_x/o_x = (i_s \oplus o_p)/(o_s \ominus o_p) \oplus (i_p \ominus i_s).$$

Bisimilarity of the compound process with the desired one is achieved if each compound action is identical to a desired one. In fact, as with the interface action above, the compound action is identical to the specification one:

Corollary 13. *Suppose* i_s/o_s, i_p/o_p *and* $i_x/o_x = (i_s \oplus o_p)/(o_s \ominus o_p) \oplus (i_p \ominus i_s)$ *are io–actions such that* $chs\text{-}of(i_s) \cap chs\text{-}of(o_p) = \emptyset$[7] *and* $chs\text{-}of(o_s) \cap chs\text{-}of(i_p) = \emptyset$, *then* $res\text{-}ch(i_p/o_p \star i_x/o_x, \mathsf{Chs}(i_s/o_s)) = i_s/o_s$.

Once we have checked that a single action from S and the compound process are identical when X actions are defined as above, the set of initial actions of S and the compound process can also be checked as identical. The X initial actions are defined as suggested above.

Definition 14. Assume $S \overset{c}{\sim} P$ and $\mathcal{D}!_S(P)$. Whenever $S \xrightarrow{i_s/o_s} S'$ and $P \xrightarrow{i_p/o_p} P'$ such that $S' \overset{c}{\sim} P'$ and $res\text{-}ch(i_s/o_s, C) = res\text{-}ch(i_p/o_p, C)$, then X has the transition $X \xrightarrow{i_x/o_x} X'$ for some X' such that $i_x/o_x = (i_s \oplus o_p)/(o_s \ominus o_p) \oplus (i_p \ominus i_s)$.

[7] $chs\text{-}of(io)$ denotes the set of channels defined for io.

As a consequence of property $\mathcal{D}!_S(P)$, if X initial actions are defined as above, then each of these actions can be combined only with the action from P which was used to define it:

Proposition 15. *Suppose $S \overset{c}{\sim} P$, $\mathcal{D}!_S(P)$ and X's initial actions are built up as in Def. 14. Then, for any action $i_x/o_x \in \mathsf{Sort}_1(X)$, there exists only one action $i_p/o_p \in \mathsf{Sort}_1(P)$ such that i_x/o_x CF i_p/o_p.*

Thus, equality of sets of initial actions of S and the compound process is defined as follows:

Theorem 16. *If $S \overset{c}{\sim} P$, $\mathcal{D}!_S(P)$ and X's prefixing actions are built up as in Def. 14, then $\mathsf{Sort}_1((P \mid X) \lceil_c \mathsf{Chs}(S)) = \mathsf{Sort}_1(S)$.*

The Operator. Actions of the interface process are created by taking each action from the desired process and "dividing" it by the similar action from the existing one, as suggested in Def. 14. Interface actions are therefore created for each pair of similar actions from S and P actions. Suppose i_s/o_s and i_p/o_p are CR–identical actions from S and P respectively. Then, the interface process must move via actions that are "divisions" of S by P actions:

$$\begin{aligned}
&\div : Act^2 \rightarrow Act \\
&\forall\, i_p/o_p, i_x/o_x \in Act \bullet \\
&\qquad (i_p/o_p) \div (i_x/o_x) = (i_p \oplus o_x)/(o_p \ominus o_x) \oplus (i_x \ominus i_p)
\end{aligned}$$

Since the creation of the interface actions depends on the similarities between S and P actions, as previously defined, the decomposition operator must only permit division of CR–identical actions. To do so, decomposition of process S by P must embed conditions which guarantee that only CR–identical actions are going to be divided. Using Def. 14 as basis, $(S \;/\; P) \xrightarrow{i_s/o_s \div i_p/o_p} (S' \;/\; P')$ only if $S \xrightarrow{i_s/o_s} S'$, $P \xrightarrow{i_p/o_p} P'$, $\textit{res-ch}(i_s/o_s, \mathsf{Chs}(P) \cap \mathsf{Chs}(S)) = \textit{res-ch}(i_p/o_p, \mathsf{Chs}(P) \cap \mathsf{Chs}(S))$ and $S' \overset{c}{\sim} P'$.

By the second property required to solve the interface equation (Theorem 12), the reachable processes must solve equations like $(P' \mid X) \lceil_c \mathsf{Chs}(S) \sim S'$. But it is reduced to the problem of finding a decomposed process for S' with respect to P'. So, the process reached by $S \;/\; P$ when performing that division action must be a decomposition of S' with respect to P'.

Based on these definitions of initial actions and reachable processes, a transitional semantics for the decomposition operator is as follows:

$$\boxed{\mathsf{Div}}\; \frac{S \xrightarrow{\alpha} S', \quad P \xrightarrow{\beta} P', \quad \{(S,P),(S',P')\} \subseteq \mathcal{B}_c, \quad \textit{res-ch}(\alpha, \mathsf{Chs}(P) \cap \mathsf{Chs}(S)) = \textit{res-ch}(\beta, \mathsf{Chs}(P) \cap \mathsf{Chs}(S))}{S \;/\; P \xrightarrow{\alpha \div \beta} S' \;/\; P'}$$

Due to the fact that the decomposed processes satisfies all properties of Theorem 12, the application of the decomposition operator provides a solution[8] for Equation 1.

Theorem 17. *Suppose* $S \overset{c}{\sim} P$ *and* $\mathcal{D}!_S(P)$. *Then, process* $S \;/\; P$ *is a solution for* $(P \mid X) \upharpoonright_c \mathsf{Chs}(S) \sim S$.

This operator constructs "overconnected" processes which can be minimized. In principle, the new process is created on the size of pairs of S and P that are related by CR–bisimulation. So, a further minimization must be performed based on minimal interactive processes. Also a minimization of connections must be performed to remove unnecessary inputs.

Example: The Traffic Light Interface Process. Having established a decomposition operator to solve the interface equation, we can now create the interface process to be composed with the major road process in order to yield bisimilarity with major/minor roads process. Once these processes have been checked to be CR–bisimilar and property $\mathcal{D}!_{TLC_2}(TLC_{mj})$ holds, the decomposition operator can be applied to those processes $(TLC_2 \;/ TLC_{mj})$ in order to find the interface element. As checked in Sect. 4.1,

$$\{(TLC_2, TLC_{mj}), (Amb_2, Amb_{mj}), (FRed_2, FRed_{mj}), (SRed_2, SRed_{mj})\} \subseteq \mathcal{B}_c.$$

So, for each action from TLC_2 there exists an action from TLC_{mj} which partially simulates it. For instance, action $(s(car), t(\overline{t_f})/l_{mj}(A), l_{mn}(R), setc(st))$[9] from TLC_2 is channel restricted identical to action $(s(car), t(\overline{t_f})/l_{mj}(A), setc(st))$, from TLC_{mj}, and they reach CR–bisimilar processes. The interface process, therefore, comprises an action which interfaces these actions. The interface action for the actions above is as follows:

$$(s(car), t(\overline{t_f})/l_{mj}(A), l_{mn}(R), setc(st)) \div (s(car), t(\overline{t_f})/l_{mj}(A), setc(st)) = (s(car), t(\overline{t_f}), l_{mj}(A), setc(st)/l_{mn}(R))$$

The transition graph for $TLC_{div} \triangleq TLC_2 \;/\; TLC_{mj}$ is shown in Fig. 3.

5 Minimizing the Interface Process

A solution for the interface equation requires bisimilarity with the desired process. So, this section is devoted to minimization of target concurrent processes in such a way that the final compound processes are bisimilar to their original ones[10]. Consider P_0 $(TS(P_0) = \{\mathsf{Sts}_{P_0}, \mathcal{A}_{P_0}, T^{P_0}, P_0\})$ and Q_0 $(TS(P_0) =$

[8] The soundness of the decomposition operator has been proved in [7].

[9] The channel names are introduced here to facilitate the comparison of actions.

[10] The minimization technique used here is based on minimization under *compatible states* [12, 32] and *interacting automata* [8], and is presented in [7].

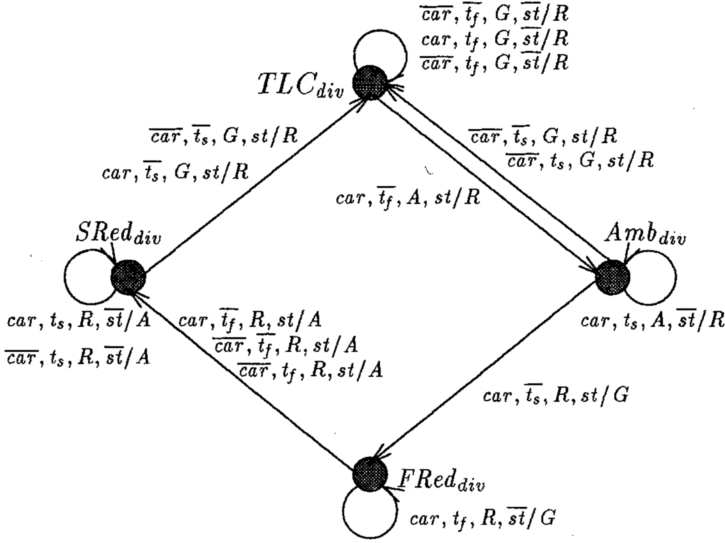

Fig. 3. Interface between the Traffic Light Controller and the Major Road Process

$\{\mathsf{Sts}_{Q_0}, \mathcal{A}_{Q_0}, T^{Q_0}, Q_0\})$ are run in parallel and restricted to the set of observable channels E_x $((P_0 \mid Q_0) \restriction_c E_x)$, and Q_0 is the target process to be minimized.

For concurrent processes (arbitrarily connected), the key point of minimization is "joining" states of the target process and still maintaining bisimilarity with the original compound process. To do so, composition of each subprocess reachable from P_0, say P_i, with a Q_0's joint process[11], say $\lfloor Q_l, Q_m \rceil$, must exclusively result in the behaviour of $((P_i \mid Q_l) \restriction_c E_x)$ or $((P_i \mid Q_m) \restriction_c E_x)$; $(P_i \mid \lfloor Q_l, Q_m \rceil) \restriction_c E_x \sim (P_i \mid Q_m) \restriction_c E_x$, or $(P_i \mid \lfloor Q_l, Q_m \rceil) \restriction_c E_x \sim (P_i \mid Q_l) \restriction_c E_x$. Since actions of the compound process are obtained by composition of conflict free actions from the component processes, P_i actions must be exclusively conflict free with Q_l or Q_m actions.

For example, the interface traffic light TLC_{div} is combined with TLC_{mj} in such a way that

$$(TLC_{mj} \mid TLC_{div}) \restriction_c \mathsf{Chs}(TLC_2), \quad (Amb_{mj} \mid Amb_{div}) \restriction_c \mathsf{Chs}(TLC_2),$$
$$(FRed_{mj} \mid FRed_{div}) \restriction_c \mathsf{Chs}(TLC_2) \text{ and } (SRed_{mj} \mid SRed_{div}) \restriction_c \mathsf{Chs}(TLC_2)$$

are the reachable processes. This means that TLC_{div} can only be joined to Amb_{div}, for example, if

[11] Suppose $\rho = \{B_1^Q, \ldots B_n^Q\}$ is a partition of Sts_Q. The process constructed from partition ρ is defined by: $\lfloor B_k^Q \rceil = \{B_k^Q \xrightarrow{\alpha} B_l^Q \mid (Q_i \xrightarrow{\alpha} Q_j) \in T^Q, \ Q_i \in B_k^Q \text{ and } Q_j \in B_l^Q\}$ and $\lfloor \rho \rceil = \{\lfloor B_k^Q \rceil \mid B_k^Q \in \rho\}$

$$(TLC_{mj} \mid TLC_{div}) \restriction_c \mathsf{Chs}(TLC_2) \sim (TLC_{mj} \mid \lfloor TLC_{div}, Amb_{div} \rceil) \restriction_c \mathsf{Chs}(TLC_2)$$

and

$$(Amb_{mj} \mid Amb_{div}) \restriction_c \mathsf{Chs}(TLC_2) \sim (Amb_{mj} \mid \lfloor TLC_{div}, Amb_{div} \rceil) \restriction_c \mathsf{Chs}(TLC_2)$$

hold. Compatibility of concurrent processes is more formally defined as follows:

Definition 18. Compatibility of Concurrent Processes (abbreviated by $\simeq_{\shortparallel}$) is a symmetric binary relation on processes such that if $Q_i \simeq_{\shortparallel} Q_j$, considering the sets of interacting channels C and external channels E_x, then for every proper action $\alpha, \beta \in Act$ such that $\alpha \in \mathsf{Sort}_1(Q_i)$, $\beta \in \mathsf{Sort}_1(Q_j)$ and $\textit{res-ch}(\alpha, C) = \textit{res-ch}(\beta, C)$:

1. whenever $Q_i \xrightarrow{\alpha} Q_i'$ then $Q_j \xrightarrow{\beta} Q_j'$ for some Q_j' such that $\textit{res-ch}(\alpha, E_x) = \textit{res-ch}(\beta, E_x)$ and $Q_i' \simeq_{\shortparallel} Q_j'$,
2. whenever $Q_j \xrightarrow{\beta} Q_j'$ then $Q_i \xrightarrow{\alpha} Q_i'$ for some Q_i' such that $\textit{res-ch}(\alpha, E_x) = \textit{res-ch}(\beta, E_x)$ and $Q_i' \simeq_{\shortparallel} Q_j'$.

Concurrently compatible processes perform identical observable actions whenever their actions equals for a restricted set of channels C, and then reach concurrently compatible processes.

Since concurrent compatibility is not an equivalence relation (it is intransitive), various partitions of a process states can be obtained under such relation. The interface traffic light process (Figure 3) can be minimized with respect to its interaction with TLC_{mj}. Considering the set of external channels $E_x = \mathsf{Chs}(TLC_2)$, the sets of concurrently compatible subprocesses reachable from TLC_{div} under composition with TLC_{mj} are as follows:

$$\{\{TLC_{div}\}, \{Amb_{div}\}, \{FRed_{div}\}, \{SRed_{div}\}, \{TLC_{div}, Amb_{div}\},$$
$$\{TLC_{div}, FRed_{div}\}, \{TLC_{div}, SRed_{div}\}, \{Amb_{div}, FRed_{div}\},$$
$$\{TLC_{div}, Amb_{div}, FRed_{div}\}\}.$$

The minimal partitions that can be constructed from those sets of concurrently compatible processes are as follows:

$$\rho_1 = \{\{TLC_{div}, SRed_{div}\}, \{Amb_{div}, FRed_{div}\}\}$$
$$\rho_2 = \{\{SRed_{div}\}, \{TLC_{div}, Amb_{div}, FRed_{div}\}\}$$

These partition elements are named as:

process $G_{mj\text{-}mn}(s\colon sensor, t\colon intervs, l_{mj}\colon colour, setc\colon stcount/l_{mn}\colon\ colour) \triangleq$
$\lfloor TLC_{div}, SRed_{div} \rceil$

process $A_{mj\text{-}mn}(s\colon sensor, t\colon intervs, l_{mj}\colon colour, setc\colon stcount/l_{mn}\colon colour) \triangleq$
$\lfloor Amb_{div}, FRed_{div} \rceil$

process $R_{mj\text{-}mn}(s\colon sensor, t\colon intervs, l_{mj}\colon colour, setc\colon stcount/l_{mn}\colon colour) \triangleq$
$\lfloor SRed_{div} \rceil$

process $GAR_{mj\text{-}mn}(s\colon sensor, t\colon intervs, l_{mj}\colon colour, setc\colon stcount/l_{mn}\colon colour) \triangleq$
$\lfloor TLC_{div}, Amb_{div}, FRed_{div} \rceil$

Apart from minimizing the number of states, ports of communication of interacting processes must also be minimized. The communication ports can be deleted as far as the possibility of communication with the existing process is maintained, and the minimal set of input and output channels (defined in Sect. 4.2) are included. So, the minimal set of channels that maintain the communication between TLC_{mj} and $G_{mj\text{-}mn}$ (or TLC_{mj} and $GAR_{mj\text{-}mn}$) is $\{l_{mn}, l_{mj}, setc\}$. Figure 4 shows the behaviour of both patitioned processes restricted to the set of channels mentioned above.

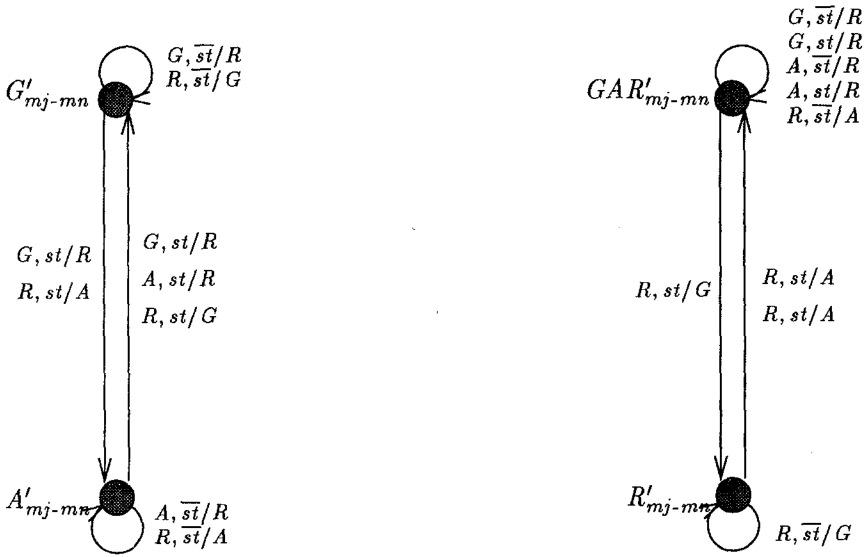

Fig. 4. The Minimal Interface Processes for the TLC

Any of these processes can be picked up as interface to construct the desired traffic light controller by reusing the major road component. It is important to

note that these processes do not represent the minor road process; they are only synchronous interfaces between the major road process and the traffic light controller. However, they are less complex than the minor road process (a four state process) and the verification of the whole system is restricted to the verification of their refinements; they are correct by construction.

6 Discussion

The reuse of hardware components through formal means is still in its infancy. Most works in the hardware area which address a kind of reuse are synthesis-based, or based on an informal approach, such as [24]. Even in the software domain, a formal approach to reuse has only been attempted by a few works that concentrate on partial definition of processes to accomplish reuse [14, 18, 35, 28, 36]. In fact, abstract definition of processes is essential to obtain effective reuse. In the present work we have assumed an existing model for representing hardware to concentrate on mechanisms to identify processes by making certain abstractions. This work has indicated that besides abstraction on representation of processes, mechanisms based on abstractions to identify processes are also useful for reuse.

Reuse undertaking the interface approach is very much concerned with the idea of decomposing the desired process into submodules, so that one of those submodules matches the existing component. This decomposition problem has already been solved, by Shields [31] and his peers [29, 25], for deterministic processes represented in CCS, taking weak bisimulation as equational equality. Here, we have presented a solution for the decomposition problem considering *nondeterministic* processes represented in EPA, and taking strong bisimulation as the equivalence relation.

In our work, we indicated the theoretical limits of reusing synchronous processes undertaking the interface approach. Such an approach depends critically on the composition operator used, and then for each process algebra the essential preconditions must be calculated accordingly. The appropriate limits have not been considered in solutions for the interface equation defined for CCS; as a result, only deterministic specification processes were considered there. It makes us believe that nondeterministic processes can be implemented by reuse of others in many process algebras as long as the behavioural similarities between processes are checked in advance.

A formal reuse of hardware design leads to a decompositional verification of processes. With the reuse of components formally verified, only the new elements being constructed need to be verified. Since composition of the interface with the existing process is proved bisimilar to the desired specification, verification of the interface element assures verification of the whole system. In fact, the topmost definition of the interface process is correct by construction, and only its refinements must be verified.

Acknowledgements. We would like to thank Alan J. Williams for his helpful comments on this paper. This work was supported by a grant from CAPES (Brazil), to whom we are indebted for the financial support.

References

1. André Arnold. *Finite Transition Systems.* International Series in Computer Science. Prentice–Hall, 1994.
2. H. Barringer, G. Gough, B. Monahan, and A. Williams. An algebraic framework for action and process. D2.3c, Formal Verification Support for ELLA, IED project 4/1/1357, University of Manchester, May 1993.
3. H. Barringer, G. Gough, B. Monahan, and A. Williams. A process algebra foundation for reasoning about core ELLA. Technical Report 94–12–1, University of Manchester, Dec 1994.
4. T. J. Biggerstaff and A. J. Perlis. *Software Reusability – Applications and Experience*, volume 2. Addison–Wesley Publishing Company, first edition, 1989.
5. T. J. Biggerstaff and A. J. Perlis. *Software Reusability – Concepts and Models*, volume 1. Addison–Wesley Publishing Company, first edition, 1989.
6. H. Busch, H. Nusser, and T. Rössel. Formal methods for synthesis. In P. Michael, U. Lauther, and P. Duzy, editors, *The Synthesis Approach to Digital System Design.* Kluwer Academic Publishers, 1993.
7. Ana C. V. de Melo. *Formal Reuse of Hardware Design.* PhD thesis, University of Manchester, March 1995.
8. Srinivas Devadas. Optimizing interacting finite state machines using sequential don't cares. *IEEE Transactions on Computer–Aided Design*, December 1991.
9. P. Freeman. A perspective on reusability. In Peter Freeman, editor, *Tutorial: Software Reusability.* IEEE – The Computer Society Press, 1987.
10. Peter Freeman. *Tutorial: Software Reusability.* IEEE – The Computer Society Press, 1987.
11. M. C. Gaudel and T. Moineau. A theory of software reusability. In *ESOP'88*, volume 300 of *LNCS.* Springer Verlag, 1988.
12. Arthur Gill. *Introduction to the Theory of Finite–State Machines.* McGraw–Hill Book Company, first edition, 1962.
13. W. Glunz, A. Pyttel, and G. Venzl. System–level synthesis. In P. Michael, U. Lauther, and P. Duzy, editors, *The Synthesis Approach to Digital System Design.* Kluwer Academic Publishers, 1993.
14. Joseph A. Goguen. Principles of parameterized programming. In T. J. Biggerstaff and A. J. Perlis, editors, *Software Reusability – Concepts and Models,* volume 1. Addison–Wesley Publishing Company, 1989.
15. M. J. Gordon. Why higher–order logic is a good formalism for specifying and verifying hardware. In G. J. Milne and P. A. Subrahmanyam, editors, *Formal Aspects of VLSI Design.* North–Holland, 1986.
16. J. W. Hooper and R. O. Chester. *Software Reuse – Guidelines and Methods.* Software Science and Engineering. Plenum Press, 1991.
17. T. C. Jones. Reusability in programming: A survey of the state of the art. In Peter Freeman, editor, *Tutorial: Software Reusability.* IEEE – The Computer Society Press, 1987.

18. Shmuel Katz, Charles A. Richer, and Khe-Sing The. PARIS: A system for reusing partially interpreted schemas. In T. J. Biggerstaff and A. J. Perlis, editors, *Software Reusability – Concepts and Models*, volume 1. Addison–Wesley Publishing Company, 1989.
19. M. D. McIlroy. Mass–produced software components. In J. M. Buxton, P. Naur, and B. Randell, editors, *Software Eng. Concepts and Techniques –1968 NATO Conf. Software Eng.* Petrocelli/Charter, 1976.
20. P. Michael, U. Lauther, and P. Duzy. *The Synthesis Approach to Digital System Design.* Kluwer Academic Publishers, 1993.
21. George Milne. *Formal Specification and Verification of Digital Systems.* McGraw–Hill, first edition, 1994.
22. Robin Milner. *Communication and Concurrency.* Prentice–Hall, first edition, 1989.
23. J D Morison and A S Clarke. *ELLA2000: A Language for Electronic System Design.* McGraw-Hill, 1993.
24. N. S. Nagaraj. OPSYN – OASYS based pseudosynthesis tool. In *VLSI Design 1992 – The fifth International Conference on VLSI Design.* IEEE – The Computer Society Press, 1992.
25. Joachim Parrow. Submodule construction as equation solving in CCS. *Theoretical Computer Science*, 68:175–202, 1989.
26. Lucia Pomello. Some equivalence notions for concurrent systems. an overview. In *Advances in Petri Nets 1985*, volume 222 of *LNCS.* Springer–Verlag, 1985.
27. Franklin P. Prosser and David E. Winkel. *The art of Digital Design.* Pentice Hall International Editions, second edition, 1987.
28. Noah S. Prywes and Evan D. Lock. Use of the model equational language and program generator by management professionals. In T. J. Biggerstaff and A. J. Perlis, editors, *Software Reusability – Applications and Experience*, volume 2. Addison–Wesley Publishing Company, 1989.
29. Huajan Qin and Philip Lewis. Factorization of finite state machines under observational equivalence. In J. C. M. Baeten and J. F. Groote, editors, *CONCUR'91*, volume 527 of *LNCS.* Springer–Verlag, 1991.
30. W. Schäfer, R. Prieto-Díaz, and M. Matsumoto. *Software Reusability.* Ellis Horwood, 1994.
31. M. W. Shields. Implicit system specification and the interface equation. *The Computer Journal*, 32(5), 1989.
32. Stephen H. Unger. *Asynchronous Sequential Switching Circuits.* Wiley–Interscience, first edition, 1969.
33. R. J. van Glabbeek. The Linear Time – Branching Time Spectrum. In *CONCUR'90*, volume 458 of *LNCS.* Springer–Verlag, 1990.
34. R. J. van Glabbeek. The Linear Time – Branching Time Spectrum II. In *CONCUR'93*, volume 715 of *LNCS.* Springer–Verlag, 1993.
35. Dennis M. Volpano and Richard B. Kieburtz. The templates approach to software reuse. In T. J. Biggerstaff and A. J. Perlis, editors, *Software Reusability – Concepts and Models*, volume 1. Addison–Wesley Publishing Company, 1989.
36. M. Wirsing, R. Hennicker, and R. Stahl. Menu – an example for the systematic reuse of specifications. In *2nd European Software Engineering Conference*, volume 387 of *LNCS.* Springer Verlag, 1989.