A Formalism for the Synthesis of Efficient Controllers for Discrete Event Systems

Antonio Camurri and Alessandro Coglio
DIST – Dipartimento di Informatica Sistemistica e Telematica
Università di Genova
Via Opera Pia 13, I-16145 Genova, Italy
{music, tokamak}@dist.unige.it

***Abstract* – We propose a formalism for the synthesis of "functions", called *computation boxes* (*C-boxes*), which possess memory: arguments are put into them and results are gotten from them, avoiding unnecessary computations and possibly using previously calculated intermediate results. C-boxes may be "functionally" composed in graph-like structures, thus allowing modularity and reusability. Different languages may be used to implement computations. C-boxes may be externally specified by means of purely functional specifications called *extended functions* (*E-functions*), without bothering about their internal structure.**

These characteristics make C-boxes particularly well-suited to be used as a software engineering formalism to synthesize efficient controllers for Discrete Event Systems in a clear and modular way.

I. INTRODUCTION

In this paper, a Discrete Event System (DES) is a triple $S = \langle X,E,T\rangle$, where X is a set of states, E is a set of events, and $T \in [X \times E \rightarrow_p X]$ is a transition function ($[X \times E \rightarrow_p X]$ is the set of all partial functions from $X \times E$ to X). For example, Petri Nets [1] and Colored Petri Nets [2] are DESs, with markings being states, transition firings being events, and firing rules defining transition functions. DESs, in particular (Colored) Petri Nets, are widely used in the automated manufacturing field, as discrete event controllers for plants [3].

In this paper, a controller for a DES $S = \langle X,E,T\rangle$ is a function $C \in [X \rightarrow 2^E]$ (2^E is the set of all subsets of E; $[X \rightarrow 2^E]$ is the set of all functions from X to 2^E): for each $x \in X$, controller C selects the subset of all events e enabled in state x (i.e. such that $T(x,e)$ is defined) which belong to $C(x)$; more precisely, DES S and controller C together define a closed-loop DES $S' = \langle X,E,T'\rangle$ such that for each state x and event e, $T'(x,e)$ is defined iff $T(x,e)$ is defined and $e \in C(x)$, and in that case $T'(x,e) = T(x,e)$. In other words, controller C constrains the behavior of S by allowing only some events to occur in each state. This approach to the control of DESs is slightly different from others found in literature: in [4] a supervisor (i.e. controller) is defined as a function mapping a string of events generated by the DES to a subset of E; our approach is motivated below.

It is sometimes useful to design a complex DES $S = \langle X,E,T\rangle$ in terms of another DES $S_0 = \langle X,E,T_0\rangle$ and a controller $C \in [X \rightarrow 2^E]$ for S_0, where S is the closed-loop DES defined by S_0 and C. For example, consider the design of a Colored Petri Net as discrete event controller for a large plant where uses of resources (machines, robots, etc.) by productive processes interact in complex ways, thus being subject to many potential deadlocks. In order to avoid such deadlocks, resources must be given to productive processes (e.g. put a part into a machine) according to the state of many other resources (e.g. which productive processes are holding them): so, a deadlock-free Net would have lots of arcs connecting places and transitions for the sole purpose of checking the states of some resources, thus making unclear real uses of resources by productive processes. It might be easier and clearer to design a non-deadlock-free Net which only deals with real uses of resources, and then constrain the behavior of the Net to avoid deadlocks by means of a controller mapping each marking into a set of transition firings which do not lead to deadlocked markings.

The implementation of a controller for a DES would reasonably consist in a functional procedure taking a state as argument and returning a set of events as result; when an event occurs and state changes the procedure is called upon the new state, and this is repeated over and over again. It is very often the case that the state of a DES can be naturally partitioned into components (e.g. the components of the marking of a (Colored) Petri Net are the markings of the individual places), and that when an event occurs only a few components change, the others remaining unaltered (e.g. when a transition fires in a (Colored) Petri Net, only the places connected to that transition change their markings): so, after an event occurs, many of the computations executed upon the previous state are equally repeated upon the new state, thus wasting time.

In this paper we propose a formalism for the synthesis of "functions", called *computation boxes* (*C-boxes* for short), which possess memory: arguments are put into C-boxes (i.e. written into memory) and results are gotten from C-boxes (i.e. read from memory). C-boxes write results into their memory by reading arguments from their memory and executing computations upon them. When getting results, computations which are unnecessary to determine such results may be avoided; furthermore, since memory may also contain intermediate results, when getting results after changing only some arguments, some computations may be avoided by using previously calculated intermediate results which are still valid. C-boxes may be "functionally" composed together in graph-like structures to build more complex C-boxes, thus allowing modularity and reusability. Computations are specified as generic mathematical functions, so that they may be implemented in any language, thus allowing the use of different languages in a clear way. An extremely important property of C-boxes is that they may be externally specified by means of purely functional specifications called *extended functions* (*E-functions* for short): in order to compose C-boxes together or to use them putting arguments and getting results, it is sufficient to know their functional specifications, without bothering about the internal structure.

These characteristics make C-boxes particularly well-suited to be used as a software engineering formalism to synthesize efficient controllers for DESs in a clear and modular way.

II. OVERVIEW OF THE FORMALISM

Since a complete formal presentation of our formalism would make this paper exceedingly long, we only give an overview of it by means of some simple examples; for details, see [5].

Firstly, a *type* is a non-empty set of objects not including $\perp$ ("nil"). In our simple examples we use the following types: $\mathcal{Z} = \{\dots,-2,-1,0,1,2,\dots\}$, the set of integers; $\mathcal{B} = \{T,F\}$, the set of booleans; $\mathcal{KEY}$, a set of *keys* (identifiers, names, or whatever); $\mathcal{TABLE}$, the set of *tables*, where a table is a finite set of entries $\langle key,val\rangle \in \mathcal{KEY} \times \mathcal{Z}$ such that no two distinct entries in the table have the same key.

Given types $\tau_1,\dots,\tau_n$ $(n \geq 1)$ and $\tau^1,\dots,\tau^m$ $(m \geq 1)$, a *computation box* (*C-box* for short) from $\langle\tau_1,\dots,\tau_n\rangle$ to $\langle\tau^1,\dots,\tau^m\rangle$ is a *level-u computation box* (*C^u-box* for short) from $\langle\tau_1,\dots,\tau_n\rangle$ to $\langle\tau^1,\dots,\tau^m\rangle$, for some $u \in \mathcal{N}$ ($\mathcal{N}$ is the set of natural numbers).

A. Level-0 Computation Boxes

Fig. 1, Fig. 2 and Fig. 3 respectively depict C^0-boxes *search* from $\langle\mathcal{TABLE},\mathcal{KEY}\rangle$ to $\langle\mathcal{B},\mathcal{Z}\rangle$, *if* from $\langle\mathcal{B},\mathcal{Z},\mathcal{Z}\rangle$ to $\mathcal{Z}$ and *plus* from $\langle\mathcal{Z},\mathcal{Z}\rangle$ to $\mathcal{Z}$. A C^0-box from $\langle\tau_1,\dots,\tau_n\rangle$ to $\langle\tau^1,\dots,\tau^m\rangle$ basically consists of n *argument memories* (*A-mems* for short), m *result memories* (*R-mems* for short) and a tree. Each A-mem or R-mem (represented as a rectangle) contains an object of the associated type or the special value $\perp$. Each node of the tree is labeled by one of $Arg_1,\dots,Arg_n,Res_1,\dots,Res_m$; each branch connecting two nodes is labeled by a *constraint*, i.e. a subset of $\tau_{j_1} \times \dots \times \tau_{j_q}$, where $Arg_{j_1},\dots,Arg_{j_q}$ are the labels among $Arg_1,\dots,Arg_n$ of the nodes preceding the branch, in that order; each node labeled by a Res_i is also labeled by a *calculation*, i.e. a mathematical function from $\tau_{j_1} \times \dots \times \tau_{j_q}$ to $\tau^i \cup \{\perp\}$, where $Arg_{j_1},\dots,Arg_{j_q}$ are the labels among $Arg_1,\dots,Arg_n$ of the nodes preceding the node, in that order. In Fig. 1 *constF* is the function from $\mathcal{TABLE}$ to $\mathcal{B} \cup \{\perp\}$ such that $constF(tbl) = F$, *is-present?* is

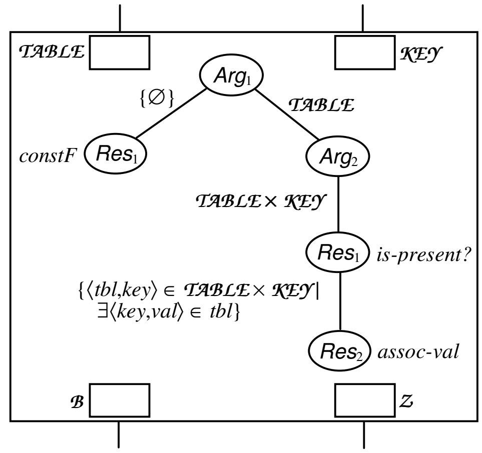

Fig.1 C^0-box *search*.

the function from $\mathcal{TABLE} \times \mathcal{KEY}$ to $\mathcal{B} \cup \{\perp\}$ such that $is\text{-}present?(tbl,key) =$ **if** $\exists\langle key,val\rangle \in tbl$ **then** T **else** F, *assoc-val* is the function from $\mathcal{TABLE} \times \mathcal{KEY}$ to $\mathcal{Z} \cup \{\perp\}$ such that $assoc\text{-}val(tbl,key) =$ **if** $\exists\langle key,val\rangle \in tbl$ **then** val **else** $\perp$; in Fig. 2 *copy-int* is the function from $\mathcal{B} \times \mathcal{Z}$ to $\mathcal{Z}$ such that $copy\text{-}int(b,val) = val$; in Fig. 3 *add* is the function from $\mathcal{Z} \times \mathcal{Z}$ to $\mathcal{Z}$ such that $add(val_1,val_2) = val_1 + val_2$.

C^0-boxes can be *executed*, as we now explain by means of *search* as an example (see Fig. 1). Execution starts at the root of the tree: label Arg_1 means that the first argument (i.e. the object in the first A-mem) must be *read*. If it is $\perp$ execution stops ($\perp$ represents a "missing value"), and can then be resumed (see next subsection); otherwise, let *tbl* be the table contained there. There are two branches from the root, labeled by constraints $\{\varnothing\}$ and $\mathcal{TABLE}$, respectively: execution goes on at the leftmost node such that the constraint labeling the corresponding branch contains *tbl*, if any. Thus, if $tbl = \varnothing$ execution goes on at the node labeled by Res_1 and calculation *constF*: Res_1 means that the first result must be *written* into the first R-mem, and *constF* means that such result is $constF(tbl)$ (i.e. F); since there are no branches from this node, execution terminates. If otherwise $tbl \neq \varnothing$, execution goes on at the node labeled by Arg_2, which means that the second argument (i.e. the object contained in the second A-mem) must be read. If it is $\perp$ execution stops and can then be resumed; otherwise, let *key* be the key contained there. There is one branch from this node, labeled by constraint $\mathcal{TABLE} \times \mathcal{KEY}$: the leftmost node such that the constraint labeling the corresponding branch contains $\langle tbl,key\rangle$ is thus the node labeled by Res_1 and calculation *is-present?*, so execution goes on there. Res_1 and *is-present?* mean that $is\text{-}present?(tbl,key)$ must be written into the first R-mem. There is one branch from this node, labeled by constraint $\{\langle tbl,key\rangle \in \mathcal{TABLE} \times \mathcal{KEY} \mid \exists\langle key,val\rangle \in tbl\}$: if $\forall\langle key,val\rangle \notin tbl$ then execution terminates because there is no node such that the constraint labeling the corresponding branch contains $\langle tbl,key\rangle$. Otherwise execution goes on at the

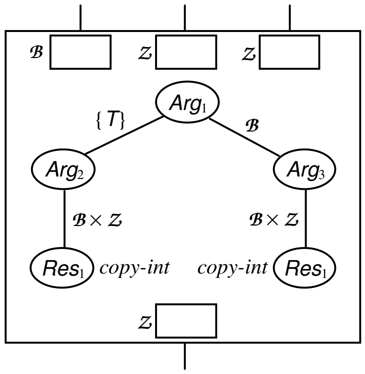

Fig.2 C^0-box *if*.

node labeled by Res_2 and calculation *assoc-val*, which mean that *assoc-val*(*tbl*,*key*) (which is not $\perp$) must be written into the second R-mem. Since there are no branches from this node, execution terminates.

Analogously, the execution of *if* (see Fig. 2) consists in reading the first argument and writing the second (third) argument as result if the first argument is T (F), and the execution of *plus* (see Fig. 3) consists in reading the first and second arguments and writing their sum as result; in both cases, if a $\perp$ argument is read execution stops and can then be resumed.

So, execution of a C^0-box moves down the tree along the leftmost path allowed by constraints: arguments are read from memory, and results are written into memory as specified by calculations; when a $\perp$ argument is read execution stops and can then be resumed; furthermore, when a calculation yields a $\perp$ result, execution stops with an error (anyway *search* is such that no error ever occurs). Calculations and tests for membership to constraints can be implemented in any language; the fact that a calculation gives $\perp$ for some arguments represents the fact that its implementation does not expect such arguments (e.g. for efficiency reasons), and this is why in our formalism execution stops with an error when a calculation yields a $\perp$ result.

Basically, two operations can be performed over C^0-boxes. One consists in *putting* an argument into memory: a $j \in \{1,\ldots,n\}$ and a $v \in \tau_j$ are specified, and v is written into A-mem j, at the same time writing $\perp$ into all R-mems (because previous results may change now). The other consists in *getting* a result from memory: an $i \in \{1,\ldots,m\}$ is specified, and if a $v \in \tau^i$ is in R-mem i such v is returned, otherwise the C^0-box is executed; execution can stop with an error, or stop because a $\perp$ argument has been read, or terminate; in the last case, if after termination R-mem i still contains $\perp$ an error occurs, otherwise the $v \in \tau^i$ contained there is returned.

We can thus put arguments into a C^0-box and get the corresponding results, and it is important that during execution the C^0-box sometimes avoids reading arguments which are irrelevant for determining results (e.g. *search* avoids reading the key if the table is empty) and avoids performing unnecessary computations (e.g. *search* avoids the computations of *is-present?* and *assoc-val* if the table is empty).

B. Higher-Level Computation Boxes

Fig. 4 depicts C^1-box *addkeys* from $\langle \mathcal{TABLE}, \mathcal{KEY}, \mathcal{KEY} \rangle$ to $\mathcal{Z}$. A C^u-box from $\langle \tau_1,\ldots,\tau_n \rangle$ to $\langle \tau^1,\ldots,\tau^m \rangle$ with $u \geq 1$ basically consists in a directed acyclic graph with *memory nodes* (represented as circles), *computation nodes* (represented as rectangles), and arcs connecting them; each arc connects nodes of different kinds; each memory node has at most one incoming arc. Each memory node is associated a type and contains an object of that type or the special value $\perp$; n distinguished memory nodes with no incoming arcs have types $\tau_1,\ldots,\tau_n$ and are called *argument memories* (*A-mems* for short); m distinguished memory nodes with no outgoing arcs have types $\tau^1,\ldots,\tau^m$ and are called *result memories* (*R-mems* for short); zero or more distinguished memory nodes (represented as thick circles) always contain the same non-$\perp$ values (indicated inside thick circles) of the associated types and are called *constant nodes*. Each computation node contains a C-box of level less than u (indicated inside rectangles) whose argument and result types match with the types associated to connected memory nodes in the obvious way. The numbers by non-constant memory nodes are for reference purposes only.

C^u-boxes with $u \geq 1$ can be *executed*, as we now explain by means of *addkeys* as an example (see Fig. 4), which "maps" a table and two keys to the sum of the integers identified by the keys in the table (if any of the keys is not present in the table, we consider 0 the value identified by that key). Suppose that nodes 1, 2 and 3 respectively contain a table *tbl* and two keys key_1 and key_2, and that all other nodes contain $\perp$; the contents of A-mems and R-mems of the five nested C^0-boxes are the same of the corresponding memory nodes, as obvious (e.g. the second A-mem of the rightmost *search* contains key_2). Since node 10 has an incoming arc from the computation node containing *plus*, the execution of *addkeys* starts by activating the execution of *plus*, which stops when

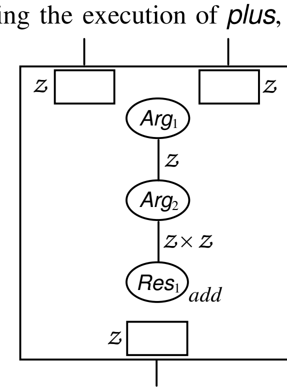

Fig.3 C^0-box *plus*.

$\perp$ is read as first argument. Since node 8 has an incoming arc from the leftmost computation node containing *if*, the execution of such *if* is activated, which stops when $\perp$ is read as first argument. Since node 4 has an incoming arc from the leftmost computation node containing *search*, the execution of such *search* is activated, which writes a boolean b_1 into node 4 (and also into the first A-mem of the leftmost *if* and into the first R-mem of *search* itself) and possibly an integer val_1 into node 5, then terminates. At this point, the execution of *if* is resumed: the first argument is read again, but now it is $b_1 \neq \perp$: if $b_1 = T$ the second argument val_1 is read and written into node 8; if $b_1 = F$ the third argument is read (which is always 0) and written into node 8. In both cases, as the execution of *if* terminates the execution of *plus* is resumed, which now reads $\perp$ as second argument; so, similarly to before the execution of the rightmost *if* and then of the rightmost *search* are activated, and at the end the execution of *plus* is resumed. The second argument is read again, and now it is an integer; the sum of the two integers in nodes 8 and 9 is written into node 10, and the execution of *addkeys* terminates.

So, execution of a C^u-box with $u \geq 1$ consists in activating executions of nested C-boxes: when one of them stops because a $\perp$ argument has been read, another one is activated as determined by node connections, in a sort of backward chaining; if one of them stops with an error, the whole C^u-box stops with an error; furthermore, if a $\perp$ argument is read from an A-mem of the C^u-box, the whole C^u-box stops its execution (e.g. if the first A-mem of *addkeys* contains $\perp$ the execution of *addkeys* stops), which can then be resumed. Thus, execution of C^0-boxes and execution of C^u-boxes with $u \geq 1$ are "externally" the same.

Basically, two operations can be performed over C^u-boxes with $u \geq 1$. One consists in *putting* an argument into memory: a $j \in \{1,\ldots,n\}$ and a $v \in \tau_j$ are specified, and v is written into argument memory j, at the same time writing $\perp$ into all those memory nodes whose content may depend on v (e.g. if we put a key $key_2' \neq key_2$ as third argument of *addkeys*, a $\perp$ is written into nodes 6, 7, 9 and 10, while the other nodes retain their contents). The other consists in *getting* a result from memory: an $i \in \{1,\ldots,m\}$ is specified, and if a $v \in \tau^i$ is in R-mem i such v is returned, otherwise the C^u-box is executed; unlike *addkeys*, in general a C^u-box may have more than one R-mem, whose incoming arcs are from distinct computation nodes, so i determines which nested C-box must be executed first. Execution of the C^u-box can stop with an error, or stop because a $\perp$ argument has been read, or terminate; in the last case, if after termination R-mem i still contains $\perp$ an error occurs, otherwise the $v \in \tau^i$ contained there is returned.

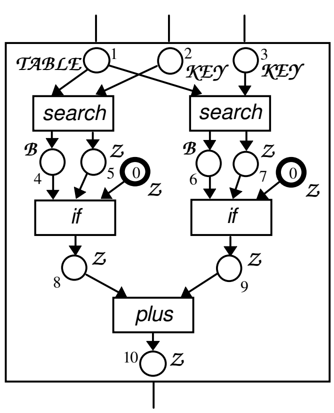

Fig.4 C^1-box *addkeys*.

We can thus put arguments into a C^u-box with $u \geq 1$ and get the corresponding results, but it is important that during execution the C^u-box sometimes avoids reading arguments which are irrelevant for determining results (e.g. *addkeys* avoids reading the two keys if the table is empty) and avoids performing unnecessary computations (e.g. *addkeys* avoids performing the computations of *is-present?* and *assoc-val* in the two *search*'s if the table is empty). Furthermore, suppose that, after the execution of *addkeys* explained above, a key $key_2' \neq key_2$ is put as third argument, and that we get the result: since an integer is still in node 8, the leftmost *if* and leftmost *search* are not executed; in general, during the execution of a C^u-box with $u \geq 1$ some computations may be avoided by using previously calculated intermediate results which are still valid.

C. Extended Functions

Given types $\tau_1,\ldots,\tau_n$ $(n \geq 1)$ and $\tau^1,\ldots,\tau^m$ $(m \geq 1)$, an *extended function* (*E-function* for short) from $\langle\tau_1,\ldots,\tau_n\rangle$ to $\langle\tau^1,\ldots,\tau^m\rangle$ is basically a function from $\tau_1\cup\{\perp\} \times \ldots \times \tau_n\cup\{\perp\}$ to $\tau^1\cup\{\perp\} \times \ldots \times \tau^m\cup\{\perp\}$. Each C-box from $\langle\tau_1,\ldots,\tau_n\rangle$ to $\langle\tau^1,\ldots,\tau^m\rangle$ is *associated* an E-function from $\langle\tau_1,\ldots,\tau_n\rangle$ to $\langle\tau^1,\ldots,\tau^m\rangle$ which constitutes the specification of the C-box, because it completely "characterizes" the C-box, as formalized by some theorems and as we now explain by means of examples.

C^0-box *search* is associated E-function *Search*, which is such that: *Search*$(tbl,key) = \langle F,\perp\rangle$ if key is not in tbl, which means that after we put arguments tbl and key, with key not in tbl, into *search*, we can then get F as first result without errors, but we cannot get the second result (because an error would occur); *Search*$(tbl,key) = \langle T,val\rangle$ if $\langle key,val\rangle \in tbl$, which means that after we put arguments tbl and key, with $\langle key,val\rangle \in tbl$, into *search*, we can then get both T and val as first and second result without errors; *Search*$(\varnothing,\perp) = \langle F,\perp\rangle$, which means that after we put $\varnothing$ as first argument into *search* we can then get F as first result without errors and without the second argument being read by *search* (i.e. the second argument may also be $\perp$, but the execution does not stop because *search* does not even attempt to read it); *Search*$(\perp,key) = \langle\perp,\perp\rangle$, which means that it is not possible to get any result without the first argument being read. The fact that the first result of *Search* is *T* iff the second one is non-$\perp$ means that, after we get the first result from *search*, we can then get the second one without errors iff the first one was *T* (and *search* does not execute again, because it has already written the second result into memory when the first one was gotten).

C^0-box *if* is associated E-function *If*, which is such that: $If(T,val_1,\perp) = val_1$ $(If(F,\perp,val_2) = val_2)$, which means that after we put *T* (*F*) and val_1 (val_2) into the first and second (third) A-mems of *if* we can then get val_1 (val_2) as result without errors and without the third (second) argument being read; $If(\perp,val_1,val_2) = \perp$, which means that the first argument is always read by *if*, and it is not possible to get the result without a boolean being present there.

C^0-box *plus* is associated E-function *Plus*, which is such that: $Plus(val_1,val_2) = val_1 + val_2$, which means that after we put two integers as arguments into *plus* we can then get their sum as result without errors; $Plus(\perp,val_2) = Plus(val_1,\perp) = \perp$, which means that *plus* always reads both its argument.

C^1-box *addkeys* is associated E-function *Addkeys*, which is such that: $Addkeys(tbl,key_1,key_2) = val_1 + val_2$, where for $i = 1,2$ val_i is the integer identified by key_i in *tbl* or 0 if key_i is not in *tbl*, which means that after we put arguments *tbl*, key_1 and key_2 into *addkeys* we can then get the sum of the integers identified by key_1 and key_2 in *tbl* (0 if not present) as result without errors; $Addkeys(\varnothing,\perp,\perp) = 0$, which means that after we put $\varnothing$ into the first argument memory of *addkeys* we can then get 0 as result without errors and without the other two arguments being read; $Addkeys(\perp,key_1,key_2) = \perp$, which means that it is not possible to get the result without the first argument being read.

So, in order to use a C-box, putting arguments and getting results, or nesting it into a higher-level one, we only need to know its associated E-function (which constitutes the specification), without bothering about its internal structure (which constitutes the implementation). For example, in order to use *search* properly we only need to know *Search*, without bothering about the tree in Fig. 1, its constraints and calculations. We can thus view a C-box as shown in Fig. 5. A very important point is that the E-function associated to a C^u-box with $u \geq 1$ is obtained by functionally composing the E-functions associated to nested C-boxes according to the graph structure of the C^u-box in the obvious way, thus allowing transparent local implementation changes.

III. AN EXAMPLE IN AUTOMATED MANUFACTURING

We now present a simple but representative example of use of C-boxes in automated manufacturing.

Consider the plant sketched in Fig. 6. Raw products (all equal to each other) from storage RP (which is never empty) are operated by machines M_1, M_2 and M_3 (each machine contains at most one product at once, and operates it in a null time) according to productive processes PP_1, PP_2 and PP_3 as indicated in Fig. 6, and the corresponding three kinds of finished products go into storage FP (which is never full); products are moved from place to place by robot R in a null time.

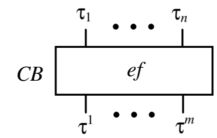

Fig.5 C-box *CB* whose associated E-function is *ef*.

The discrete event control of this plant may be realized by means of DES $S = \langle X,E,T \rangle$ defined as follows. We have

$$X = (\mathcal{MS} \times \mathcal{PP}) \times (\mathcal{MS} \times \mathcal{PP}) \times (\mathcal{MS} \times \mathcal{PP}),$$

where $\mathcal{MS} = \{\text{empty},\text{ready},\text{finished}\}$ (*machine states*) and $\mathcal{PP} = \{PP_1,PP_2,PP_3,\text{none}\}$, i.e. states are triples of pairs, each pair referring to a machine M_i: if no product is present in M_i the pair is $\langle \text{empty},\text{none} \rangle$; otherwise the pair is $\langle \text{ready},PP_j \rangle$ or $\langle \text{finished},PP_j \rangle$ if (respectively) the product has still to be operated or has already been operated by M_i, and PP_j is the productive process of the product. We have

$$E = \{O_1,O_2,O_3,PP_1,PP_2,PP_3,M_1,M_2,M_3\},$$

each event corresponding to a command to the physical controllers of M_1, M_2, M_3 or R: event O_i causes M_i to operate the product present in it, event PP_i causes R to move a raw product from RP to the first machine of process PP_i, and event M_i causes R to move the product in M_i to its next destination (another machine or FP). For each state $x = \langle\langle ms_1,pp_1 \rangle,\langle ms_2,pp_2 \rangle,\langle ms_3,pp_3 \rangle\rangle$ we have:

- $T(x,O_i) = x'$ is defined iff $ms_i = \text{ready}$, and in that case x' is x with ms_i changed to finished;
- $T(x,PP_i) = x'$ is defined iff $\langle ms_j,pp_j \rangle = \langle \text{empty},\text{none} \rangle$ (M_j being the first machine of PP_i), and in that case x' is x with $\langle ms_j,pp_j \rangle$ changed to $\langle \text{ready},PP_i \rangle$;
- $T(x,M_i) = x'$ is defined iff one of two cases holds: first case, M_i is the last machine of pp_i, and in that case x' is x with $\langle ms_i,pp_i \rangle$ changed to $\langle \text{empty},\text{none} \rangle$; second case, $\langle ms_j,pp_j \rangle = \langle \text{empty},\text{none} \rangle$ (M_j being the machine immediately following M_i in process pp_i), and in that case x' is x with $\langle ms_i,pp_i \rangle$ changed to $\langle \text{empty},\text{none} \rangle$ and $\langle ms_j,pp_j \rangle$ changed to $\langle \text{ready},pp_i \rangle$.

Such DES *S* may lead to deadlocks (e.g. no event can take place in state $\langle\langle \text{finished},PP_1 \rangle,\langle \text{finished},PP_3 \rangle,\langle \text{finished},PP_2 \rangle\rangle$). We may avoid deadlocks by using a controller *C* for *S* which maps each state *x* into the set $C(x)$ of all events *e* such that state $T(x,e)$ is safe, where a state is *safe* iff state $\langle\langle \text{empty},\text{none} \rangle,\langle \text{empty},\text{none} \rangle,\langle \text{empty},\text{none} \rangle\rangle$ is reachable from it.

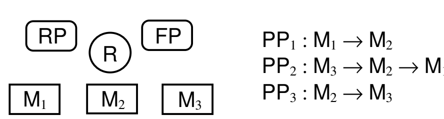

Fig.6 Sketch of the plant.

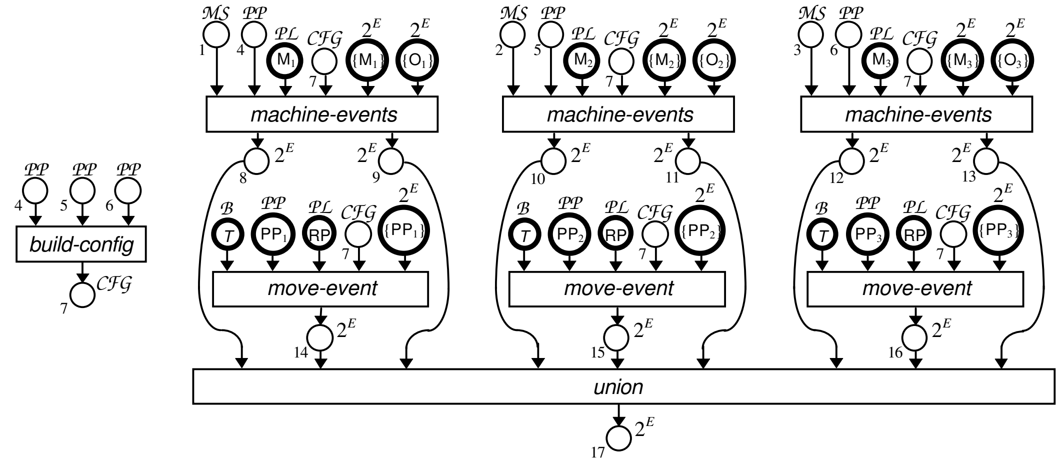

Fig.7a C-box *controller*.

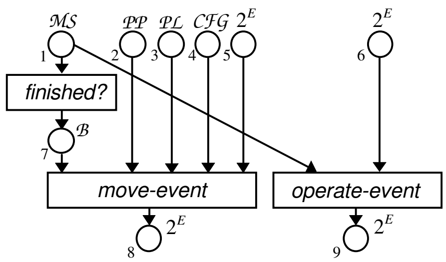

Fig.7b C-box *machine-events*.

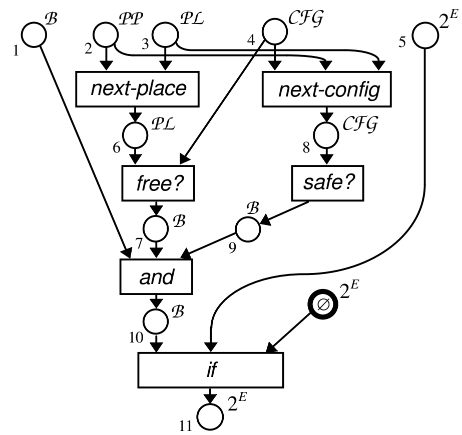

Fig.7c C-box *move-event*.

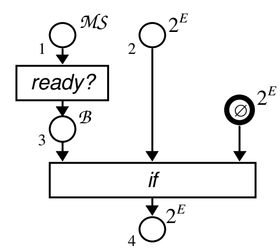

Fig.7d C-box *operate-event*.

Controller C is realized as C-box *controller* from $\langle \mathcal{MS},\mathcal{MS},\mathcal{MS},\mathcal{PP},\mathcal{PP},\mathcal{PP}\rangle$ (i.e. the types of the objects which compose states) to 2^E, depicted in Fig. 7a (the A-mems are nodes 1, 2, 3, 4, 5 and 6; to improve clarity some memory nodes appear more than once, like node 7); Fig. 7b, 7c and 7d respectively depict C-boxes *machine-events* from $\langle \mathcal{MS},\mathcal{PP},\mathcal{PL},\mathcal{CFG},2^E,2^E\rangle$ to $\langle 2^E,2^E\rangle$, *move-event* from $\langle \mathcal{B},\mathcal{PP},\mathcal{PL},\mathcal{CFG},2^E\rangle$ to 2^E and *operate-event* from $\langle \mathcal{MS},2^E\rangle$ to 2^E, where $\mathcal{PL} = \{\mathsf{RP},\mathsf{M}_1,\mathsf{M}_2,\mathsf{M}_3,\mathsf{FP}\}$ (*places*) and $\mathcal{CFG} = (\{\mathsf{M}_1,\mathsf{M}_2,\mathsf{M}_3\}^* \cup \{\mathsf{free}\})^3$ (*configurations*, see below; $\{\mathsf{M}_1,\mathsf{M}_2,\mathsf{M}_3\}^*$ is the set of all finite sequences of elements of $\{\mathsf{M}_1,\mathsf{M}_2,\mathsf{M}_3\}$).

Memory nodes 9, 11, 13, 14, 15, 16, 8, 10, 12 respectively "correspond" to events O_1, O_2, O_3, PP_1, PP_2, PP_3, M_1, M_2, M_3 in the following sense: just after getting the result from *controller*, each of these nodes contains the singleton set of the "corresponding" event (e.g. node 13 contains $\{\mathsf{O}_3\}$) if such event is enabled by C, otherwise it contains $\varnothing$.

C^0-box *union* reads the arguments and writes their union as result, which is also the result of *controller*.

C^0-box *build-config* reads the arguments pp_1, pp_2 and pp_3, and writes result $\langle cfg_1,cfg_2,cfg_3\rangle$ defined as follows: if $pp_i = \mathsf{none}$ then $cfg_i = \mathsf{free}$, otherwise cfg_i is the sequence of all machines which follow M_i in process pp_i (e.g. $cfg_3 = [\mathsf{M}_2,\mathsf{M}_1]$ if $pp_3 = \mathsf{PP}_2$).

C^0-boxes *finished?* and *ready?* read the argument and write result T if that argument is finished or ready (respectively), F otherwise.

C^0-box *if* is exactly like the one of the previous section, except that it deals with sets of events instead of integers.

C^0-box *next-place* reads the two arguments pp and pl, and writes result pl' which is the next place after pl where the product of process pp must go (e.g. $pl' = \mathsf{FP}$ if $pp = \mathsf{PP}_1$ and $pl = \mathsf{M}_2$); if $pp = \mathsf{none}$ an error occurs (i.e. the calculation specifies a $\perp$ result).

C^0-box *free?* reads the first argument pl, and if $pl = \mathsf{FP}$ then result T is written; otherwise, if $pl = \mathsf{M}_i$ it reads the second argument $\langle cfg_1,cfg_2,cfg_3\rangle$, and if $cfg_i = \mathsf{free}$ then result *T* is written, otherwise result *F* is written.

C^0-box *next-config* reads its arguments *cfg*, *pp* and *pl* and writes result *cfg′* which is the configuration obtained by moving the product of process *pp* from place *pl* to the next place where it must go; if $pp = \mathsf{none}$ or the product cannot be in *pl* or the next place is a non-empty machine then an error occurs.

C^0-box *and* reads the first argument, and if it is *F* then it writes result *F*; otherwise it reads the second argument, and if it is *F* then it writes result *F*; otherwise it reads the third argument and writes it as result.

C^0-box *safe?* reads the argument $cfg = \langle cfg_1,cfg_2,cfg_3\rangle$ and writes result *T* if *cfg* is safe (i.e. it corresponds to a safe state), result *F* otherwise. The calculation might be implemented by means of the following self-explaining Prolog program:

```
readcfgcomp(cfg(X,Y,Z),m1,X).
readcfgcomp(cfg(X,Y,Z),m2,Y).
readcfgcomp(cfg(X,Y,Z),m3,Z).
writecfgcomp(cfg(X,Y,Z),m1,W,cfg(W,Y,Z)).
writecfgcomp(cfg(X,Y,Z),m2,W,cfg(X,W,Z)).
writecfgcomp(cfg(X,Y,Z),m3,W,cfg(X,Y,W)).
directlyreachable(C1,C2):-
        readcfgcomp(C1,M,[]),
        writecfgcomp(C1,M,free,C2).
directlyreachable(C1,C2):-
        readcfgcomp(C1,M1,[M2|ML]),
        readcfgcomp(C1,M2,free),
        writecfgcomp(C1,M1,free,C),
        writecfgcomp(C,M2,ML,C2).
reachable(C1,C2):-directlyreachable(C1,C2).
reachable(C1,C2):-directlyreachable(C1,C),
                  reachable(C,C2).
```

The above program is executed upon

```
?-reachable(cfg(x1,x2,x3),cfg(free,free,free))
```

where x_i is `free` if $cfg_i = \mathsf{free}$, otherwise it is the Prolog list of `m1`, `m2`, `m3` corresponding to sequence cfg_i of M_1, M_2, M_3; result *T* or *F* is determined by answer `Yes` or `Failure`, respectively.

E-function *Controller* associated to *controller* is such that for each state $\langle\langle ms_1,pp_1\rangle,\langle ms_2,pp_2\rangle,\langle ms_3,pp_3\rangle\rangle$ with $pp_i = \mathsf{none}$ if $ms_i = \mathsf{empty}$, we have that *Controller*$(ms_1,ms_2,ms_3,pp_1,pp_2,pp_3)$ is non-$\perp$ and it is the set of all events enabled by *C*. Thus, after putting arguments constituting the initial state (i.e. $\langle\langle\mathsf{empty},\mathsf{none}\rangle,\langle\mathsf{empty},\mathsf{none}\rangle,\langle\mathsf{empty},\mathsf{none}\rangle\rangle$) we can repeatedly get the result and change some arguments according to the state transition determined by any event of that result: computations will always terminate without errors, and no deadlock in the plant will ever occur. Note that when for example event O_3 takes place, we only change the third argument (in node 3), thus when we get the result only the following C^0-boxes are executed (in order): *union*; *if* and *and* inside *move-event* inside the rightmost *machine-events*; *finished?* inside the rightmost *machine-events*; *if* and *ready?* inside *operate-event* inside the rightmost *machine-events*.

IV. FUTURE WORK

As the example in the previous section has shown, the characteristics of C-boxes make them particularly well-suited to be used as efficient controllers for DESs. Anyway, we think that C-boxes constitute a general software engineering formalism for the modular synthesis of efficient procedures, which allows the use of different languages in a clear way: thus, it is worth investigating other fields in which C-boxes could be applied.

Another interesting direction for future work is the development of a software system for the synthesis of C-boxes, in which different languages may be used to implement calculations. In principle, the user should define a procedure for each calculation and a procedure for test of membership to each constraint; in practice, it would be better to allow more flexibility (e.g. in C-box *search* the two results should be produced by one procedure, even if we have two distinct calculations). Efficiency of synthesized C-boxes could be greatly increased by automatic and user-transparent flattening of C-boxes to C^1-boxes, and also of parts of them to C^0-boxes (in fact, if we have computation nodes containing very simple and fast C-boxes, it is too costly to read from and write into memory nodes). It would also be interesting to integrate such software system into existing packages for the synthesis of software systems, such as PETREX [6].

V. REFERENCES

[1] J. L. Peterson, *Petri Net Theory and the Modeling of Systems*, Englewood Cliffs, NJ, Prentice Hall; 1981.

[2] K. Jensen, “Coloured Petri Nets: A High Level Language for System Design and Analysis”, in G. Rozenberg (ed.), *Advances in Petri nets 1990*, Lecture Notes in Computer Science, vol.483, Springer-Verlag, 1990, pp.342–416.

[3] M. C. Zhou and F. DiCesare, *Petri Net Synthesis for Discrete Event Control of Manufacturing Systems*, Kluwer Academic Publishers; 1993.

[4] P. J. Ramadge and W. M. Wonham, “The Control of Discrete Event Systems”, *Proceedings of the IEEE*, vol.77, no.1, Jan. 1989, pp.81–98.

[5] A. Camurri and A. Coglio, *Computation Boxes and Extended Functions*, DIST Technical Report, in preparation.

[6] A. Camurri, P. Franchi, M. Vitale, “Future Directions: Object Oriented High Level Petri Nets for Real-Time Scheduling of FMS”, in L.F. Pau and J.O. Willums (eds.), *Manufacturing Automation at the Crossroads – Standardization in CIM Software*, IOS Press, 1993, Chapter 6, pp.57-83.