STCT: An Efficient Algorithm for Supervisory Control Design

Zhonghua Zhang and W.M.Wonham
Systems Control Group
ECE Department, University of Toronto
zhangzh, wonham@control.utoronto.ca

June 26, 2001

1 Abstract

This paper introduces a new synthesis approach for the supervisory control of discrete-event systems (DES). Our algorithm, named S(mart)TCT after our software package CTCT hitherto in use, is much more efficient than CTCT. Efficiency is achieved by exploiting the modular composition of the plant and specification in DES, and its embodiment in integer decision diagrams (IDDs) as the basic data structure.

2 Introduction

There are (at least) two widely recognized approaches to the supervisory control of discrete-event systems.

The (so-called) RW theory (often now designated SCT), introduced in [RW87], is based on automata and formal languages (in practice, finite automata and regular languages). Automata are used to model both the plant to be controlled and the specification. An optimal (i.e. maximally permissive) supervisor, constraining the plant to generate only specified behavior, always exists (and is nonempty if the control problem is solvable at all). It is effectively computable, as a finite supervisory automaton representing optimal controlled behavior, and is implementable by synchronizing this supervisor with the plant via shared events. SCT treats both standard 'safety' specifications, and a weak form of 'liveness' specifications (nonblocking, or coreachability of reachable states with respect to marker states - in practice often quite adequate). Recent expositions can be found in [CL99],[Won99]. Several software implementations of SCT are publicly available; we use CTCT [Won99] for comparison purposes in this paper. Its main limitation is that it employs 'extensional' listing of the relevant state transition structures, and thus processing time and memory that typically grow exponentially with the number of modules making up the total system. It will turn out that this limitation is not inherent in SCT, except in 'worst' cases, but is mainly, in practice, a consequence of naivete of representation.

The main alternative approach to SCT is through modeling the plant and specification by vector addition systems, represented as Petri nets [Kro87] or (more-or-less equivalently) by VDES [LW93][LW94][Won99]. Theory and examples in the idiom of Petri nets can also be found in [UM88] [Den89] [GD92] [ZD93] [BA95] [MA98] [ZV99]. The VDES/PN formulation is most natural when the system to be controlled is composed from groups of similar entities (e.g. machines) of which the relevant states are the numbers within each group with a given attribute (e.g. working, down). The control-theoretic framework in most of these instances is adapted from SCT; because of the vector-additive setting, specifications are usually in the form of systems of linear (integer) inequalities; and linear integer programming, for instance, is an effective tool of synthesis. Unfortunately, VDES/PN does not lend itself conveniently to the incorporation of nonblocking as a synthesis objective (whereas in SCT via automata, this objective is achieved 'automatically'). For this an analysis (via PN concepts such as siphons) specific to some special class of systems of interest is usually needed in advance [BA95] [DAJ95] [MA98] [SS00] : an SCT-style search over the entire state space is intentionally avoided, but efficiency is only achieved at the price of nontrivial prior analysis and consequent loss of generality.

In this paper we report recent success with an approach (STCT) which combines the exploitation of structural properties of the target problem along with universality in the treatment of nonblocking. To us, much of the usefulness of VDES/PN lies in the retention in the state modeling of cartesian product structure, and not so much in vector additivity *per se*. Our product factors are the functional modules of the system, each represented by a (typically small) automaton as called-for. These factors are ordered, quasi-optimally and 'automatically' in accordance with built-in heuristics. The relevant properties of reachability,

Approach	Algorithm	Generality	Efficiency
CTCT	Exhaustive search	Good	Poor
VDES /PN	Structural Analysis	Poor	Good
STCT	Structural Search	Good	Good

Table 2.1: Comparison of three approaches to the supervisory control problem with nonblocking

coreachability (thus, nonblocking) and controllability, as well as the enablement/disablement rules for the optimal controls, are encoded as predicates on the product space. This encoding makes crucial use of integer decision diagrams (IDDs) for modeling compactness. Related previous work includes [Ger95] [Gun97] (see below), [Tro97] (which exploits Boolean decision diagrams) and [MP00] (symbolic bisimulation).

A (perhaps sweeping) overall comparison of CTCT, VDES/PN, and STCT, as we see them, is displayed in Table 2.1.

For numerous details, omitted from this paper, see [Zha01].

3 Structural predicates and control synthesis

3.1 Synthesis algorithm for simple DES

Suppose we have a simple DES, whose plant and specification are each given in the form of a single automaton [1]

- Plant: $\mathbf{G}_p = (X, \Sigma, x_0, \xi, X_m)$.
- Specification: $\mathbf{G}_s = (Y, \Sigma, y_0, \eta, Y_m)$.

Note that we *require* that the event sets of the plant and specification be the same.

The synchronized product of $\mathbf{G}_p$ and $\mathbf{G}_s$ can be obtained by CTCT procedure **meet** ([Won99]) as

$$\mathbf{G}_{meet} = \mathbf{meet}(\mathbf{G}_p, \mathbf{G}_s) = (X \times Y, \Sigma, (x_0, y_0), \xi \times \eta, X_m \times Y_m).$$

[1] Notes on the terminology used in this paper. Different symbols are used to represent the elements of $\mathbf{G}_p$ and $\mathbf{G}_s$ where it is desired to distinguish the automaton model of plant and specification. When this distinction is not necessary, we write $\mathbf{G} = (Z, \Sigma, z_0, \zeta, Z_m)$ where $Z := X \times Y$, $z_0 = (x_0, y_0)$, $\zeta := \xi \times \eta$, $Z_m := X_m \times Y_m$. Also, vector form of all those notations will be used later, i.e. we have $\mathbf{G}_p = \mathbf{sync}(\mathbf{G}_{pi})$, $i = 1, 2, \ldots N_p$; $\mathbf{G}_s = \mathbf{sync}(\mathbf{G}_{sj})$, $j = 1, 2, \ldots N_s$; and $\mathbf{G} = \mathbf{sync}(\mathbf{G}_i)$, $i = 1, 2, \ldots N = N_p + N_s$. For **sync** see [Won99].

In order to place conditions on the state (x, y) of $\mathbf{G}_{meet}$, it will be convenient to use a logic formalism. A *predicate* P defined on $X \times Y$ is a function $P : X \times Y \to \{0, 1\}$. P can be identified with the corresponding state subset

$$(x, y)_P := \{(x, y) \in X \times Y | P(x, y) = 1\}.$$

Remark: P is defined on a '2-dimensional' set and thus holds not only information about conditions we place on the states, but also structural information about the system. We refer to such P as a 'structural predicate'. The logic operators defined for predicates for simple sets apply immediately to this kind of predicate. For clarity, if $P(x, y) = 1$ we write $(x, y) \models P$, otherwise $(x, y) \not\models P$.

We define the *closed language* $L(\mathbf{G}_{meet}, P)$ induced by a predicate $P \in Pred(X \times Y)$ to be

$$L(\mathbf{G}_{meet}, P) := \{w \in L(\mathbf{G}_{meet}) | (\forall v \leq w)\big((\xi(x_0, v), \eta(y_0, v)) \models P\big)\}$$

and the *marked language* $L_m(\mathbf{G}_{meet}, P)$ induced by a predicate $P \in Pred(X \times Y)$ to be

$$L_m(\mathbf{G}_{meet}, P) := \{w \in L(\mathbf{G}_{meet}, P) | (\xi(x_0, w) \in X_m, \eta(y_0, w) \in Y_m)\}$$

The following definitions introduce the controllability and nonblocking properties of a structural predicate P.

Definition 3.1 Reachability of a given state.

For a given P, a state (x, y) is *reachable* iff

$$(\exists w \in L(\mathbf{G}_{meet}, P)) \big(\xi(x_0, w) = x \text{ and } \eta(y_0, w) = y\big)$$

Definition 3.2 Coreachability of a given state.

For a given P, a state (x, y) is *coreachable* iff

$$(\exists w \in \Sigma^*) \big[\xi(x, w) \in X_m \text{ and } \eta(y, w) \in Y_m \text{ and } (\forall v \leq w)(\xi(x, v), \eta(y, v)) \models P\big]$$

Definition 3.3 Controllability of structural predicate P.

For the given $\mathbf{G}_p$ and $\mathbf{G}_s$, P is *controllable* iff $L(\mathbf{G}_{meet}, P)$ is controllable w.r.t $L(\mathbf{G})$.

Definition 3.4 Nonblocking property of structural predicate P.

For the given $\mathbf{G}_p$ and $\mathbf{G}_s$, P is *nonblocking* iff $\overline{L_m(\mathbf{G}_{meet}, P)} = L(\mathbf{G}_{meet}, P)$.

We have two useful lemmas about structural predicates.

Lemma 3.1 For the given $\mathbf{G}_p$ and $\mathbf{G}_s$, P is controllable iff

$$(\forall (x, y) \models P, \forall \sigma \in \Sigma_u)\big[\xi(x, \sigma)! \Rightarrow \eta(y, \sigma)! \text{ and } (\xi(x, \sigma), \eta(y, \sigma)) \models P\big]$$

Lemma 3.2 For the given $\mathbf{G}_p$ and $\mathbf{G}_s$, P is nonblocking iff

$$(\forall (x,y) \models P)$$
$$((x,y) \text{ is reachable}) \Rightarrow ((x,y) \text{ is coreachable})$$

Now we can present the main result of this section:

Theorem 3.1 Given $\mathbf{G}_p$ and $\mathbf{G}_s$, there exists a unique structural predicate P_{sup}[1] such that

1. $L_m(\mathbf{G}_{meet}, P_{sup}) = \sup\mathcal{C}(L(\mathbf{G}_p), L_m(\mathbf{G}_s))$.
2. P_{sup} is reachable and nonblocking.

where $\sup\mathcal{C}(L(\mathbf{G}_p), L_m(\mathbf{G}_s))$ is the supremal controllable sublanguage of $L_m(\mathbf{G}_s)$ w.r.t $L(\mathbf{G}_p)$ as defined in [Won99].

Proof:
Given $\mathbf{G}_p$ and $\mathbf{G}_s$, we can construct P_{sup} as follows:

1. Initialize $P_{good} \equiv 1$.
2. $P_{bad1}: \ (x,y) \models P_{bad1}$ iff
   $((x,y) \not\models P_{good})$
   or $(\exists \sigma \in \Sigma_u)\big((\xi(x,\sigma)!) \text{ and } (\neg(\eta(y,\sigma)!))\big)$
   or $(\exists \sigma \in \Sigma_u)$
   $\big(\xi(x,\sigma)!, \eta(y,\sigma)!$
   and $(\xi(x,\sigma), \eta(y,\sigma)) \not\models P_{good}\big)$
3. $P_{bad2}: \ (x,y) \models P_{bad2}$ iff
   $(\exists w \in \Sigma_u^*)((\xi(x,w), \eta(y,w)) \models P_{bad1}$
4. $P_{re}: \ (x,y) \models P_{re}$ iff
   $(\exists w \in \Sigma^*)$
   $(\xi(x_0,w) = x, \eta(y_0,w) = y)$,
   $(\forall v < w)(\xi(x_0,v), \eta(y_0,v)) \not\models (P_{bad1} \cup P_{bad2})$
5. $P_{cr}: \ (x,y) \models P_{cr}$ iff
   $(\exists w \in \Sigma^*)$
   $(\xi(x,w) \in X_m, \eta(y,w) \in Y_m)$,
   $(\forall v < w)(\xi(x,v), \eta(y,v) \not\models (P_{bad1} \cup P_{bad2})$
6. $P_{newgood} = P_{re} \cap P_{cr}$.
7. If $P_{newgood} \subset P_{good}$ repeat steps 2-6 with $P_{good} = P_{newgood}$, otherwise let $P_{sup} = P_{newgood}$ and the algorithm terminates here.

It can be shown that P_{sup} is the desired result and we write $P = \mathbf{supcon}(\mathbf{G}_p, \mathbf{G}_s)$. The remaining details are omitted owing to space limitations (see [Zha01]).□

[1]It is possible that $P_{sup} \equiv 0$. This means $\sup\mathcal{C}(L(\mathbf{G}_p), L_m(\mathbf{G}_s)) = \emptyset$. If this happens, we know that a solution to the given control problem does not exist. In other words, the constraints are too tight for the system.

3.2 Synthesis algorithm for complex DES

The algorithm described in the previous section can be easily extended to complex DES systems.

Suppose we have a complex DES system, whose plant and specification are both given in the form of the synchronous product of more than one automaton module:

- Plant: $\mathbf{G}_p = \mathbf{sync}(\mathbf{G}_{pi})$.
  $\mathbf{G}_{pi} = (X_i, \Sigma, x_{0i}, \xi_i, X_{mi})$. $i \in 1, \ldots, N_p$.
- Specification: $\mathbf{G}_s = \mathbf{sync}(\mathbf{G}_{sj})$.
  $\mathbf{G}_{sj} = (Y_j, \Sigma, y_{0j}, \eta_j, Y_{mj})$. $j \in 1, \ldots, N_s$.

Then similarly as in the previous section, it can be shown that the synthesis and implementation of the controller can be done in the form of a $N_p + N_s$-variable structural predicate:

Theorem 3.2 Given $\mathbf{G}_{pi}$ and $\mathbf{G}_{sj}$ as shown above, there exists a structural predicate P_{sup} such that

1. $L_m(\mathbf{G}_{meet}, P_{sup}) =$
   $\sup\mathcal{C}(L(\mathbf{sync}(\mathbf{G}_{pi})), L_m(\mathbf{sync}(\mathbf{G}_{sj})))$.
2. P_{sup} is reachable and nonblocking.

Proof: The proof is identical to the proof in the previous section, except for a description of higher dimension. Also, the same constructive steps can be used to compute P_{sup}. □

3.3 Implementation of supervisory controller by structural predicate

As mentioned before, the *supervisory control* is a function

$$V : Q \to \Gamma$$

where $\Gamma = \{\gamma \in Pwr(\Sigma) | \gamma \supseteq \Sigma_u\}$. Thus it can also be expressed as

$$V' : Q \to (e_{\sigma 1}, e_{\sigma 2}, \ldots e_{\sigma n_e})$$

where $e_{\sigma i} \in \{0,1\}$, n_e is the number of events, and $e_{\sigma i} = 1$ if $\sigma_i \in \Sigma_u$. Then we can express V' in the form of n_e scalar functions:

$$V' \equiv \big\{V_i' : Q \to \{0,1\} \; i \in \{1, 2, \ldots, n_e\}\big\}$$

This is nothing but n_e structural predicates. Therefore *supervisory control* can be implemented by structural predicates in a straightforward way.

The V' defined above can be obtained as follows. Let $z = (x_1, \ldots, x_{N_p}, y_1, \ldots, y_{N_s})$, $\zeta : Z \times \Sigma \to Z$ be the synchronized (partial) transition function $\xi \times \eta$, and let

$$e_{\sigma i}(z) = \begin{cases} 1 & \text{if } z \models P_{sup}, \zeta(z,\sigma_i)!, \zeta(z,\sigma_i) \models P_{sup} \\ 0 & \text{otherwise} \end{cases}$$

Obviously, for any $\sigma \in \Sigma_u$, $e_\sigma \equiv 1$, because P_{sup} is controllable. Also, clearly $L(V'/\mathbf{G}) = L(\mathbf{G}, P_{sup})$ and $L_m(V'/\mathbf{G}) = L_m(\mathbf{G}, P_{sup})$, since exactly the same constraints are imposed upon $\mathbf{G}$ by V' and by P_{sup}.

4 Algorithm, IDDs, optimization and complexity

As shown in the previous section, all the computation needed for procedure **supcon** can be realized by using structural predicates. However, if we use the naive representation of structural predicate, the value table, we gain no advantage compared with the CTCT implementation. Fortunately, Integer Decision Diagrams (IDD) can be used here so that we can actually represent the structural predicate in *compressed* form.

IDDs are an extended form of Binary Decision Diagram (BDD) which was introduced by Akers [Ake78]. Then Bryant [Bry86] introduced operations and algorithms that utilize the ordering of the variables in BDDs. To extend the Boolean representation to general finite functions has been an important task. A direct IDD representation was studied in [Gun97], though with focus on the relational representation, not the computation[2].

A direct and efficient algorithm based on IDDs, namely STCT, will be presented here. It has several features worth mentioning:

- It is a native C++ implementation and thus is very fast and compact.
- It has the ability to find the (nearly) optimal order of the variables of IDDs.
- It uses an incremental method to improve the performance of the synthesis of a system with buffers.

[2]The algorithm in [Gun97] was implemented by first translating the control logic to logical expressions in Mathematica, and then completing the synthesis in Mathematica. Although Mathematica is considered an efficient tool for symbolic manipulations for mathematicians, its efficiency can hardly be compared with a program of native binary code. For example, the simple logical 'AND' can be done by one CPU instruction in a binary code, but in Mathematica, where all objects are stored in a symbolic way, this operation needs many more CPU cycles (transformation between symbolic value and machine data, and there are quite a few function calls that must be invoked). Besides, the translation adds considerable overhead to the synthesis procedure.

4.1 Operations necessary to complete the synthesis

The computational procedure was outlined in the proof of Theorem 3.1. In actual computation, all the manipulations of structural predicates are realized by IDDs; furthermore, no prior extensional representation of the relevant (conceptual) synchronous products is necessary. We can find the required procedures by analyzing the procedure listed in the previous section.

Steps 1,2,6,7 can be easily realized by logical operations, i.e. AND, OR, and NEGATION.

Steps 3,4,5 need more consideration. We define the operation $R : \mathit{Pred}(Z) \rightarrow \mathit{Pred}(Z)$ as follows. Let $P_1 = R < P_0, \Sigma', P' >$ be the reachable predicate starting from P_0, while enabling all $\sigma \in \Sigma'$ and satisfying predicate P', i.e.

$$z \models P_1 \Leftrightarrow (\exists z' \models P_0, s \in \Sigma'^*)\big((\forall v \leq s)(\zeta(z', v) \models P')\big)$$

Similarly we define operation CR to produce the coreachable predicate. Thus steps 3,4,5 can be realized by R and CR, i.e.

- Step 3. $P_{bad2} = CR < P_{bad1}, \Sigma_u, \mathit{true} >$.
- Step 4. $P_{re} = R < P_0, \Sigma, \neg(P_{bad1} \cup P_{bad2}) >$
- Step 5. $P_{cr} = CR < P_m, \Sigma, \neg(P_{bad1} \cup P_{bad2}) >$

Here P_0 is true for and only for the initial state, and P_m is true for and only for the marker state(s).

Define the operation $N_\sigma : \mathit{Pred}(Z) \rightarrow \mathit{Pred}(Z)$ to produce the *one-step reachable predicate*. That is to say, $P' = N_\sigma(P)$ is defined as

$$(z' \models P') \Leftrightarrow (\exists z \models P)(\zeta(z,\sigma)!, \zeta(z,\sigma) = z')$$

The operation $P_1 = R < P_0, \Sigma', P >$ can then be realized by using N_σ and logical operations as follows.

1. $P_1 = P_0$.
2. $P_1' = P_1$
3. For each $\sigma \in \Sigma'$ compute $P_1' = P_1 \cup N_a(P_1)$.
4. If $P_1 \not\equiv P_1'$ goto step 1. Otherwise the procedure terminates and P_1 is what we want.

Similarly, define M_σ to be the *one-step coreachable predicate* operation, i.e.

$$(z' \models M_\sigma(P)) \Leftrightarrow (\exists z \models P)(\zeta(z',\sigma)!, \zeta(z',\sigma) = z)$$

and then the operation CR can be done using logical operations and M_σ.

As we can see, there are two types of computation that need to be done on IDDs:

1. Logical operations, i.e., AND, OR, and NEG(ATION).
2. The transformations N_σ and M_σ.

The implementation of logical operations is not very different from BDDs. The reader can refer to [Gun97] for further information on the general manipulation of IDDs.

Procedure *PostPred* is used to implement the predicate transformer *one-step reachable predicate*, i.e. $PostPred(IDD, \mathrm{a}) = N_a(P)$.

Definition 4.1 Procedure PostPred is prototyped as void PostPred(IDDA, σ);
where IDDA is the IDD representing the predicate P and σ is the given event. After the execution of the procedure, IDDA will represent $N_\sigma(P)$. The procedure is:

```
void PostPred(IDDA, σ)
{
 IDDB=EnableCondition(σ);
 IDDA=AND(IDDA, IDDB)
 for each level in 0..(IDDA.nLevels-1) do {
     for each nd in IDDA.nodeIndex[level] do {
        newNode=copyNode(nd)
        for each j in 0..(nd.nChildren) do {
           newNode.listChild[j]=nd.listChild[j];
           nd.listChild[j]=IDDA.nodeFalse;
        }
        listTransition=getListTrans(σ, level);
        for each t in listTransition do{
           nd.listChild[t.to]=
                       newNode.listChild[t.from];
        }
     }
  }
}
```

where

- *EnableCondition(σ)* returns a predicate E_σ defined as

  $$(z_1, z_2, \dots, z_N) \models E_\sigma \Leftrightarrow (\forall i \in \{1 \dots N\})\zeta_i(z_i, \sigma)!$$

  i.e., a state satisfies E_σ if event σ is enabled at that state.

- *copyNode(aNode)* creates an identical new node as *aNode* . In procedure PostPred *newNode* is created to keep a backup copy of *nd* .

- *getListTrans(σ, level)* returns the list of transitions of event a in the automaton model of component *level* . If all transitions of σ in that component are selfloops and σ is enabled at all states, *getListTran(σ, level)* will return an empty list.

The transformation M_σ can implemented similarly.

4.2 Ordering of variables

Ordering of variables in an IDD (corresponding to the ordering of the components in the system) is the most important optimization other than reduction. In fact, an effective reduction relies on a good ordering of all the variables.

From the IDD point of view, the reason why a reduction is almost always possible is that the information given in level i, is often not correlated to the information given in another level j. If the information from two different levels is not correlated, we can expect a reduction to be possible, otherwise we normally can't.

For most systems in practice, it will be true that in P_r, the predicate representing its reachable state space, not all the components' information contents are correlated to each other. Intuitively, we can get a hint of whether two components' information contents will be correlated by looking at their events. If they have shared event(s), it will be likely that their information will be correlated, and vice versa. A similar idea is also applicable to the coreachable-state specification predicate.

Therefore, the basic idea behind the ordering of variables in the STCT algorithm is that

- the more closely coupled two components are, the 'closer' the two components should be placed in the IDD.

As we can see, in order to improve the efficiency of an IDD, we should reduce the separation of two coupled components as much as we can. From another point of view, this is equivalent to reducing the crossing over of components. 'Crossing over' of a given component i means that in the given ordering, there is a component in front of i that is coupled to a component behind i. For an 'ideal' ordering of an 'ideal' system, each component is only coupled with the two components in front of and behind it, thus the overall 'crossing' is 0.

The overall crossing can serve as a good measurement of the quality of a given ordering. Thus we can define a function to evaluate a given ordering as follows.

Definition 4.2 Connection Matrix.
Let $C_{i,j}$=the number of events shared by $Component_i$ and $Component_j$.

For a given ordering of N components, $O = (O_1, O_2, \dots, O_N), O_i \in 0, \dots, N-1$, the number of crossings over of $Component_i, i \in 1 \ldots N$ is defined as

$$Cross_i = \sum_{\substack{j,k \\ Rank(j) < Rank(i) \\ Rank(k) > Rank(i)}} C_{j,k}$$

where $Rank(i)$ is the rank of component i and always satisfies $Rank(O(i)) = i$.

Then the overall crossing, as a measure of the optimality of the given ordering $Eval(O)$, is defined as

$$Eval(O) = \sum_i 4^{Cross_i}.$$

This definition is empirical rather than theoretical [3]. When we try to reduce the overall crossing in the IDD, we effectively try to reduce those larger $Cross_i$.

Now the ordering is an optimization problem. Given the nonlinear nature of the measurement function, it is very unlikely that we can find a global optimal solution in reasonable time.Therefore a randomized simulated-annealing algorithm is used instead and yields quite satisfactory results.

4.3 Incremental computation on buffer sizes

For a system with many nontrivial components, a problem of "intermediate node number explosion" will become quite serious. That's to say, although in the final results the number of nodes is not large, it will be tens of times larger during the synthesis process, or more accurately, during the generation of reachable or coreachable structural predicates. This has an obvious impact on the performance, since the time complexity of the algorithm depends directly on $|nodes|$.

This is an intrinsic problem of IDDs. An IDD is actually a classification of states based on their structural information and is obtained by an iterative process. In other words, the IDD is formed 'little by little', therefore we just cannot expect it to 'stay' in the optimal form all the way.

Fortunately, for a system with a lot of buffer-like components, as many practical systems have, there is a simple way to deal with this problem.

Actually, we *can* get good results for the final IDDs; the problem is the IDDs become too large at the halfway point of the computation. Thus if we can always keep those IDDs near to the final form, we would probably get a good result. This can be achieved by enlarging the system step by step. For a system with a maximum buffer size of M, we can start with a system with the same structure but with all buffer (and 'buffer-like' components) sizes set to 1. After we complete the system with buffer size of 1, we enlarge all buffers to size 2, and generate the IDDs again. After that we enlarge the buffers to size 3 ... until every buffer reaches its final size. In this way, we always keep the system near the 'final' stage. The performance improvement gained by the incremental approach will be reported in the next section.

[3] The number '4' isn't crucial at all. The sole purpose is to make the evaluation function grow fast with $Cross_i$.

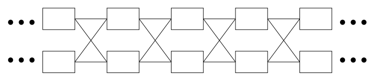

Figure 4.1: Topology of a production line

4.4 A conceptual complexity analysis

Unfortunately, it is difficult to provide a rigorous analysis of complexity of this algorithm, because the time and space complexity are not directly related to any given factors or their easily computable functions. In previous sections, it was implied that the complexity is actually directly related to $|nodes|$ in the IDD representation of the structural predicates. [4] However, $|nodes|$ cannot easily be estimated since it is related to both the parameters (such as number of components, numbers of states of each component, number of events) and the underlying structure of the system(especially the pattern of event sharing). We will provide a rough estimate of the relationship between $|nodes|$ and the parameters of the system.

4.4.1 Estimate of $|nodes|$

As a system becomes more complicated (e.g. a production line becomes longer), the number of modules will certainly become larger. However, we might expect that the degree of coupling remains constant. This is simply because of the nature of human intelligence – we are able and willing to build a 'production line' instead of a 'production manifold'(in which components are densely cross-linked). In other words, most man-made systems are probably more like Fig 4.1 than Fig 4.2. Thus we can expect the degree of coupling to be constant when N increases.

Suppose the following:

[4] It can be shown that computation between two IDDs has a quadratic complexity with respect to the number of nodes in the worst case.

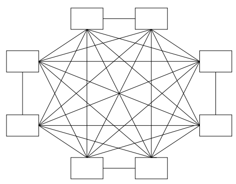

Figure 4.2: Topology of a production manifold

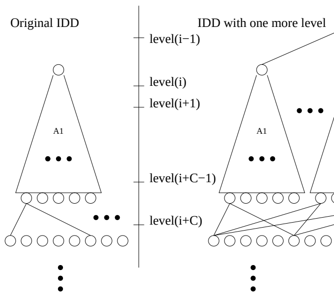

Figure 4.3: The growing of an IDD

1. Each component has exactly K states, i.e., there are exactly K branches at each non-terminal node of the IDD.

2. In the IDD, the component of $level_i$ is only coupled with components in the adjacent C levels.

Remark: The first assumption is made for simplicity only, and it places no actual constraint on the system. The second one is true for most practical systems, but not all.

With the above assumption, it can be estimated that

$$|nodes| = O(N \times K^C)$$

This is illustrated in Fig 4.3.

We can (theoretically) build the IDD by adding levels one by one. Assume we have an IDD with N levels and we want to add another level to the top of the IDD. With the above assumption, the added level can at most have a close relationship with the top C levels; therefore we can expect that levels far down will not be severely disturbed by adding a new top level. Adding a top level can have a big impact on the upper levels, but at most only K^C more nodes can be added to the IDD. (This is because each Ai shown in the graph can contain at most $(K^C - 1)/(K - 1)$ nodes and there are at most $K - 1$ such Ai that can be added to the IDD.)

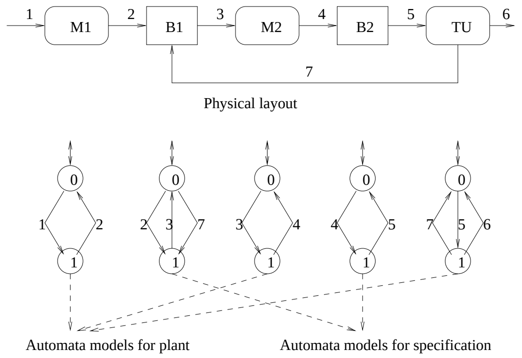

Figure 5.4: Transfer Line

Thus one more level means K^C more nodes, and we have the above formula.

Here we get a very interesting implication from this estimate:

- For a practical system, the complexity of solving the control problem may be polynomial (even linear) and not exponential w.r.t. N.

Of course this estimate is rather inaccurate and ideal. However it does give us some idea about the complexity of a system.

5 Examples

5.1 Transfer Line

A simple Transfer Line([DAJ95]) and its automaton model([Won99]) are shown in Fig 5.4. Transfer Line consists of two machines M1 and M2, followed by a test unit TU, linked by buffers B1 and B2, in the configuration shown. A workpiece tested by TU may be accepted or rejected; if accepted, it is released from the system; if rejected, it is returned to B1. Thus the structure incorporates 'material feedback'. The specification is simply that B1 and B2 must be protected against underflow and overflow. The events are defined as shown in Table 5.2:

The model depicted in Fig 5.4 can be easily synthesized using CTCT. Practically speaking, STCT doesn't enjoy any particular advantage in such a simple system.

However, real-life systems are not always so simple. A typical FMS comprises numerous resources - buffers, machines, robots, automatic guided vehicles. And normally, those resources have multiple instances. For example, we may have two identical machines to perform M1's task, and thus increase the efficiency of the system. Also the buffers B1 and B2 can have larger ca-

event	controllable	definition
1	Y	M1 takes a raw workpiece
2	N	M1 sends one workpiece to B1
3	Y	M2 takes a workpiece
4	N	M2 sends a workpiece to B2
5	Y	TU takes one workpiece from B2
6	N	TU releases one workpiece
7	N	TU returns one workpiece to B1

Table 5.2: Event definition in Transfer Line

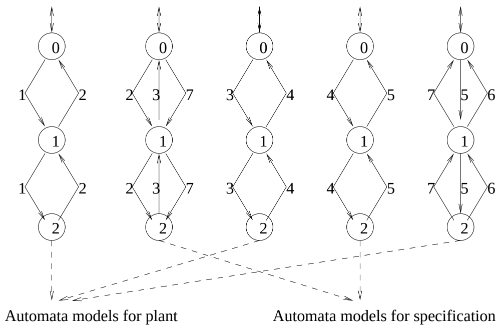

Figure 5.5: Automaton model for Transfer Line with M=2

pacities. If we generalize Transfer Line by allowing everything in it to have multiple instances, it will become more complicated. For example, assuming everything has two instances, we will get the automaton model of Transfer Line as shown in Fig 5.5. For simplicity of explanation, we denote the number of instances as M. Thus the system being studied is denoted as *Transfer Line with M=2* from now on.

Transfer Line can always be extended to allow more instances. The number of states of the system will increase accordingly, and so does the computational burden.

We can also connect arbitrary numbers of Transfer Lines together, as shown in Fig 5.6.

We denote the number of components in the total system by N. For a simple transfer line, $N = 5 = 3 + 2$ (No. of plant modules + No. of specification modules). For a system built by connecting L simple Transfer Lines, we have

$$N = 5L + L - 1,$$

because each Transfer Line has L components and we need $L - 1$ buffers to connect them together.

A performance comparison between STCT and

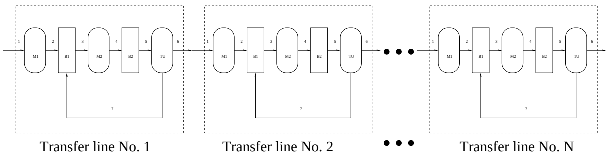

Figure 5.6: Production Line

M	$\|states\|$	CTCT Time	CTCT Mem
1	66	<< 1sec	< 1M
2	4536	0.5sec	2M
3	84800	60sec	10M
4	821250	> 5000sec	120M
5	5.29×10^6	n/a	n/a
7	1.02×10^8	n/a	n/a
10	2.81×10^9	n/a	n/a
M	$\|nodes\|$	STCT Time	STCT Mem
1	19	< 1sec	< 1M
2	36	< 1sec	< 1M
3	57	1sec	1M
4	82	1.5sec	1.5M
5	111	2sec	1.6M
7	181	8sec	2M
10	316	20sec	3M

Table 5.3: Performance of CTCT and STCT vs. M on expanded Transfer Line (L=2)

CTCT is given in Table 5.3.

We can see that CTCT gives up as soon as M reaches 5 while STCT scales well for much larger values of M. If N is larger, CTCT will face an even harder situation. For case $L = 4$ ($N = 23$), CTCT does not work except for the simplest case, $M = 1$. But STCT still works well, as shown in Table 5.4.

The performance improvement given by various types of optimization is shown below.

First let's look at the contribution of variable ordering. Consider the configuration with $L = 2$. Table 5.5 shows the difference between an optimized ordering and a random ordering. Obviously we achieve a big performance gain by a good ordering of the components. We see that this gain becomes larger when the system becomes more complicated.

For a complex system, the incremental approach to increasing M becomes very important. For a system with $M = 10$ and $L = 2$, using the optimal ordering, the variation of $|nodes|$ during the first computation of reachability state space is shown in Fig 5.7. Also, this

M	$\lvert states \rvert$	$\lvert nodes \rvert$	STCT Time	STCT Mem
1	7986	41	6sec	< 1M
2	5.29×10^7	76	11sec	1M
4	2.74×10^{12}	170	62sec	1.8M
7	6.58×10^{16}	371	98sec	4M
10	6.74×10^{19}	644	340sec	8M
15	2.59×10^{23}	1259	955sec	15M

Table 5.4: Performance of STCT vs. M on expanded Transfer Line (L=4)

M	$\lvert states \rvert$	$\lvert nodes \rvert$ (random)	Time (random)	Memory (random)
1	66	41	< 1sec	< 1M
2	4536	115	< 1sec	1.7M
5	5.29×10^6	727	27sec	5M
7	1.02×10^8	1610	70sec	10M
10	2.81×10^9	3947	300sec	30M
M	**$\lvert states \rvert$**	**$\lvert nodes \rvert$ (optimal)**	**Time (optimal)**	**Memory (optimal)**
1	66	19	< 1sec	< 1M
2	4536	36	< 1sec	1M
5	5.29×10^6	111	3sec	1.6M
7	1.02×10^8	181	8sec	2M
10	2.81×10^9	316	20sec	3M

Table 5.5: Performance of ordered and unordered STCT vs. M (L=2)

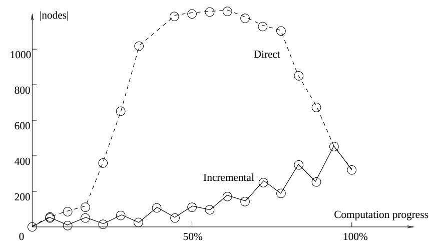

Figure 5.7: Variation of $\lvert nodes \rvert$ during direct and incremental computations

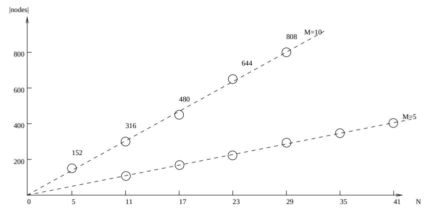

Figure 5.8: $\lvert nodes \rvert$ vs. M and N

	$N = 11$		$N = 23$	
M	$\lvert nodes \rvert$	$\hat{C}$	$\lvert nodes \rvert$	$\hat{C}$
1	19	N/A	41	N/A
2	36	0.92	76	0.89
3	57	1.13	119	1.10
4	82	1.27	170	1.25
5	111	1.35	229	1.33
7	181	1.45	371	1.43
10	316	1.56	644	1.54
15	621	1.66	1259	1.65
30	2136	1.78	4304	1.77

Table 5.6: Fitting of data $\lvert nodes \rvert$ vs. M, (N=11 and N=23)

difference becomes larger as the system becomes more complicated (i.e. as N increases). It is quite difficult to compute the system with $M = 10$ and $L = 4$ directly – the maximum of $\lvert nodes \rvert$ exceeds 11500, i.e. more than 20 times greater than the final $\lvert nodes \rvert$. Actually, for a simpler system such as the one with $M = 10$ and $L = 2$, the incremental approach is slower, because of more 'idle' cycles. However, for a larger system, the incremental approach is much faster.

In the previous section we estimated that

$$\lvert nodes \rvert = O(N \times K^C)$$

This estimate can be tested against the data collected from the Transfer Line example[5].

As we see in Fig 5.8, there is a rather obvious linear relationship between $\lvert nodes \rvert$ and N, i.e,

$$\lvert nodes \rvert \propto N$$

[5] Note that K is the number of states in the automaton while M is the number of instances. For a simple buffer-like model, as Transfer Line, we always have $K = M + 1$.

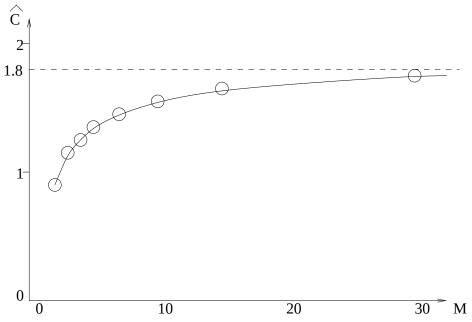

Figure 5.9: $\hat{C}$ vs. M for Transfer Line

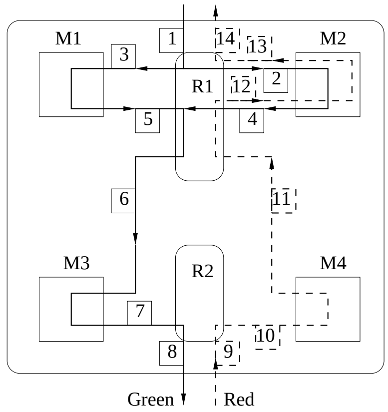

Figure 5.10: Physical layout of Workcell

For a fixed L, i.e. fixed N, given k_1, k_2 and the corresponding $|nodes|$ (denoted $|nodes|_{k_1}$ and $|nodes|_{k_2}$ respectively), we can estimate C by computing [6]

$$\hat{C} = \frac{log(\frac{|nodes|_{k_1}}{|nodes|_{k_2}})}{log(\frac{k_1}{k_2})}$$

For $L = 2, 4$, i.e. $N = 11, 23$, the values of $\hat{C}$ are shown in Table 5.6.

We can see that the values of $\hat{C}$ are pretty much the same for $N = 11$ and $N = 23$, that's to say, C is nearly constant w.r.t. N. If we plot the values as shown in Fig 5.9, we can estimate that $C \leq 1.8$.

From the above analysis, we can see that the estimate

$$|nodes| = O(N \times M^C)$$

fits the empirical data of Transfer Line very well. Although this example is somewhat artificial, it does suggest that it is often possible to find a polynomial relationship between the real complexity of a system and the number of components in it.

5.2 Workcell [7]

Fig. 5.10 shows the layout of Workcell consisting of 4 machines, M1, M2, M3, M4, and 2 robots, R1 and R2.There are two kinds of products being manufactured in Workcell, namely product *Red* and product *Green* .

As we see, Workcell can be easily modeled in SCT. Assuming that every event is controllable,[8] we proceed to compute the supremal nonblocking sublanguage of the system. This works out fine too. Using CTCT, it can be easily computed that the controlled Workcell will have 205 states.

As usual, we use M to denote the number of instances allowed and for simplicity of representation, assum that every component has the same M value.

Again, a larger value of M implies a harder task for CTCT. For $M = 2$, CTCT takes several minutes, hundreds of megabytes of memory and then yields a controlled automaton with 12912 states. CTCT is unable to provide any result when $M = 3$ or larger.

However, STCT can compute the case M=3 in several seconds. Actually, STCT can synthesize the system even when M is as large as 30 (even larger, if necessary). The results of the performance of CTCT and STCT are summarized in Table 5.7.

We can safely say that STCT can deal with the system given any practical setup. Of course, if we put $M = 1000$, STCT would not be able to produce the result in a feasible time, but then it is very unlikely that we will encounter a system with such huge buffers.

5.3 FMS [9]

This section presents a case study of the implementation of an automaton-based controller for the operation

[6] This is obtained as follows. Assuming that $|nodes|_{k_1} = N * (k_1)^{\hat{C}}$, $|nodes|_{k_2} = N * (k_2)^{\hat{C}}$, divide the former by the latter and then apply log to both sides of the equation, to get $log(\frac{|nodes|_{k_1}}{|nodes|_{k_2}}) = \hat{C} \times log(\frac{k_1}{k_2})$. Thus we can find $\hat{C}$ easily.

[7] In Petri net form, this example was introduced in [BA95] and extended by Michael Seidl [private communication to WMW, August 2000]. See also [SS00].

[8] Considering all events in a manufacturing system to be controllable is a common practice in PN literature, although it is not always appropriate. For instance, breakdown events (if any) and operation-completion events could be uncontrollable. Here all events are considered controllable to keep the automaton model comparable to its PN counterpart.

[9] This example is directly adapted from Section 8.2 of [DAJ95] and is the most complex example in that book.

M	$\|states\|$	CTCT Time	CTCT Mem
1	205	$<$ 1sec	$<$ 1M
2	12912	210sec	$>$ 500M
4	1.9×10^6	n/a	n/a
8	7.3×10^8	n/a	n/a
10	5.8×10^9	n/a	n/a
15	2.9×10^{11}	n/a	n/a
30	3.4×10^{14}	n/a	n/a
M	$\|nodes\|$	STCT Time	STCT Mem
1	79	1sec	1M
2	101	2sec	2M
4	192	6sec	3M
8	444	17sec	4M
10	666	30sec	5M
15	1116	78sec	7M
30	3546	691sec	17M

Table 5.7: Performance of CTCT and STCT vs. M on Workcell

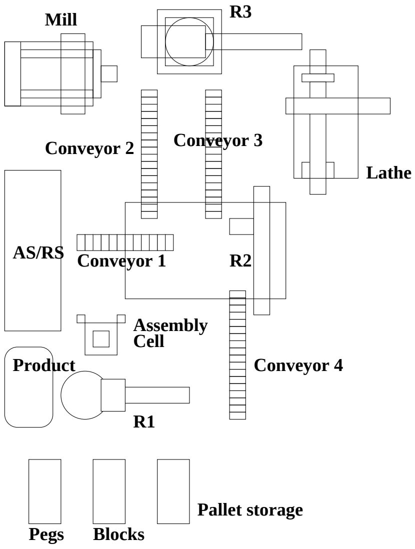

Figure 5.11: A flexible manufacturing system

of a flexible manufacturing system, FMS. The basic approach starts with the development of an automaton model for the system (plant). After that, STCT is used to solve the synthesis problem for the maximally permissive nonblocking controller.

This laboratory system consists of six workstations: a lathe, mill, conveyor system, automatic storage and retrieval system, and two robotic workstations. The system, shown in Fig 5.11, starts with raw stock for two different part-types; it machines one into a cylindrical peg and the other into a block with a hole in it. The finished product is the peg and block mated together. No human operators are required.

We form an automaton model of the system consisting of 21 components. Because all events are controllable, the only additional requirement is nonblocking operation of system. This can be done by STCT without too much difficulty. Because of the complexity of the system, CTCT fails to produce any result. The performance of STCT vs. various sizes of the buffers in this FMS is shown in Table 5.8. In the table, Ma is the sum of the capacity of AS/RS and Conveyor 1 for each component. All the other buffers have size Mb. Mp is the total number of available pallets. The remaining non-buffer-like components are kept fixed.

Ma	Mb	Mp	$\|states\|$	$\|nodes\|$
2	5	6	2.48×10^6	360
5	5	10	2.87×10^8	541
7	5	20	4.21×10^{11}	1346
14	10	40	1.96×10^{15}	2853
Ma	Mb	Mp	STCT Time	STCT Memory
2	5	6	26sec	4M
5	5	10	62sec	6M
7	5	20	272sec	8M
14	10	40	1425sec	18M

Table 5.8: Performance of STCT vs. M on FMS

6 Conclusions

It is easily seen that for complex DES , STCT offers far better performance than the CTCT implementation. Furthermore, STCT is much more systematic, straightforward and general than Petri Net approaches. The STCT algorithm again demonstrates the importance of structural information in a DES. Given the nature of most practical systems, which are highly structured, the computational complexity can be dramatically reduced while the advantages of the algorithm, such as straightforwardness and generality, are undiminished.

References

[Ake78] S.B. Akers. Functional testing with binary decision diagrams. *Eighth Annual Conference on Fault-Tolerant Computing*, pages 75–82, 1978.

[BA95] K. Barkaoui, I. Ben Abdallah. A deadlock prevention method for a class of FMS. *Proc.*

IEEE Int. Conf. on Systems, Man, and Cybernetics, pp. 4119-4124, Vancouver, Canada, October 1995.

[Bry86] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 12(8):677–691, August 1986.

[CL99] C.G. Cassandras and S.Lafortune. *Introduction to Discrete Event Systems.* Kluwer Academic Publishers, 1999.

[DAJ95] A. A. Desrochers and R. Y. Al-Jaar. *Applications of Petri Nets in Manufacturing Systems, Modeling, Control and Performance Analysis.* IEEE, 1995.

[Den89] M.J. Denham. A Petri net approach to discrete event control. In *IEE Colloquium on Modelling, Simulation and Control of Discrete Event Systems*, December 1989.

[GD92] A. Giua and F. DiCesare. On the existence of Petri net supervisors. In *Proc. of 31st IEEE Conf. Decision and Control*, pages 3380–3385, Tucson, AZ, USA, December 1992.

[Ger95] R. Germundsson. *Symbolic Systems - Theory, Computation and Applications.* Linköping Studies in Science and Technology, Dissertation No. 389, Linköping University, 1995.

[Gun97] J. Gunnarsson. *Symbolic Methods and Tools for Discrete Event Dynamic Systems.* Linköping Studies in Science and Technology, Dissertation No. 477, Linköping University, 1997.

[Kro87] B.H. Krogh. Controlled Petri nets and maximally permissive feedback logic. In *Proc. of 25th Allerton Conf. on Communication, Control and Computing*, volume 1, pages 317–326, Monticello, IL, USA, 1987.

[LW93] Y. Li and W.M. Wonham. Control of vector discrete-event systems I – the base model. *IEEE Trans. Autom. Control*, 38(8):1214–1227, August 1993.

[LW94] Y. Li and W.M. Wonham. Control of vector discrete-event systems II – controller synthesis. *IEEE Trans. Autom. Control*, 39(3):512–531, March 1994.

[MA98] J. Moody and P. Antsaklis. *Supervisory Control of Discrete Event Systems Using Petri Nets.* Kluwer Academic Publishers, 1998.

[MP00] H. Marchand, S. Pinchinat. Supervisory control problem using symbolic bisimulation techniques. *Proc. American Control Conference*, pp. 4067-1071, Chicago, June 2000.

[RW87] P.J. Ramadge and W.M. Wonham. Supervisory control of a class of discrete event processes. *SIAM J. Control Optim.*, 25(1):206–230, January 1987.

[SS00] M. Seidl and G. Schmidt. Avoiding deadlocks in flexible manufacturing systems. In *Discrete event systems: analysis and control*, pages 149–158. Kluwer Academic Publishers, 2000.

[Tro97] E. Tronci. On computing optimal controllers for finite state systems. *Proc. 36th IEEE Conf. on Decision and Control*, pp. 3592-3593, San Diego, December 1997.

[UM88] T. Ushio and R. Matsumoto. State feedback and modular control synthesis in controlled Petri nets. In *Proc. of 27th IEEE Conf. Decision and Control*, pages 1502–1507, Austin, TX, USA, December 1988.

[Won99] W. M. Wonham. *Notes on Control of Discrete-Event Systems.* Department of Electrical and Computer Engineering, University of Toronto, December 1999.

[ZD93] M.C. Zhou and F. DiCesare. *Petri Net Synthesis for Discrete Event Control of Manufacturing Systems.* Kluwer Academic Publishers, 1993.

[Zha01] Z.H. Zhang. *Smart TCT: An Efficient Algorithm for Supervisory Control Design.* M.A.Sc. Thesis, Dept. of Electl. & Cmptr. Engrg., Univ. of Toronto, April 2001.

[ZV99] M.C. Zhou and K. Venkatesh. *Modeling, Simulation, and Control of Flexible Manufacturing Systems : a Petri Net Approach.* World Scientific, 1999.