The STS Control Problem Solver Program (stscps)

User's and Programmer's Manuals

Daniel Côté et Richard St-Denis

Département d'informatique
Université de Sherbrooke

May 7, 2012

We want to thank Mr. Benoît Fraikin from the Département d'informatique de l'Université de Sherbrooke for providing the main LaTeX package used to typeset this brochure, and more generally for his assistance in matters of typesetting with LaTeX.

Contents

User's Manual

The STS Control Problem Solver (stscps) is a program that performs the resolution of control problems specified using the STS modeling framework. The main reference for the user's and programmer's manuals with respect to the STS framework is [2] which introduced the nonblocking control of *State Tree Structures* as a major variant of Supervisory Control Theory (SCT).

1.1 Program Usage

The program is invoked at the command line and requires as a parameter a valid Control Problem Model file. The model file is first parsed to verify that its contents defines a valid Control Problem Model. Most of these verifications center around the properties of well-formed STS given at section 2.3 of [2]. If the model file contains a valid Control Problem Model then the program proceeds to apply the synthesis procedures exposed in chapters 3 and 4 of [2] to compute a State Feedback Control Law as a set of predicates f_σ, one for each controllable event σ of the STS. The main output of the program is the set of control predicates f_σ which are simply printed out on the console.

If the command line invocation of the program is incorrect, a short error message is printed (Listing 1.1 below) along with a summary of the program command line usage.

```
$ stscps
### To few arguments.
### Usage is:
  stscps ( <file name> | --stdin ) [ options ]

  When using --stdin, the source model must be piped in instead of giving a
  file name.
  Options:
  --Parse --Semantic --Compile --Solve control the compilation level,
    default is --Solve.
  --Silent --Summary --Verbose control the amount of details given
    during compile, default is --Silent
  --bddSmall --bddMedium --bddLarge --bddBig control the  size of the bdd
    package node table, default is --bddMedium
  Also --bdd n (where n is an integer power of two in the range [14,24])
    can be used.
```

Listing 1.1: stscps program command line

The program requires the name of a file containing a Control Problem Model and can take a few options. Option `--stdin` allows the user to *pipe in* the Model file instead of giving its name as an argument. The other options are divided in three categories:

- compile level control options;
- output detail level control options;
- BDD package run time options.

Compile level control options (`--Parse`, `--Semantic` and `--Compile`) control the compilation level. They are intended to provide assistance to a user that is designing a Control Problem Model while this model still contains errors (syntax or consistency errors). They can be seen as *model debugging* options. In particular they will automatically enable the `--Summary` output detail level so that each phase of the compilation gives a summary of its operation, and they enable the source code of the model file to be listed along with the diagnostics of: the parser, the semantic checker and the model compiler. The default option for that category is `--Solve` which assumes that the Control Problem Model is valid and suppresses as much output as possible by assuming output detail option `--Silent`. Note that any error or diagnostic generated during compilation will always be printed during the compilation phase no matter the compile level or the output detail level requested. Using compilation level of `--Solve`, if the compilation generates no error and no diagnostic, the compilation generates no output.

Output detail level options behave as follow.

- Option `--Silent` suppresses as much output as possible during compilation. If no error and no diagnostic are generated during compilation then compilation is *silent* (generates no output). However, compilation errors or diagnostics are never suppressed.
- Option `--Summary` prints summaries for all phases of the compilation process: parsing, semantic checks and compilation proper.
- Option `--Verbose` is really only a *debugging facility* for programmers to perform diagnostic or maintenance on the program. With the verbose option, all internal data structures of each phase of the compilation are printed along with the summaries for each phase.

Of course, if the Control Problem Model is error and diagnostic free, then the program can compute a solution to the control problem and printing that solution is never suppressed.

BDD package run time options are supplied here only to try to isolate the user from the complexities of configuring the package. The Buddy BDD package requires mainly that a user supplies at least the BDD node table size at package start-up time (routine `bdd_init()` of the package). Heuristic rules are given to size the cache tables relative to the BDD node table size, and other parameters all have a default behavior. Therefore BDD options concern mainly BDD node table size.

- Option `--bddSmall` corresponds to a BDD node table size of 2^{14} nodes;
- Option `--bddMedium` corresponds to a BDD node table size of 2^{17} nodes;
- Option `--bddLarge` corresponds to a BDD node table size of 2^{20} nodes;
- Option `--bddBig` corresponds to a BDD node table size of 2^{24} nodes.

The default BDD node table size corresponds to option `--bddMedium`. Each BDD node in Buddy takes up 20 octets of space, so that the default requires about 2.5 Mb of dynamic memory plus about 10% of that for cache memory. There is also an option `--bdd <int>` taking an integer parameter n in the range $14 \leq n \leq 24$ representing a power of 2 used to calculate the size of the BDD node table so that `--bdd 17` gives the same result as `--bddMedium` (i.e. a BDD node table size of 2^{17} nodes).

1.2 Control Problem Model Syntax

According to [2], abstractly, a control problem is fully specified by the two following objects:

1. the definition of an STS;
2. the definition of $\mathcal{S}$, a set of forbidden state trees for the STS, called the control problem specification.

The STS definition is in turn made of six objects:

1. a set of holons $\mathcal{H}$, an alphabet Σ and a forward transition function Δ;
2. the definition of $\mathbf{S}T$, the state tree describing the structure of the state space of the STS;
3. the definition of $\mathbf{S}T_0$, the set of initial configurations of the STS;
4. the definition of $\mathcal{S}T_m$, the set of final configurations of the STS.

Each OR state of $\mathbf{S}T$ must be *matched* to one and only one the holons in $\mathcal{H}$. By extending the definition of $\mathbf{S}T$ with a description of the holon *matched* to everyone of its OR states (as part of the OR state description), the elements of $\mathcal{H}$, Σ and Δ can be recovered from the extended definition of $\mathbf{S}T$. Consequently, in the syntax adopted here for the specification of control problems, a complete specification is therefore made of the following four elements:

1. the definition of $\mathbf{S}T$;
2. the definition $\mathbf{S}T_0$;
3. the definition of $\mathcal{S}T_m$;
4. the definition of $\mathcal{S}$.

In the proposed syntax, a control problem definition must follow that specific order. The details of the proposed syntax for Control Problem Models are given in the grammar at Appendix A. What follows is a high level summary of it.

1.2.1 Syntactic Assumptions and Inference Rules

In a Control Problem Model, the definition of $\mathbf{S}T$ takes the form of a non-empty flat list of structured state definitions. Structured states here are understood to be OR and AND super-states. The first element of that list is considered to be the root state of $\mathbf{S}T$. Hence, in this syntax, the definition of $\mathbf{S}T$ is not optional. It must contain at least one element, the root, and that element must be a structured state. As a consequence, in this syntax, neither the empty state tree nor the trivial state tree, composed of a sole simple state at the root, can be specified.

The syntax used has been designed to require a minimum of information in order to build a correct STS model and to minimize the potential of error in specification. For that purpose some rules have been built into the syntax to allow the parser to infer as much as possible about the structure of the specified state trees and also about the elements of dynamic associated with OR super-states, the holons.

The first thing to notice about the concrete syntax is that the definition of $\mathbf{S}T$ only contains structured state definitions whereas state trees also contain so called *simple states* (see [2], p. 13, for the formal definition of simple states). This has been done to remove the burden of having to assign unique names to simple states as is required in [2]. Simple states, in a *well-formed* state tree, can only be found as children of OR super-states and their names are usually meaningful only in the context of their parent state. In particular the names of simple states must be unique within the scope of their parent's direct expansion to distinguish between the children of the super-state. But, beyond that, it is often meaningless and cumbersome to distinguish

between two simple states that already belong different parents. Indeed, one is often tempted to re-use the same simple state names for different parents super-states particularly when these names do not have any specific meaning (e.g. q0, q1). There is, however, reasonable ground for assigning unique names to structured states (i.e. names that are unique across the entire scope of an STS). Since OR super-states are assigned variables for synthesis, expressing the results of synthesis using the name of the corresponding OR super-states often makes those results more human-readable. Besides, structured states are usually scarcer than simple states in an STS and often have system-wide significance. Consequent to that, the following naming policy has been adopted and a related inference rule is followed by the parser.

NP1) Structured states must be assigned names (identifiers) that are unique system-wide (across the entire scope of the STS).

I1) Any state name that is not registered system-wide as the name of a structured state is inferred to be the name of a simple state.

As a consequence of this inference rule, any structured state definition bearing the same name as a previous one is diagnosed as redundant and ignored. The parser issues a warning to that effect but continues parsing. However, the user cannot receive any warning about a mis-spelling on the name of a structured state.

In order to make control problem models as compact as possible, the definition of **S***T* has been extended to include holon definitions as part of the definition of the OR super-state to which they are matched. Most information about the structure of a holon is comprised within its transition relation. The transition relation of a holon is partitioned in three transition sets: the boundary input, boundary output and internal transition sets. From these the parser can recover most of the other elements of the holon structure by making the following assumptions.

A1) All states that are either source or target of an internal transition are assumed to be part of X_I, the internal state set of the holon.

A2) All targets of boundary input transitions are assumed to be part of X_0, the initial state set of the holon.

A3) All sources of boundary output transitions are assumed to be part of X_m the terminal state set of the holon.

A4) The entire content of $X_0 \cup X_m$ is assumed to be a subset of X_I, the internal state set of the holon.

A5) All sources of boundary input transitions are assumed to be part of X_E, the external state set of the holon.

A6) All targets of boundary output transitions are assumed to be part of X_E, the external state set of the holon.

A7) All events labeling internal transitions are assumed to be part of Σ_I the internal event set of the holon.

A8) All events labeling boundary transitions (input or output) are assumed to be part of Σ_B, the external event set of the holon.

The only structural element of a holon that cannot be recovered from its transition relation is the partition of its internal event set in controllable and uncontrollable events. For that reason the syntax requires the user to supply the list of internal uncontrollable events of the holon which, of course, are assumed to be part of the internal event set of the holon (assumption **A9**). The parser then makes the following inferences.

I2) Assumptions A1, A2, A3 and A4 allow the parser to infer the contents of X_0, X_m and X_I (the internal state set of the holon).

I3) Assumptions A5 and A6 allow the parser to infer the content of X_E (the external state set of the holon).

I4) Assumptions A7 and A9 allow the parser to infer Σ_I and Σ_{Ic} (respectively the internal event set and internal controllable event set of the holon), Σ_{Iu} (the uncontrollable internal event set of the holon) being given explicitly.

I5) Assumption A8 allows the parser to infer the content of Σ_B (the boundary event set of the holon).

In relation to naming, one of the consequences of these rules and assumptions is that simple state names require qualification when they are used outside of the scope of their parent OR super-state definition. For this reason a limited form of name qualification is defined in the syntax for simple state names.

NP2) Simple state name can be qualified. Only one level of qualification is necessary and allowed and the qualification operator is the *dot*, for example `A.b`. Both *qualifier id* and *qualified id* names must be valid state names and the qualifier must be the name of the parent OR state of the qualified name. Moreover, only simple state names used in the definition of *active state sets* (e.g. in order to define $\mathbf{S}T_0$) need and can be qualified.

1.2.2 Main Constructs of the Syntax

A full example of a control problem model is given by Listing 1.2. This specification contains no error and generates no diagnostic. It contains all four parts of a valid model.

- Lines 13 to 66 contain the definition of $\mathbf{S}T$ with the root defined as AND state `x0` at line 13 to 16.
- Line 77 is the definition of $\mathbf{S}T_0$.
- Lines 82 to 84 is the definition of $\mathcal{S}T_m$.
- Lines 91 to 94 is the control specification $\mathcal{S}$.

The syntax allows both line comments and block comments. Block comments can be embedded. All names and keywords are case sensitive.

Listing 1.2: Example of a Control Problem Model

```
1  /*
2    Ma and Wonham, LNCIS 317 (2005), p. 95 Fig. 4.12
3    Example of an STS with a simple but non trivial vertical structure.
4    The STS itself is not interesting for a Control Problem specification,
5    but in terms of STS structure it covers most of the non-trivial cases and
6    is simple enough that one can visually check the data structures resulting
7    from its compilation.
8  */
9
10 /******************** Definition of ST ************************/
11
12 // ST root state.
13 x0(AND) : {
14   // State direct expansion.
15   E = { x1 x2 }
16 }
17
18 // OR Superstate b1.
19 b1(OR) : {
20   // State direct expansion.
21   E = { b10 b11 }
22   // Matched holon.
23   SIu  = { mu }
24   tBI  = { <a,alpha,b10> }
25   tI   = { <b10,mu,b11> }
26   tBO  = { <b11,beta,c> }
27 }
28
29 // OR Superstate b2.
30 b2(OR) : {
31   // State direct expansion.
32   E = { b20 b21 }
33   // Matched holon.
34   SIu  = {}
35   tBI  = { <a,alpha,b20> }
```

```
36    tI   = { <b20,lambda,b21> }
37    tBO  = { <b21,beta,c> }
38  }
39
40  // OR Superstate b.
41  b(AND) : {
42    // State direct expansion.
43    E = { b1 b2 }
44  }
45
46  // OR Superstate x1.
47  x1(OR) : {
48    // State direct expansion.
49    E = { a b c }
50    // Matched holon.
51    SIu = { beta }
52    X0  = { a } // a
53    tI  = { < a, alpha ,b > < b, beta,c > <c,alpha,a> }
54    Xm  = {a}
55  }
56
57  // OR Superstate x2.
58  x2(OR) : {
59    // State direct expansion.
60    E = { d e }
61    // Matched holon.
62    SIu = { beta }
63    X0  = {d} // d
64    tI  = { < d, beta ,e > < e, sigma,d > }
65    Xm  = { d } // d
66  }
67
68  /******************** Definition of ST0 and STm ************************/
69  /*
70    Initial configuration of the STS given as an active state set.
71    N.B.: In active state set specifications (below). Simple state labels must
72      be qualified with their parent's name in order to be correctly identified
73      since they are not registered at the global (i.e. STS) level here.
74      However structured states (i.e. OR or AND super-states) must use
75      unqualified names.
76  */
77  ST0 : { x1.a b1.b10 b2.b20 x2.d }
78  /*
79    Set of terminal configurations of the STS, given as a list of active state
80    sets.
81  */
82  STm : {
83    { b x1.c }
84  }
85
86  /***************** Definition of the control problem S *******************/
87  /*
88    Control specification:
89      Set of forbidden state trees given as a list of active state sets.
90  */
91  S : {
92    { x1.a x2.d }
93    { x1.a x2.e }
94  }
```

The Abstract Syntax Tree (AST, the output of the parser) is formed of a set of state descriptors, one of which is designated as the $\mathbf{S}T$ root, and three special objects for $\mathbf{S}T_0$, $\mathcal{S}T_m$ and $\mathcal{S}$ which are given as active state sets and lists of active state sets. Therefore the three main constructs of the syntax are: state descriptors, active state sets and transitions. Transitions are part of holon descriptions. All other constructs are collections of these three main constructs. Note that the syntax for the main constructs is strict and assumes a specific order of appearance of their sub-constructs. The parser implements no parse error recovery so that a parse error stops the parse.

To define $\mathbf{S}T$ there is basically one main construct: the state descriptor. The header of the state descriptor identifies the state itself and its type. All structured states have an *expansion* introduced by the keyword `E`, which is formed of an ordered set of labels. A state's expansion can contain labels of other structured states (e.g. `x0` an AND state, lines 13 to 16, refers to OR states `x1` and `x2`). Note that the expansion of a state may contain the labels of simple states (as for OR state `b1`, at lines 19 to 27), structured state labels (as for `x0`) or a mixture of both (as for OR state `x1` at lines 47 to 55).

For AND states, the expansion is the only element of internal structure. For OR states, the expansion must be followed by a description of their *matched holon* (see, for example, OR state `b1` at lines 23 to 26 and contrast it to OR state `x1` at lines 51 to 54). A holon description can start with the keyword `Memory` to indicate that it is the holon of a memory element (e.g. Listing 1.3). If this is the case, the keyword `Memory` must be the first element of the holon description. The other elements are:

`SIu` standing for Σ_{Iu} the uncontrollable internal event set;

`X0` standing for X_0 the initial state set;

`tBI` standing for δ_{BI} the Boundary Input transition set;

`tI` standing for δ_I the internal transition set;

`tBO` standing for δ_{BO} the Boundary Output transition set;

`Xm` standing for X_m the terminal state set.

Note that the syntax precludes specifying both the initial state set X_0 and the boundary input transition set δ_{BI} together within the same holon specification; only one of the two can be specified. The same relationship exists between X_m and the boundary output transition set δ_{BO}. This rule is consequent to the definition of *holon* given in section 2.2 of [2]. Basically if an OR state's holon expands a parent holon (see [2] p. 39 to get a definition of '*expands*'), then δ_{BI} and δ_{BO} are generally not

```
1  // Example of a memory element...
2  mem1(OR) : {
3    // State direct expansion.
4    E = { q0 q1 }
5    // Matched holon.
6    Memory
7    SIu  = { mu }
8    tBI  = { q0 }
9    tI   = { <q0,beta,q1> <q1,mu,q0> }
10   tBO  = { q0 }
11 }
```

Listing 1.3: Example of a memory element

empty and their contents completely define X_0 and X_m respectively. However, when an OR state is AND-adjacent to the root, its external structure is empty and both, δ_{BI} and δ_{BO}, are then empty. In that last case (e.g. `x1` and `x2` in Listing 1.2) the contents of `X0` and `Xm` should be given explicitly by the user since these sets cannot be empty.

Transition sets are made of three-tuples of comma-separated labels enclosed by angle-brackets each representing a transition (e.g. line 53 in Listing 1.2). Their structure is `<source-state-label, event, target-state-label>`. Internal transitions `tI` source and target states must be in the state's expansion. Boundary Input transition sources must refer to external states as well as target states of Boundary Output transitions. External states can be the labels of simple states in other structured states. State labels in transition structures need not be qualified because of the relationship that must exist between structured states in $\mathbf{S}T$.

The definition of $\mathbf{S}T_0$ is given as an active state set (see [2], p. 27, for the formal definition of active state set). The definitions of $\mathcal{S}T_m$ and $\mathcal{S}$ are given as lists of active state sets. The definitions of $\mathbf{S}T_0$, $\mathcal{S}T_m$ and $\mathcal{S}$ are all optional but, when any combination of these three elements is present, it must come after the definition of $\mathbf{S}T$ and it must follow the specific order: $\mathbf{S}T_0$, $\mathcal{S}T_m$ and $\mathcal{S}$. Any one of these can be omitted in a given control problem specification and the omitted elements are considered to be empty sets. Omitting any or all of these elements will not prevent synthesis (control problem resolution) but may give rise to pathological situations since omitting $\mathbf{S}T_0$, for example, makes all states of $\mathbf{S}T$ inaccessible and, conversely, omitting $\mathcal{S}T_m$ makes all states of $\mathbf{S}T$ non co-accessible.

Active state sets are basically given as sets of state names. In active state sets, the name of simple states must be qualified with the name of their OR state parent. This is necessary since simple state names are not registered at the global STS level and are not unique except within the scope of their OR state parent. This is the only circumstance where it is necessary to use qualified names to uniquely identify states. The names of structured states are never qualified, they must be always be

```
1  $ stscps Tests/test.mdl
2  ======================================
3
4    PS : <x1:0>
5
6    C := cupC2P( ! PS ) : <x1:2><x1:1>
7
8    P0 & C : <x1:1, b1:0, b2:0, x2:0>
9
10   -- Non optimized solution ---
11   f_alpha : <x1:0>
12   f_lambda : <x1:1, b2:0>
13   f_sigma : <x1:2, x2:1><x1:1, x2:1>
14
15   -- Optimized solution ---
16   f_alpha : F
17   f_lambda : T
18   f_sigma : T
19
20 ======================================
```

Listing 1.4: Example of a control problem resolution

unqualified identifiers, even in active state sets.

1.2.3 Results Interpretation

The resolution for the problem model given in Listing 1.2 is given on Listing 1.4. The command line (line 1) invokes stscps with the appropriate model file with all default options (compile level `--Solve`, output detail level `--Silent` and `--bddMedium` BDD node table size). The results produced are predicates which are printed by the BDD package itself.

- `PS` stands for the predicate identified with the control specification $P_{\mathcal{S}}$, the family of forbidden configurations of the STS;
- `C` stands for the control (or synthesis) predicate $C := sup\mathcal{C}^2\mathcal{P}(\overline{P_{\mathcal{S}}})$;
- `P0 & C` stands for $P_0 \wedge C$ the predicate identified with the family of allowable initial system configurations under C.

The other predicates represent the control function in non-optimized form (raw predicates such as calculated by the synthesis procedure) and optimized form (simplified predicates such as in [2] pp. 120-122). Predicate computation and optimization (or simplification) is detailed in subsection 2.2.2.

The BDD package provides a way to print variable names instead of their numeric IDs in printed results. One can see in Listing 1.4 that, for example, `PS` is given by the term `<x1:0>` which represents an assignment of value `0` to variable `x1`. Of course,

variable `x1` stands for its namesake at line 47 of Listing 1.2, but the value `0` does not correspond to any value of the *expansion* of state `x1` which are given at line 49 of Listing 1.2 (`E ={ a b c }`). This is because the BDD package does not provide for registering symbolic names for variable values. Values are encoded starting from index 0 and following the same order as in the expansion. For `E ={ a b c }`, 'a' gets the index 0, 'b' gets the index 1 and 'c' gets the index 2. The reader must do the translation himself.

Apart from that, any variable bindings in a list between `<` and `>` stand for a conjunction. For example, `P0 & C :<x1:1, b1:0, b2:0, x2:0>` means

$$P_0 \wedge C \equiv (x_1 = 1) \wedge (b_1 = 0) \wedge (b_2 = 0) \wedge (x_2 = 0)$$

Two or more terms on the same variable such as in `C :=cupC2P( !PS ):<x1:2><x1:1>` stand for

$$C := sup\mathcal{C}^2\mathcal{P}(\overline{P_S}) \equiv x_1 \in \{1, 2\}$$

This is a little inconvenient to read, but in order to do better, one would have to massage the BDD package results significantly which is not a trivial task.

Programmers Manual

This section contains notes on the software architecture of stscps and other subjects useful for someone intending to modify the source code of stscps.

2.1 Software Architecture of the stscps Program

2.1.1 Useful Documentation

The documentation on the *Buddy 2.4* BDD package can be found in PS format in the `doc/` subfolder of the standard distribution of the package. The main document is [1]. The *Buddy 2.4* BDD package distribution can itself be found at http://sourceforge.net/projects/buddy/.

The documentation on the GNU implementation of the STL libraries (`libstdc++` version 6.0) can be found at
http://gcc.gnu.org/onlinedocs/libstdc++/latest-doxygen/,
but a more usable documentation set can be found at
http://www.cplusplus.com/reference/.

The documentation on the *Boost* C++ libraries can be found at http://www.boost.org/. The documentation on the `shared_ptr` class is part of the `Smart Pointers` module documentation.

Apart from that, as mentioned before, [2] is the main reference for everything concerning the supervisory control of STS.

2.1.2 Build Environment

Program stscps has been coded in the *cygwin* environment from *red hat* on the PC of the Reactive Systems Lab of the Université de Sherbrooke (2012-04). The distribution of the cygwin environment used was current at the time and the installation was the default one adding gcc compilers and tools. The version of the compiler is "gcc version 4.5.3 (GCC)" using libstdc++ v. 6.0 for the STL C++ libraries (Standard Template Libraries).

The program is also known to compile (and run) correctly on a sun system (SunOS 5.10) using gcc3 with the compiler version "gcc version 3.4.6". However version 6.0 of the libstdc++ libraries is important as the program may not work with prior versions

of that library. Version 6.0 of libstdc++ represents a major re-haul of the libraries relative to prior versions and that can be considered a real dependency for program stscps.

The program also depends on the following libraries:

- the *Buddy* BDD package version 2.4 from sourceforge (http://sourceforge.net/projects/buddy/);
- the *Boost* C++ libraries, version 1.33.1 or higher (http://www.boost.org/).

The dependency on the *Boost* C++ libraries is thin: only one class is used `shared_ptr` to implement a simple dynamic memory management scheme. All data structures of program stscps have a clear hierarchical aggregation relationship amongst them. So the `shared_ptr` class from *Boost* is used to supply automatic dynamic memory management since it readily implements a reference counting scheme for dynamic memory management and takes care of releasing memory that is no longer used. The dependency on the *Buddy* library is intrinsic for BDD usage.

2.1.3 Functional Units and their Relationships

There are three main functional units to the program:

1. the application driver;
2. the problem model;
3. the model compilation instrumentation.

The application driver basically analyses the command line for consistency, then eventually instantiates a control problem and solves it. The control problem uses the model compilation instrumentation in order to load the textual representation of the control problem model and transform it into data structures suitable for problem resolution. Then the control problem model can apply the procedures of [2] to obtain a solution and list it on the console.

The project comprises twelve source code modules.

AppDrv.cpp Implements the application driver.

TSTS.cpp Represents the control problem model and implements the problem resolution routines.

The rest of the modules implement the compilation instrumentation. Their function is to build a control problem representation that is suitable for the problem resolution procedures implemented by `TSTS.cpp`.

TMdlCompiler.cpp Implements the control problem model compiler. It is a central class that plays the role of interface between the rest of the compilation instrumentation and the control problem model `TSTS.cpp`.

CompilerDS.cpp This modules defines most of the *runtime* data structures involved in problem resolution. Contrary to other source code modules, this module defines more than one data type.

TMdlSemantic.cpp This module implements the semantic verification logic of the compilation process. It mainly verifies the properties defined in [2] for State Trees and State Tree Structures.

TMdlParser.cpp Implements the concrete syntax parser for control problem models (cf. Appendix A).

TAST.cpp This module define all the data structures used to represent the abstract syntax corresponding to Appendix A. This module also defines more than one data type.

TTokenizer.cpp Implements a simple tokenizer for use with the parser.

TToken.cpp Implements the token representation.

TANDState.cpp Implements the representation of AND super-states (a *runtime* data structure).

TORState.cpp Implements the representation of OR super-states (a *runtime* data structure).

TState.cpp Implements an abstract class representing structured states (a *runtime* data structure).

The build `Makefile`, given in Appendix B, compiles and assembles them into an application.

As a rule, one source code module usually defines a single data type or a small set of tightly coupled data types. Exception to that rule are the source modules **CompilerDS.cpp** and **TAST.cpp**. Most data types are prefixed with the capital letter 'T' as in `class TMdlCompiler`. This is done for quick recognition in source code inspections. Most of the attributes and method names follow the usual naming convention of C++. Attributes are prefixed with a lower-case 'f' (for field), e.g. attribute `fSemanticChecker`. Names are usually written in lower-case with a capital letter at the beginning of each word except for the first word (e.g., method `listParentChild`). There are also exceptions for that rule primarily with respect to names of formal

entities defined in [2], for example function `Gamma()`, function `CR()` and function `supC2P()`. The program was coded with the goal that functional objects and formal entities defined in [2] should be recognizable as such in the source code (as much as possible).

2.1.4 The Application Driver

The code for that module should stand for itself and is indeed quite simple. As mentioned previously, the application driver basically analyses the command line for consistency, then eventually instantiates a control problem and solves it.

2.1.5 The STS

The data type `TSTS` is a class that represents the model of an STS. In fact, in the current version of stscps, `TSTS` represents a control problem model because it contains the control specification `PS`. Properly speaking there should be another class to represent a control problem model of which the STS would be an attribute. However, synthesis is very tightly coupled with the STS through the definition of `Gamma` the reverse progression function, so that isolating STS and control problem model in two different data types would have required gymnastics that would have obscured the logic.

Most of the functional entities required for synthesis are defined within the `TSTS` class. In particular the operators `supC2P()` (operator $sup\mathcal{C}^2\mathcal{P}(\cdot)$ in [2]), `Bracket` (operator $[\cdot]$ in [2]) and `CR()` (operator $CR(\mathbf{G}, \cdot)$ in [2]). The reverse progression function `Gamma` ($\Gamma(\cdot, \cdot)$ in [2]) is declared and implemented as an internal compiler data structure in module **CompilerDS.cpp** because it has internal state in the form of a set of transition relations (entities N_σ in [2]).

Module **TSTS.cpp** and **CompilerDS.cpp** are the modules that interface and use the BDD package, other modules do not.

2.1.6 Tokens and the Tokenizer

Most of the logic to identify, extract and classify tokens from the input stream is part of the `TToken` class. Most of its routines are `inline` routines and the structure itself is constant. The source code should stand on its own and is straightforward.

There are primarily two token classes: identifiers and delimiters. Identifiers are defined as case-sensitive alphanumeric strings beginning with an alphabetic character. Delimiters are special characters with specific meaning such as `=`, `{` and `}` to name a few. Delimiters are further sub-classified as keywords as are also a certain number of *reserved identifiers* such as `ST0`, `STm` among others. An exhaustive list of keywords and

token classes can be retrieved from the class implementation, **TToken.cpp**, where they are listed in tables.

There are three special token types. One, the `eos` (for end of stream) token, is there for the tokenizer to be able to recognize the end of the input stream. A second one, the `error` token, is there to be returned by the tokenizer to the parser when the text of the input stream does not match any token class. The `error` token always contain an error diagnostic information attribute to allow the parser to issue a more meaningful diagnostic than a bland '*unrecognized token*' kind of message. Finally, the `nat` (or *not a token*) token, if ever returned by the tokenizer, signals a logic error in the tokenizer itself.

The source code for class `TToken` should stand on its own and is really pretty straightforward.

The `TTokenizer` class implements a simple straightforward tokenizer. It manages the input stream, breaking it in tokens, bypassing comments and illegal characters within it. The tokenizer has an explicit command to acquire the next token in the input stream (the `nextToken()` command). The tokenizer automatically recognizes and bypass line comments, block comments and illegal characters that are present in the input stream as part of its `nextToken()` acquisition logic. Block comments can be embedded. Apart from the logic for bypassing comments and illegal characters, the tokenizer keeps a copy of the last token that has been extracted from the input stream (the *current* token). This current token can be queried for a match on its type or on its text (the `lookup()` queries).

2.1.7 The Model Parser

The parser uses the tokenizer to read an input file (the model of a control problem) and produces five objects as result of the parse:

1. a map containing the state descriptors of the list of structured states making up the state tree of the STS model;
2. a pointer to the state descriptor of the presumed root of the state tree of the STS model;
3. the active state set for $\mathbf{S}T_0$;
4. the list of active state sets making up $\mathcal{S}T_m$;
5. the list of active state sets making up $\mathcal{S}$.

Data structures for $\mathbf{S}T_0$, $\mathcal{S}T_m$ and $\mathcal{S}$ simply use STL as they are only sets of strings or lists of sets of strings. The definitions for state descriptors, transition sets and

the related sets and map are contained in the module `TAST.cpp` or, rather, its header `TAST.h`. This set of data structures can be seen as forming the *Abstract Syntax Tree* (AST) of the concrete syntax used for the specification of a control problem model; Appendix A.

Despite its source code size the parser is a simple straightforward recursive descent parser with no parse error recovery. That means that the parse process is terminated as soon as a single syntax error is detected. In that case the parser issues the best parse error message that it can and aborts the parsing process.

The grammar (Appendix A) is very simple and parsing its syntax does not require any lookahead. The next token, if legal, always determines the next construct and the next action that the parser should take. Therefore the general parsing pattern consists in:

1. *looking-up* the current token in the input stream (matching);
2. if the token is not one of the expected tokens at that point then issue an error message and abort the parse, otherwise do whatever is required and go to the next token;
3. go back to step 1;

until no more token is expected and the end of the input stream has been reached. Of course that takes the form of linear procedures and procedure calls to keep track of progression in the text of different constructs. The hardest part is the necessity of giving the best possible parse error message on syntax error detection.

Apart from syntax error detection, the parser also detects static consistency error and diagnostic (or warning) conditions. On consistency error detection, the parser issues an error message but the parsing process is not terminated if no syntax error is present. Also, some diagnostic conditions must be signalled but do not require terminating the parse as they may not be error as such, e.g. a duplicate state name declaration in the expansion of a structured state. When a diagnostic condition arises, the parser issues a warning message and undertake a recovery action to correct the condition, the parsing process is not terminated.

The parser can be interrogated by the semantic checker (a *friend* class) as to the result of the parse. As a debug facility, all data structures from module **TAST.cpp** can be printed to see if the results of the parsing process are as expected. The parser can print a parse summary and a listing of the source of the model file depending on options given to it in its constructor.

Parse error, diagnostics and consistency error messages are printed on *stderr* (in C++ *cerr*). No matter what the parse options are, none of the error and diagnostic messages printed by the parser can be suppressed; an incorrect model always results in error messages (or diagnostics) being printed.

2.1.8 The Semantic Checker

The semantic checker class `TMdlSemantic` takes the AST such as defined in the parser section (subsection 2.1.7) and applies different tests to it in order to verify that it conforms to properties given in [2].

1. The model is first verified to form a tree structure from the presumed root involving all states of the model. This tree structure is verified to be *well-formed* as defined in [2] definition 2.2 pp. 14–15.

2. Holons *matching* properties, such as defined in [2] definition 2.14 pp. 35–36, are then verified.

3. Holons boundary consistency properties, pp. 38–39, are also verified.

4. The model is also verified to obey the *local coupling* rule (definition 2.16 of [2] pp. 40–41).

5. The STS is verified to be deterministic as part of the parsing process and this property is assumed by the semantic checker if the parser completes without any consistency error.

6. Holons sharing events among them are verified to agree on the controllability status of shared events.

7. In order to insure the consistency of the transition relation of the STS, the model is verified to fulfill the sufficient conditions given in p. 57 of [2] for the *soundness* of function Δ.

As part of this verification process some attributes of the state descriptors are calculated in preparation for the compile phase.

2.1.9 The Model Compiler

The model compiler class (`TMdlCompiler`) plays a pivotal role in the compilation process. As mentioned previously, it isolates class `TSTS` (the real problem solver class) from the parsing and consistency verification. As a result a characteristic of class `TMdlCompiler` is that it shares most of its attributes with other classes. This is a design decision that may be challenged but it was taken in order to minimise the copying of data structures around.

The model compiler receives the AST data structures (source module `TAST.cpp/h`) from the parsing and semantic verification phases, and starting from these, computes a set of *run time* data structures that it hands over to the `TSTS` class. Those *run time*

data structures are defined in source code modules `CompilerDS.cpp/h`, `TState.cpp/h`, `TORState.cpp/h` and `TANDState.cpp/h`.

Here the term *run time* data structures only means that the set of data structures used during control problem resolution is a *lightweight* version of those produced as a result of model parsing. This is due to a design decision made on the basis that BDDs are potentially memory hungry data structures. In order to reduce memory consumption during control problem resolution (where it is necessary to use BDDs), it has been decided to tightly isolate parsing from control problem resolution. After successful parsing and semantic checking have been achieved, the compiler must take the resulting model and produce an entirely different set of data structures that are compact and tailor-made to allow the efficient encoding of the control problem into a form that can be used to solve the problem with BDDs. Once the compiler has performed this task, the compile phase terminates destroying all data structures related to the AST leaving only the so called *run time* data structures. Then, and not before, the BDD package can be initialized and used for control problem resolution.

From chapters 3 and 4 of [2] it is possible to gather what are the required entities for control problem resolution:

1. the tree structure of $\mathbf{S}T$;
2. a predicate encoding (i.e. BDD encodings) of $\Gamma(\cdot,\cdot)$ the reverse transition relation of the STS;
3. predicate encodings, P_0, P_m and $P_{\mathcal{S}}$ respectively, for $\mathbf{S}T_0$, $\mathcal{S}T_m$ and $\mathcal{S}$.

The tree structure of $\mathbf{S}T$ need not be complete at control problem resolution time.

- The containment information is required so the expansion of the structured states must be explicit.
- OR states need to add events (alphabet and uncontrollable event set) information.
- Whereas AND states need to add child states clustering information (see p. 108 of [2], definition 4.4, for information on the notion of *clusters*).

That explains the form of the set of classes used for *run time* representation of $\mathbf{S}T$: `TState`, `TORState` and `TANDState` (in corresponding source modules).

Predicate representation, however, requires special data structures. Apart from the fact that the compiler should not be too tightly coupled with the BDD package (in case the program is eventually ported to other BDD packages), the BDD package is not operational during compilation, hence direct BDD encoding of the predicates is ruled out. The data structures provided by the compiler must contain all the

information necessary for the BDD package to produce correct BDD encodings of the predicates at a latter phase.

In the STS framework predicates are used mainly for two purposes:

1. as characteristic functions identified with state sets (see section 4.2.1 of [2]);
2. to express transition relations in the STS (see section 4.2.1 of [2]).

In their simplest expression, apart from the constants *true* and *false*, such predicates express variable bindings of the form $v_x = q_0$ or $v_x \in \{q_0, q_1, ...\}$ where x is the label of an OR state, and identifiers such as q_0 are the labels of states within the holon matched to x. This kind of predicate, that use a single variable, is enough to express characteristic functions of state sets. However, to express a transition inside a holon, both the source and target states of the transition must be expressed which requires a second variable. As a consequence OR states must be assigned two variables one called the **normal** variable and the second called the **primed** variable. Most predicates, including characteristic functions of state sets, are defined over **normal** variables only. Transition relations are defined over both **primed** and **normal** variables.

Most BDD packages, and *Buddy* in particular, identify BDD variables by integer indexes. Since, at compile time, BDD variable numbers are not known (because the BDD package is not operational) a data structure must be supplied to allow keeping track of BDD variable numbers once they are assigned by the BDD package at problem resolution time (*run time*). For that purpose OR states in $\mathbf{S}T$ are assigned an index at compile time. The order of compile time index assignment is more or less arbitrary. Here indexes are assigned in depth-first pre-order. From the compile time OR state index i, a reference position for the two variables of a node (**primed** and **normal**) can be calculated according to the following formulae:

$$\begin{aligned} \mathbf{primedVar}(i) &= 2 * i \\ \mathbf{normalVar}(i) &= (2 * i) + 1 \end{aligned}$$

The reference indexes can thereafter be used to index a table mapping reference variable numbers to BDD variable numbers. That process is called *variable resolution* and occur in class `TSTS` as an initialization of the BDD package. Variable resolution consists of scanning data structures prepared by the compiler and replacing each occurrence of a reference index by the actual BDD variable number computed at BDD variable assignment which is part of the BDD start-up process.

Apart from variable index mapping, a set of data structures must be provided to encode predicates. A small class hierarchy is provided for that purpose; class `TPPExpr` and its sub-classes (in source module `CompilerDS.cpp/h`). The name `TPPExpr` stands for *Type Pseudo-Predicate Expression*. This set of classes is designed to drive the

computation of basic predicates used in control problem resolution. The compiler builds these expressions for transition relations and for active state sets. Part of the *variable resolution* process is to scan those *Pseudo-Predicates* to replace their reference variable indexes by actual BDD variable number at BDD package start-up time. Once the variable of the predicates have been resolved, a call to method `TPPExpr::eval()` returns a `bdd` structure which is a BDD encoding of the expression.

There are three main types of expressions for `TPPExpr`: conjunction lists, disjunction lists and simple expressions. Conjunction lists are lists of factors that must be combined with the conjunction operator of the BDD package, each factor being itself a `TPPExpr`. Similarly, disjunction lists are lists of terms that must be combined with the disjunction operator of the BDD package, each term being itself a `TPPExpr`. Apart from the constants *true* and *false*, simple expressions are *variable bindings* expressions. Variable bindings are represented as a table of indexes and use the STL data structure `valarray<index_t>`. They represent one or more `<variable,value>` pairs all on the same BDD variable. Suppose `vb` is a variable bindings expression, then `vb[0]` is the BDD variable number and subsequent values `vb[i]` for `i =1, ..., k` represent the associated values. The smallest such expression is a table containing two integers, the first being the BDD variable number and the second its assigned value. Such an expression represents a predicate of the form $v_x = q_0$. More complex expressions of the form $v_x \in \{q_0, q_1\}$ for example would use a table with more than two elements (in the example a table of size 3). All predicates mentionned in item (1) and (2) above can be represented by expressions of type `TPPExpr`. In particular, by checking in [2], one can observe that the negation operator is not needed.

The compiler produces a few other data structures (such as a variable name map) to allow the BDD package to print some results. Also the data structure `TSFBC` is provided to allow storing the result of control problem resolution.

2.2 Special Subjects

The following are notes on some subjects that may require clarification. In particular, *Buddy* is presented as a BDD package but little is said on how it can handle *Finite Functions*. Also, the subject of control function simplifications is not exhaustively covered in [2] and requires some clarification.

2.2.1 Finite Domain Predicates Using *Buddy*

This software uses the BDD package *Buddy* to represent and manipulate predicates which are *finite functions* $f : \mathbb{D}_p \to \mathbb{B}$. Here $\mathbb{D}_p = \{0, 1, ..., p-1\}$ is a finite domain of indexes and $\mathbb{B} = \{0, 1\}$. BDD packages cannot represent directly the domain of finite functions since it is multi-valued as opposed to binary valued. To represent finite

domains they usually *encode* domain values using a set of BDD binary variables. That means that each *finite domain variable* is defined by a set of BDD variables that are logically *grouped* together to represent the *finite domain variable* in question. *Buddy* supplies a *layer*, over the BDD basic layer, that it calls *finite domains*, or `fdd`, in order to simplify the manipulation of finite domains predicates. In order to use that facility one must include the header file `fdd.h`. This header file also contains C++ wrapper constructs that make using many features of the BDD package easier and transparent.

- Many operators are *overloaded*, in particular, BDD negation is invoked through `operator!()`, conjunction is invoked through `operator&()` and disjunction is invoked through `operator|()`.
- Dynamic memory management is taken care of automatically by the C++ wrappers, otherwise the user has to do it manually using routines of the BDD package.

To encode finite domains, *Buddy* uses the *encoding* known as the *least significant bit first* encoding. That means that, for each finite domain variable (for us the variable associated with an OR state), *Buddy* allocates $\lceil log_2(p) \rceil$ *basic BDD variables*, where p is the number of values in the domain of the variable (for us the number of states of the holon matched to the OR state). Of course that means that, for a domain of 17 values, five BDD variables are required to represent the values [0..16] and the values [17..31] are not really used and should not appear in predicates for that finite domain variable. However, values [17..31] may appear, particularly as a result of predicates negation. Such a fact will not alter the results of synthesis but will make reading the results of synthesis a bit harder.

The program uses a very little sub-set of the routines of the BDD package. Initialization of the BDD package must occur first and uses the `bdd_init()` routine called with appropriate parameters.

Immediately after initialization finite domain variables must be allocated using the `int bdd_extdomain( int[] domain_sizes, int array_size )` routine. This routine can allocate more than one finite domain variable in a single call. That is the reason for the `domain_sizes` array parameter. This array must contain the *sizes* of the domain variables being allocated. This corresponds to the parameter p of $\mathbb{D}_p = \{0, 1, ..., p-1\}$ for a specific domain variable. The parameter `array_size` is the number of finite domain variables to be allocated. The function returns the *variable numbers* of the domain variables. These variable numbers are referred to as **BDD variables** in subsection 2.1.9 (the compiler section). These variable numbers are very important. They are the variable numbers that must be used for reference variable number resolution. As such they are collected in the map `fV2BDD` of the `TSTS` class

```
1       bdd TPPSimpleExpr::eval()
2       {
3         bdd result = bdd_false();
4         TVarBinds& simple_expr = *fSimpleExpr;
5         index_t var = simple_expr[0];
6
7           for ( int i = 1; i < simple_expr.size(); i++ )
8           {
9             index_t val = simple_expr[i];
10
11              result = result | fdd_ithvar( var, val );
12          };
13
14        return result;
15      };
```

Listing 2.1: Simple expression BDD encoding

which map reference variable numbers (**primed** and **normal** variables) onto BDD (or more accurately `fdd`) variables assigned by the BDD package.

Orderly call to `bdd_init()` and `bdd_extdomain()` are both performed by method `TSTS::startBDD()`. If the BDD package did not return any error during the call to `TSTS::startBDD()`, then the BDD package has been successfully started, all Pseudo-Predicate variables have been resolved and the model is ready for synthesis. Otherwise, if the BDD package refused to start for some reason, then the BDD package has been orderly stopped and is not working and variable resolution has been canceled and all variables restored to their original *reference* variable values.

Some other uses of the BDD package can be seen in the `TSTS` class where mostly the operators for conjunction, disjunction and negation are used. In particular, functions `TSTS::supC2P()`, `TSTS::CR()` and `TSTS::Bracket()` closely follow their formal definitions in [2], respectively $sup\mathcal{C}^2\mathcal{P}(\cdot)$, $CR(\mathbf{G}, \cdot)$ and $\Gamma(\cdot, \cdot)$.

Other interesting instances of the use of the BDD package can be found in source module **CompilerDS.cpp**. For example the translation of a basic Simple expression (i.e. a variable bindings expression) is illustrated in Listing 2.1. Note that at line 5 the `fdd` domain variable number is copied into `var`. This variable number is used for all variable values of the variable bindings expression. At line 9 a value for the variable `var` is copied into `val` and, at line 11, a predicate for the variable binding `<var,val>` is computed with `bdd_ithvar( var, val )`. Finally it is combined with previous variable bindings (if any) through disjunction `operator|()`. Notice that the result of `bdd_ithvar( var, val )` is an encoding of a prediacte of the form $v_x = val$ provided that `var` contains the BDD variable number corresponding to OR state x. If the Simple expression contained more than one variable bindinds then the effect of method `TPPSimpleExpr::eval()` above is an encoding of a predicate of the form $v_x \in \{...\}$.

Finally the most complex use of the BDD package can be seen in source module **CompilerDS.cpp** in the definition of the data structure corresponding to `Gamma`. `Gamma` is defined as a data structure because it has state in the form of a set of transition relations N_σ. As a result `Gamma` is an object and redefines its `operator()()` with the specific signature of $\Gamma(P, \sigma)$. The text of `TGamma::operator()()` contains the most complex examples of the use of the BDD package.

`Gamma` requires existential quantification (`bdd_exist()`) over variable sets (not only single variables). The function used to obtain variable sets with *Buddy* is `fdd_makeset()` that takes a table of variable numbers as input. Also `Gamma` requires variable replacement (`bdd_replace()`) which in turn requires that sets of paired variables be made through a call to `fdd_setpairs()` that takes two sets of variables to be paired as input (typically **primed** and **normal** variables will be paired for replacement with one another). Pairs must be stored in special BDD package data structures called `bddPair` allocated through a call to `bdd_newpair()` and deallocated through a call to `bdd_freepair()`.

Finally routines `bdd_true()` and `bdd_false()` are also used to get the special constants bddTrue and bddFalse respectively. And shutting down the BDD package is effected through a call to routine `bdd_done()`.

2.2.2 Computations of Control Predicates

The un-simplified control predicates are obtained from the definitions for the deterministic case at pp. 118-120 of [2]. There, predicate N_{good} is specific to a particular event σ, and is calculated from predicate $Next_G(\sigma)$ the predicate identified with all basic state trees which are the target of a transition on event σ. This predicate can itself be obtained from the transition relation N_σ, embedded in Gamma, by removing all the source (i.e. primed) variables from N_σ through existential quantification. Using buddy we have:

```
1       bdd N_good = Gamma.nextG( sigma ) & C; // p. 118
2       bdd f_sigma = Gamma( N_good, sigma ); // p. 120
```

Note that only the deterministic case is covered here. In particular, as said in [2] at p. 120, this definition of f_σ only enables σ at those system configurations that it has to. This is important in the context of f_σ simplification.

2.2.3 Control Function Simplifications

In [2] p. 120 it is suggested that the control function (i.e. the set of predicates f_σ) can be simplified using the `bdd_simplify()` procedure of the Buddy BDD package. What is suggested there is that, under certain circumstances, predicate f_σ can be simplified by restricting its domain to $(C \wedge Elig_G(\sigma)\ )$ where $Elig_G(\sigma)$ is the eligibility predicate of controllable event σ within the STS model. However, a further

example on p. 122 clearly shows that there is more than one case involved and that the simplification predicate must vary in accordance to some set of criteria. Unfortunately a full rationale allowing to identify the useful cases and their corresponding simplification formulae is missing. The following attempts to clarify that situation.

To begin with, one can observe that restricting f_σ to C, the control problem solution space predicate, is always a sound and safe default strategy. Effectively, when the controlled system is started from a valid configuration $b \models C \wedge P_0$, no configuration $b' \models \overline{C}$ should ever be reached while the controlled system is in operation. If ever such a configuration was reached, it would mean that something has gone wrong either with the equipment of the system, or with the models that were originally used for synthesis. Restricting f_σ to C means that the controlled system disables σ for all configurations $b' \models \overline{C}$ which is a safe default strategy. This suggest that the general form of the simplification predicate is $(C \wedge P)$ where P is an event eligibility predicate of some kind. However, the suggestion made on p. 120, $P \equiv Elig_G(\sigma)$, is clearly based on a special case and cannot be relied upon to be adequate in all cases (as is also clearly illustrated on p. 122).

For an STS model, event eligibility can be defined by $Elig_G(\sigma)$, such as given in [2] pp. 47-48. This definition considers the eligibility of event σ involving all holons of the STS model, including all memory elements holons. Therefore $Elig_G(\sigma)$ contains all control constraints on σ which are explicit in memory elements of the STS model. Explicit constraints on controllable events still require the control function to enforce them; it would be unsafe to factor them out of the control function while simplifying it.

In the RW framework there exists another definition of event eligibility that considers event eligibility in the free (uncontrolled) behavior of something called 'the Plant'. There, memory is not modeled explicitly. What are modeled instead of memory elements are control constraints. Therefore, within the RW framework, event eligibility is relative to 'Plant components' and 'Plant components' are loosely defined as any specification element that does not model a control constraint. This notion of event eligibility, call it $Elig_{Plant}(\sigma)$, is very useful since 'Plant components' sharing events are assumed to synchronize on shared events without the intervention of the control function. As a result, since an event σ cannot occur in a controlled system unless it is eligible in 'the Plant', the expression of a control function such as f_σ never needs to contain $Elig_{Plant}(\sigma)$ explicitly in its formulation. This is exactly the form of simplification that is sought for the STS control functions. Both, STS memory elements and RW control constraints, have much in common and, even though there is no notion such as 'the Plant' in the STS framework, a definition analogous to $Elig_{Plant}(\sigma)$ can be developed starting from the definition given for $Elig_G(\sigma)$ in [2] by disregarding STS memory elements in the definition.

Using the definition of $Elig_G(\sigma)$, define $Elig_{Plant}(\sigma)$ for an STS model where

$Elig_{Plant}(\sigma)$ disregards the effect of memory elements on the eligibility of σ. Then, by definition, $Elig_G(\sigma) \implies Elig_{Plant}(\sigma)$. The practical value of this definition of $Elig_{Plant}(\sigma)$ is that it is always safe and sound to simplify f_σ by restricting it to $(C \wedge P)$ where $P \equiv Elig_{Plant}(\sigma)$. The rationale is the same as in the RW framework, an event cannot occur in the controlled system unless it is eligible in 'the Plant', meaning that the system's current configuration b satisfies $Elig_{Plant}(\sigma)$. This restriction factors out $Elig_{Plant}(\sigma)$ from f_σ thus simplifying its expression. It should be mentioned that this simplification of f_σ covers the case illustrated on p. 121 for control function f_1. This may not be obvious at first glance since the suggested simplification predicate is $(C \wedge Elig_G(\sigma))$. However, it should be obvious that when an STS model contains no explicit constraint on a controllable event σ, then $Elig_G(\sigma)$ is the same as $Elig_{Plant}(\sigma)$ since no memory element of the STS is involved in the eligibility of σ. This is the situation that prevails at p. 121 of [2] for control function f_1.

More generally, it can be true that $(C \wedge Elig_G(\sigma))$ is the same as $((C \wedge Elig_{Plant}(\sigma))$ even if $Elig_G(\sigma) \prec Elig_{Plant}(\sigma)$ in the STS model at large. However it is not necessary to test for that specific condition since a simplification on $(P \wedge Elig_{Plant}(\sigma))$ is always sound and safe. The only thing that can happen is that this simplification may not deliver the simplest expression for f_σ.

On the other end of that spectrum, when $(C \wedge Elig_G(\sigma))$ is at least as strong as f_σ, then σ is allowed to happen as soon as it is eligible (within C, the control problem solution space). This condition means that, within the confines of the control problem solution space, $Elig_G(\sigma)$ itself is adequate as a control function because then all control over the occurrences of σ is exercised exclusively by explicit constraints expressed in memory elements of the STS. Under that circumstance, the simplest expression of the control exercised by the system is the restriction of $Elig_G(\sigma)$ (not f_σ) to $(C \wedge P)$ where $P \equiv Elig_{Plant}(\sigma)$. This factors out $(C \wedge Elig_{Plant}(\sigma))$ from $Elig_G(\sigma)$ leaving only the factors involved in explicit control constraints. Fortunately this condition is easy to test since, when $(C \wedge Elig_G(\sigma)) \preccurlyeq f_\sigma$, then $((C \wedge Elig_G(\sigma)) \wedge f_\sigma) \equiv (C \wedge Elig_G(\sigma))$.

Finally, following this overall rationale, a general procedure given in Listing 2.2 can be devised for the simplification of the set of predicates f_σ. The algorithm of Listing 2.2 assumes the usage of *Buddy* and also f_σ such as calculated previously and C the predicate identified with the control problem solution space.

```
1    bdd elig_G_sigma = ...;
2    bdd elig_Plant_sigma = ...;
3    bdd f_sigma_opt = bdd_false();
4
5      if ( ( f_sigma & ( C & elig_G_sigma ) ) == ( C & elig_G_sigma ) )
6        // Explicit control only (within C)...
7        f_sigma_opt = bdd_simplify( elig_G_sigma, ( C & elig_Plant_sigma ) );
8      else
9        // All other cases including no explicit control at all...
10       f_sigma_opt = bdd_simplify( f_sigma, ( C & elig_Plant_sigma ) );
```

Listing 2.2: Control function simplification

Grammar of the Control Problem Model Syntax

Control Problem Model Syntax

```
1
2   Lexical elements:
3     Keywords ::=
4       { "AND" "OR" "E" "Memory" "SIu" "X0" "tBI" "tI" "Xm" "tBO"
5         "ST0" "STm" "S" }
6     Delimiters ::= { '(' ')' '{' '}' '<' '>' ':' '=' ',' '.' }
7     id ::= non-empty string of alphanumeric characters beginning with an
8            alphabetic character and such that it is not in Keywords.
9     qualified_id ::= a string corresponding exactly to the pattern id.id
10                     without any intervening white-space character.
11    N.B.: Identifiers are case sensitive.
12
13
14
15  ControlProblemDefinition ::=
16    STSDefinition
17    | STSDefinition CtrlSpecDefinition
18
19  STSDefinition ::=
20    StateTreeDefinition
21    | StateTreeDefinition SpecialObjects
22
23  StateTreeDefinition ::= SuperStateDefinitionList
24
25  SuperStateDefinitionList ::=
26    SuperStateDefinitionList SuperStateDefinition
27    | SuperStateDefinition
28
29  SuperStateDefinition ::= ANDStateDefinition | ORStateDefinition
30
31  ANDStateDefinition ::=
32    id '(' "AND" ')' ':' '{' StateExpansionDefinition '}'
33
34  ORStateDefinition ::=
35    id '(' "OR" ')' ':' '{' StateExpansionDefinition HolonDefinition '}'
36
37  StateExpansionDefinition ::= "E" '=' '{' UnqualifiedNameList '}'
38
39  HolonDefinition ::= "Memory" HolonStructure | HolonStructure
40
41  HolonStructure ::=
42    InternalUncontrollableEventSet
43    ( InitialStateSet | InputBoundaryTransisionSet )
44    InternalTransitionSet
45    ( TerminalStateSet | OutputBoundaryTransisionSet )
46
47
```

```
InternalUncontrollableEventSet ::=
    "SIu" '=' '{' UnqualifiedNameList '}'
  | "SIu" '=' '{' '}'

InitialStateSet ::=
  "X0" '=' '{' UnqualifiedNameList '}'

InputBoundaryTransisionSet ::=
  "tBI" '=' '{' TransitionList '}'

InternalTransitionSet ::=
    "tI" '=' '{' '}'
  | "tI" '=' '{' TransitionList '}'

TerminalStateSet ::=
  "Xm" '=' '{' UnqualifiedNameList '}'

OutputBoundaryTransisionSet ::=
  "tBO" '=' '{' TransitionList '}'

TransitionList ::=
    TransitionList Transition
  | Transition

Transition ::=
  '<' id ',' id ',' id '>'

SpecialObjects ::=
  InitialConfigurationsDefinition
  | TerminalConfigurationsDefinition
  | InitialConfigurationsDefinition TerminalConfigurationsDefinition

InitialConfigurationsDefinition ::=
  "ST0" '=' ActiveStateSet

TerminalConfigurationsDefinition ::=
  "STm" '=' '{' ActiveStateSetList '}'
  | "STm" '=' '{' '}'

CtrlSpecDefinition ::=
  "S" '=' '{' ActiveStateSetList '}'
  | "S" '=' '{' '}'

ActiveStateSetList ::=
  ActiveStateSetList ActiveStateSet
  | ActiveStateSetList

ActiveStateSet ::=
  '{' NameList '}'
  | '{' '}'

NameList ::=
  UnqualifiedNameList
  | NameList id
  | Namelist qualified_id
  | qualified_id

UnqualifiedNameList ::= UnqualifiedNameList id | id
```

Makefile of program stscps

Makefile of program stscps

```
#
# Makefile for the STS Control Problem Solver: stscps
# First version completed 2012-04-16.
# Author: Daniel Côté from Université de Sherbrooke.
#

########################## Variable definitions
#
# Adjust the following to the appropriate distribution of gcc for the
# target machine.
#
# Here the standard distribution of gnu c++ is assumed.
#
#  The application was designed under cygwin (2012-04) with the standard
#  distribution of gcc which, at the moment, is gcc version 4.5.3 and uses
#  the stdc++ v. 6.0 library (for STL among other things).
#  It requires:
#  - a good version of the Standard Template Library (STL);
#  - the Buddy BDD package distribution (buddy-2.4.tar.gz from
#   http://sourceforge.net/projects/buddy/);
#  - the Boost c++ libraries distribution (version 1.33.1 or higher from
#    http://www.boost.org/).
#  Only the 'shared_ptr' data type from the Boost libraries is used in the
#  application to implement a simple dynamic memory management scheme
#  (i.e. hierarchical data structure containment with reference counting).
#
CC = c++
INCL = -I/usr/local/include
LIBS = -L/usr/local/lib -lbdd -lstdc++

# The following works on host mirka.dmi.usherb.ca for the gcc4 distribution.
# To compile using gcc 4 on that host, comment the previous definitions and
# uncomment the following definitions...
#
#CC = /opt/csw/gcc4/bin/c++
#INCL = -I/usr/local/include -I/opt/csw/include
#LIBS = -L/usr/local/lib -lbdd -lstdc++
#

# Object code modules...
BINS =\
 AppDrv.o\
 TSTS.o\
 TMdlCompiler.o\
 CompilerDS.o\
 TMdlSemantic.o\
 TMdlParser.o\
```

```
48   TAST.o\
49   TTokenizer.o\
50   TToken.o\
51   TANDState.o\
52   TORState.o\
53   TState.o
54
55  ########################## Target definitions
56  #
57  all: stscps
58   ls -al stscps*
59
60  test: stscps
61   stscps ./Tests/test.mdl --Compile --Verbose
62
63  SF1test: stscps
64   stscps ./Tests/SF1.mdl --Summary
65
66  TLtest: stscps
67   stscps ./Tests/TL.mdl --Summary
68
69  stscps: $(BINS)
70   $(CC) -o stscps $(BINS) $(LIBS)
71
72  clean:
73   rm -f *.o stscps stscps.exe *.stackdump core
74
75  #-----------------------------------------------------------------------
76
77  AppDrv.o : AppDrv.h TSTS.h
78
79  TSTS.o : TSTS.h TMdlCompiler.h CompilerDS.h TMdlSemantic.h TState.h\
80   TANDState.h TORState.h
81
82  TMdlCompiler.o : TMdlCompiler.h CompilerDS.h TAST.h TSTS.h\
83   TMdlSemantic.h TState.h TANDState.h TORState.h
84
85  CompilerDS.o : CompilerDS.h TState.h
86
87  TMdlSemantic.o : TMdlSemantic.h TMdlParser.h TAST.h
88
89  TMdlParser.o : TMdlParser.h TAST.h TTokenizer.h TToken.h
90
91  TAST.o: TAST.h TState.h
92
93  TTokenizer.o : TTokenizer.h TToken.h
94
95  TToken.o : TToken.h
96
97  TState.o : TState.h
98
99  TANDState.o : TANDState.h TState.h
100
101 TORState.o : TORState.h TState.h
102
103 #-----------------------------------------------------------------------
104
105 .cpp.o:
106  $(CC) $(INCL) -c $*.cpp
```

Bibliography

[1] J. Lind-Nielsen. *BuDDy: Binary Decision Diagram Package, Release 2.2*. IT-University of Copenhagen (ITU), Copenhagen, 2002.

[2] C. Ma and W. M. Wonham. *Nonblocking Supervisory Control of State Tree Structures*, volume 317 of *Lecture Notes in Control and Information Sciences*. Springer-Verlag, Berlin, 2005.