2008 American Control Conference
Westin Seattle Hotel, Seattle, Washington, USA
June 11-13, 2008

A formal framework of reconfigurable control based on model checking

He-xuan Hu, Anne-lise Gehin, and Mireille Bayart

Abstract— **This paper proposes a formal framework for reconfigurable control, based on model checking. This framework first generates a flexible model (i.e., an execution structure) according to the diagnosis, then defines a temporal specification language to deal with the problems due to infinite execution cycles and non-determinism, and finally provides the algorithms that will automatically verify whether the updated model satisfies the desired specification.**

I. Introduction

Dynamic reconfigurable control is a field of fault tolerant control that has emerged over the past decade [1]. The reconfigurability is defined as the possibilities the system has, despite the occurrence of faults, to fulfill its missions without (unbearable) loss of performance. A first class of approaches has been devoted to the design of reconfigurable systems, e.g., [2], [3], [4] and [5] and is in the most of cases limited to the selection or the redesign of the input vector, the output vector, the control law and the set-point, making them inappropriate in the case of a complete loss of an actuator. So a second class of approaches has been developed to analyze system's reconfigurability from the system's functional redundancies [6], [7]. However these approaches do not allow the reconfigurability calculation by the system itself.

That's the reason why, we propose in this paper, a framework to automatically calculate reconfigurability qualitatively. This framework rests on three elements: 1) an appropriate formulation of the system's proprieties allowing the system to update its description when faults occur, 2) a specification language for describing the system's objectives, 3) a verification method to establish whether the current system description satisfies the specification.

The authors are members of Laboratoire d'Automatique, Génie Informatique & Signal, UPRESA CNRS 8021, Université des Sciences et Technologies de Lille. Address: Bat. Poly-Tech Lille, Cité Scientifique, 59655 Villeneuve D'Ascq Cedex, France. Email: {He-Xuan.Hu@polytech-lille.net, Anne-Lise.Gehin@polytech-lille.fr, mireille.bayart@univ-lille1.fr }

The present research work has been supported by Science and Technology for Safety in Trans-portation, and founded by the European Union, the Délégation Régionale à la Recherche et à la Technologie, the Ministère Délégué à l'Enseignement Supérieur et à la Recherche, the Région Nord Pas de Calais and the Centre National de la Recherche Scientifique.

Figure 1 shows the architecture of the proposed method for reconfigurable control. For a given series of observations and diagnoses, if there are faults in system, then a revised system model is constructed flexibly, according to the diagnoses, observations and available system descriptions. Next, model checking is applied to verify whether or not the current system model satisfies the desired objectives. This automatic reconfigurability calculation is one of the most important steps in our framework. If the given control objectives are achievable, the system will run according to the observations and current system model. Otherwise, the reconfiguration is considered as failed.

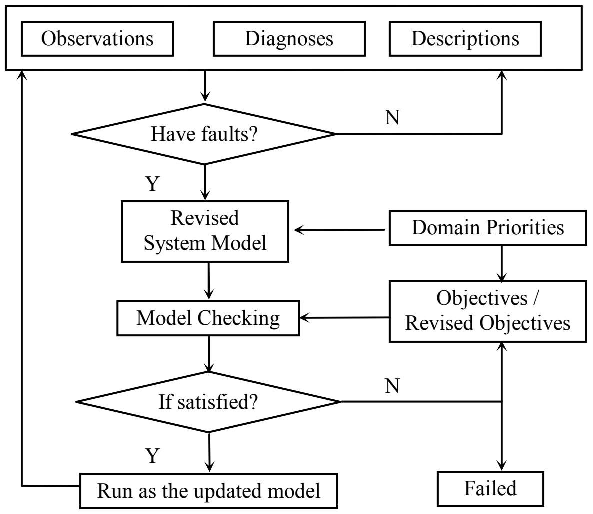

Fig. 1. The reconfigurable control architecture

The paper is organized as follows. In section II, the framework for flexibly generating current system model is described. The temporal logic used for describing extended goals is introduced in section III. The model checking method used to automatically calculate reconfigurability is presented in section IV. Section V illustrates the approach on an example. Section VI summarizes the work done and discusses directions for future research.

II. Flexible System Modelling

The faults considered in this paper are faults which totally change the system's physical structure, such as the complete loss of an actuator or the complete loss of a system component. The system model is no longer valid in this case. This is the reason why we propose a flexible system modeling mechanism to automatically update the system model when faults occur. The mechanism rests on two parts: a database containing basic knowledge about the system, and an algorithm for constructing a system model based on that knowledge.

A. System Knowledge

The basic system knowledge of the system corresponds to the first block at the top of the chart in figure 1 (i.e.: observations, diagnoses, description). These elements refer to three different kinds of knowledge: variable knowledge, semi-variable knowledge, and fixed knowledge. The first includes such information as observations, which vary according to the execution of control commands and environmental changes; the second includes such information as diagnoses, which only change if the system moves from the normal to faulty operation mode and vice versa, and the last includes such information as system descriptions and domain priorities, the non-changing information that expresses the intrinsic system properties as intended by the designers. Domain priorities allow expressing the possibility to achieve "degraded" or non-nominal objectives in faulty modes.

To formalize the system's knowledge, we propose to use a state transition representation STRIPS (STanford Research Institute Problem Solver) [18] where it provides information about the event's pre-conditions and effects. The system knowledge is represented as logical atoms derived from first-order logic. Each state corresponds then to a set of logical atoms (e.g., observations, diagnoses and system descriptions) that are either true or false within a certain degree of interpretation, the transitions are the control actions that change the truth values of these atoms.

Definition 1 (The system model): Let $L = \{P1, P2, \dots Pn\}$ be a finite set of logical atoms. With this set, a system can be represented as a state-transition system. Formally, it is a 3-tuple $\Sigma = (S, A, \gamma)$, where:

(1) $S = \{s1, s2\dots\} \subseteq 2^L$ is a finite or recursively enumerable set of states. Each state s is a subset of L. Intuitively, s tells us which logical atoms currently hold true. If $p \in s$, then p holds true in the state represented by s, and if $p \notin s$, then p does not hold true in the state represented by s.

(2) $A = \{a1, a2\dots\}$ is a finite or recursively enumerable set of control actions. Each control action $a \in A$ is a multiple of subsets of L, which can be expressed as a = *(preconditions, determin-effects, nondetermin-effects)*. As the term suggests, the set of *preconditions* is called the preconditions of a, and the set of *determin-effects (nondetermin-effects)* represents the deterministic (resp. non-deterministic) effects of a. Non-deterministic effects represent an unknown, since it is not known which of them will actually take place. We assume that for any a, any nondeterministic effect is consistent with the deterministic effects (i.e., no single atom and its negation exist simultaneously in the effect sets of a).

(3) $\gamma: S \times A \rightarrow 2^S$ is a state-transition function. $\gamma(s, a)$ = *determin-effects(a)* $\bigcup$ *nondetermin-effects(a)* if $a \in A$ is applicable to $s \in S$, otherwise, $\gamma(s, a)$ is undefined. In other words, whenever an action is applied to a state, it produces another state. This is useful information because once A is known, S can be specified by giving just a few of its states.

This model allows describing actions in terms of their preconditions and effects and describes the states as conjunctions of positive literals. The precondition states what must be true in a state before an action can be executed. The effect describes how the state changes when the action is executed. An action is 'applicable' in any state that satisfies the precondition; otherwise, the action has no effect. It should be noted that γ is defined for one action but the action is defined non-deterministically as above. In other words, when an action is applied to a state, it produces one or several follow-up states. That is the reason why $\gamma(s, a)$ is introduced as a set in the definition of the model.

The list of the possible states can be generated from the Cartesian product of the diagnosis atoms, the control-command atoms and the observation atoms. The initial state is one of these possible states and is captured by the diagnoser and the observer at the moment that the fault report is received. The interesting succeeding states are generated automatically by the algorithm presented in the next section.

B. System modeling Algorithm

```
Algorithm 3.1: Expand $(S, A, T, s_0)$
1 $S \leftarrow s_0$; $Old \leftarrow s_0$; $T \leftarrow \emptyset$; $j = 0$;
2 While $Old \neq \emptyset$
3   { $New = \emptyset$;
4     Repeat
5       { Non-deterministically choose a state
          $s_{old}$ of Old;
6         $s_{old} \in Old$; $Old = Old - s_{old}$;
7         $Com \leftarrow \{a \in A \mid preconditon(a) \subseteq s_{old}\}$;
8         $Reach = \{s_r \mid s_r \in \gamma(s_{old}, Com)\}$;
9         For each $s_r$
10          { if $s_r \notin S$ then
11              { $s_{j+1} = s_r$;  $S = S + s_{j+1}$;
12                $t_{j+1} = (s_{old}, a, s_{j+1})$; $T = T + t_{j+1}$;
13                $New = New + s_{j+1}$; $j = j+1$;}
            }
14        Until $Old = \emptyset$}
15    $Old = New$;}
16 End.
```

From an initial state identified by the diagnoser and the observer, the proposed algorithm returns an automaton *(S,* *A, T, s_0)* as the updated model, where *S* is the set of states, *A* is the set of available control commands, *T* is the automata's transition set, and s_0 is the initial state. The algorithm performs a procedure close to iterative deepening, discovering at each step of iteration, a new part of the state space. In the algorithm, *t* is a element of *T*; *Old* is the set of states that are checked at the current iteration, while *New* is the state set for next iteration; *Com* is the set of acceptable control commands whose preconditions belong to the current checked state sold (line 7). *A* set of states that can be reached from the state sold is found (line 8) by applying the commands *Com*. Then, each of these reachable states is checked to see whether or not it is new to this automaton (line 10). If it is, then it will be added to the set *New* for next iteration (line 11-13). If there is no new state (line 2, 15), then the algorithm terminates.

III. CTL* FOR GOAL FORMULATION

Once the revised system model is obtained, the next step will consist in checking whether the revised model satisfies the system's goals. For the formulation of the system's goals, two conditions have to be taken into account: 1) many systems are designed not to terminate, such as the cycle execution; 2) systems present a non-determinist part in the sense where the effects of a control action are linked to the actual state of the system and to the dynamic of the system.

Taking account of these two conditions, the computation tree logic CTL* [8] seems to be an appropriate tool for the system's goal formulation. In CTL*, time is not mentioned explicitly, the present time instant corresponds to the current state and next time instant corresponds to the state following immediately after. CTL* uses 'temporal operators' and 'path quantifiers' to generate formulae taking the cycle execution and non-determinism into account.

- Path quantifiers are used in a given state to specify that all or some of the paths starting at that state have certain properties. Here, a path is an infinite sequence of states. Two types of path quantifiers, *'A'* and *'E'*, are possible:

(1) '*A'* is a universal path quantifier, meaning that certain properties hold true on all paths starting from a given state.

(2) '*E'* is an existential path quantifier, meaning that certain properties hold true on some paths starting from a given state.

- Temporal operators describe path properties. There are five basic types of operators:

(1) *'X'* (next time) requires that a property hold true in the path's second state.

(2) *'F'* (eventually or in the future) is used to affirm that a property will hold true at some state on the path.

(3) *'G'* (always or globally) specifies that a property holds true at every state on the path.

(4) *'U'* (until) holds true if there is a state on the path in which the second property holds true, and if the first property holds true at every previous state on the path.

(5) *'R'* (release) requires that the second property holds true along the path, up to and including the first state where the first property holds, although the first property is not required to hold true in the future.

Definition 2 (CTL*): There are two types of formulae in CTL*: state formulae, which are true in a specific state, and path formulae, which are true along a specific path. Let *AP* be the set of atomic propositions. The goal language CTL* is defined by the following rules:

(1) If $p \in AP$, then p is a state formula.

(2) If f and g are state formulae, then $\neg f, f \wedge g$ and $f \vee g$ are state formulae.

(3) If f is a state formula, then f is also a path formula.

(4) If f is a path formula, then $E\ (f)$ and $A\ (f)$ are state formulae.

(5) If f and g are path formulae, then $\neg f, f \wedge g, f \vee g, X\ (f), F\ (f), G\ (f), f\,U\,g$ and $f\,R\,g$ are path formulae.

CTL* formulae allow to specify different system's requirements expressed, for example as: (1) reachability goals, such as *EF (g)*, which requires that the system may be able to reach desired states where *g* holds true and *AF(g)*, which requires that the system will be guaranteed to reach those desired states; (2) safety goals, such as *AG (¬g)*, which means *g* must absolutely be avoided and *EG (¬g)*, which means that an attempt must be made to avoid *g*; and (3) maintainability goals, such as *AG (g)*, which means *g* must be maintained and *AF (AG (g))*, which means that the system will always reach some future state from which g can be permanently maintained.

As shown in figure 1, automatic reconfiguration is a complex procedure during which the goals can be changed at the moment that the fault report is received. For a desired property *g*, the temporal goal should be *AG (g)* (i.e., *g* always holds true) in the normal operating mode. If there is a fault that causes *g* to deviate from its desired value, then the reconfiguration will correct this deviation and will keep *g* within its desired value range. But in a non-deterministic system, a control action sent by the reconfiguration procedure cannot be guaranteed to produce the desired effects (i.e., the original goal cannot be guaranteed). This situation can be described as *EF (AG (g))*, which means that *g* will eventually be accomplished at some future state from which *g* will be permanently maintained. However, this does not satisfy the requirements of some high security system. Thus, *EF (AG (g))* must be changed into a strong solution, such as *AF (AG (g))*. If the execution structure satisfies the goal *AF (AG (g))*, then the reconfiguration will be successful in spite of non-determinism. All that remains to be done is to verify whether an execution structure satisfies the temporal goal, and next section explains how this can be done.

IV. THE MODEL CHECKING

Model checking aims to verify whether the revised system's model satisfies the system's objectives. System's objectives are expressed by CTL* formulae. System's model is given by the execution structure (S, L, T, s_0) which is an extension of the automaton $\Sigma = (S, A, \gamma)$ obtained as a result of the algorithm 2.1. In this new structure A is replaced by L, a finite set of logical atoms, T is the state transition function and s_0 is the initial state. Each state is labelled with a set of atomic propositions $L(s)$, which contains all atoms true in that state. The model checking task is then, to determine which states in S satisfy the goal formula ψ.

The model checking rests on a labelling algorithm. The principle of this algorithm is given here [9]. Let $M = (S, L, T, s_0)$ be the execution structure for which we want to determine the states in S which satisfy the goal formula ψ. First, the goal formula ψ is pre-treated to be written in terms of the connectives $\neg$, $\wedge$, $\perp$, EX, EG, and EU using the following equivalences:

(1) $AX(f) = \neg EX(\neg f)$
(2) $EF(f) = E(True\ U f)$
(3) $AG(f) = \neg EF(\neg f)$
(4) $AF(f) = \neg EG(\neg f)$
(5) $A[f U g] \equiv \neg E[\neg g U (\neg f \wedge \neg g)] \wedge \neg EG(\neg g)$
(6) $A[f R g] \equiv \neg E[\neg f U \neg g]$
(7) $E[f R g] \equiv \neg A[\neg f U \neg g]$

Second, the states of M are labelled with the sub-formulae of ψ, starting with the smallest sub-formulae and working recursively towards ψ. The rules to perform the labelling are the following:

If ψ is:

(1) $\perp$: then no states are labelled with $\perp$;
(2) p: then label s with p if $p \in L(s)$;
(3) $\psi_1 \wedge \psi_2$: label s with $\psi_1 \wedge \psi_2$ if s is already labelled both with ψ_1 and with ψ_2;
(4) $\neg\psi_1$: label s with $\neg\psi_1$ if s is not already labelled with ψ_1;
(5) $EG(\psi_1)$:
— Label all the states with $EG(\psi_1)$;
— If any state s is not labelled with ψ_1, delete the label $EG(\psi_1)$;
— Repeat: delete the label $EG(\psi_1)$ from any state if none of its successors is labelled with $EG(\psi_1)$; until there is no change.
(6) $E[\psi_1 U \psi_2]$:
— If any state s is labelled with ψ_2, label it with $E[\psi_1 U \psi_2]$;
— Repeat: label any state with $E[\psi_1 U \psi_2]$ if it is labelled with ψ_1 and at least one of its successors is labelled with $E[\psi_1 U \psi_2]$, until there is no change.
(7) $EX(\psi_1)$: label any state with $EX(\psi_1)$ if one of its successors is labelled with ψ_1.

As $AF(\psi1)$ is often used in practice, the following algorithm to deal with it directly.

(8) $AF(\psi_1)$:
— If any state s is labelled with ψ_1, label it with $AF(\psi_1)$.
— Repeat: label any state with $AF(\psi_1)$ if all successor states are labelled with $AF(\psi_1)$, until there is no change.

V. THE TWO TANK EXAMPLE

To illustrate the proposed approach, let's take an example. The chosen example comprises a level regulation process involving two identical connected tanks (Fig. 2). The inflow *Qp* is provided by pump *P1*. The flow *Qv* between the two tanks is controlled by valve *V1*, with *V2* as a backup valve that should always be closed during normal behaviour. The connecting pipe is at a level of *30 cm (*resp. *0 cm)*. The valve *Vo*, which is always open, is an outlet valve, located at the bottom of tank *T2*. In this example, it is assumed that all the valves are on/off valves, all the pipes have the same diameter, and the flow rate delivered by *P1* is equal to the flow rate through *V1* as the water level of tank *T1* is *45 cm*.

Tank *T1* is equipped with a continuous level sensor and an on/off controller to regulate the level of water, keeping it between *45 cm* and *50 cm* (*25 cm* and *30 cm* in the event of a fault in tank *T1*). The controller turns on the pump when the water level is at *45 cm* (*25 cm*) and turns it off when level reaches *50 cm* (*30 cm*). Tank *T2* is also equipped with a continuous level sensor. Water is fed into tank *T2* from tank *T1* via valve *V1* (*V2* in the faulty case). The controller turns on the valve (*V1* or *V2*) if the level falls below *9 cm* and turns it off when level rises above *11 cm*.

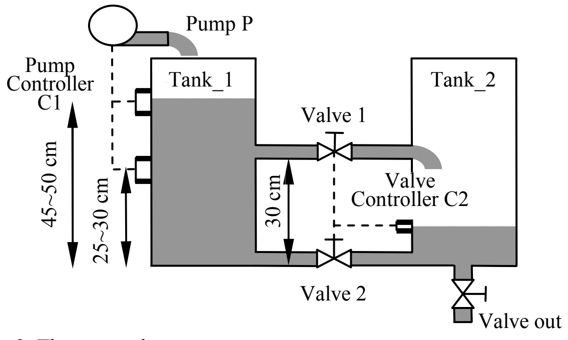

Fig. 2: The two-tanks system

Though this example is small, its behaviour is not simple. For example, the flow between tank *T1* and *T2* varies according to the different water levels in each tank. Thus, during the regulation process, the same valve action results in different water level changes in each tank. This is a non-deterministic property. This regulation process is designed not to terminate, and its desired goals are temporally extended.

Using the formalism proposed in section II, a possible representation of the basic system knowledge is:

L = {****Diagnosis atoms***:

Normal (x): the component x is normal;

¬Normal (x): the component x is abnormal, as is the diagnosis;

Block-on (x): the component x is blocked in the 'on' position (i.e., can not be closed).
Block-off (x): the component x is blocked in the 'off' position (i.e., can not be opened).
***Observation atoms**:*
On (x): the component x (e.g., a valve) is opened;
Off (x): the component x (e.g., a valve) is closed;
L1, L1(x), L1(x: y): the level of tank1 is x or between x and y;
L2, L2(x), L2(x: y): defined as L1
Rise (Li) / Fall (Li): increase or decrease current Li to the limits;
Static (Li): Li does not change;
***The system priorities**:*
(1) if Normal (V1) and Normal (V2), then Goal (L1 (45:50));
(2) if ¬Normal (V1) and Normal (V2), then Goal (L1 (25:30));
Goal (Li(x)/Li(x:y)): goal of Li is x or between x and y;
S = {s0, s1, …sn}, where: for example,
s0 = {L1(30: 50), L2(0: 9), Off (V2), On(P1), ¬Normal(V1), Normal(V2, T1, T2, P1)}…

The list of the possible states can be generated from the Cartesian product of the diagnosis atoms, the control-command atoms and the observation atoms.

A = {Open (V1), Close (V1), Open (V2), Close (V2), Open (P1), Close (P1)}, Where: for example,

```
Open (V2) {
Preconditions: ¬Normal (V1) ∧ Normal (V2) ∧ Off (V2)
               ∧ (L2 ≤ 9);
Determin-effects: On (V2)
Nondermin-Effects:
  if (L1-L2)>L2 then Rise(L2) else Fall (L2);
  if Off (P1) then Fall (L1)
         else in case {(L1-L2)<15 then Rise (L1);
                       (L1-L2)=15 then Static (L1);
                       (L1-L2)>15 then Fall (L1)};}
```

The other commands are not given here due to space limitations.

The model generated from the previous description is shown in figure 3. The goal of the normal operating mode is to maintain the level of tank *T2* between *9* cm and *11* cm and to try to maintain the level of tank *T1* between *45* cm and *50* cm. This temporal goal can be described as $AG\ (9 \leq L2 \leq 11) \wedge EG\ (45 \leq L1 \leq 50)$. The fault scenario is that the valve *V1* is blocked in the closed position, thus *L1* will be below *9* cm and the goal of tank *T1* will be changed into $EG\ (25 \leq L1 \leq 30)$ according to the system priorities. The goal of the reconfiguration can be expressed as $AF\ (AG\ (9 \leq L2 \leq 11) \wedge EG\ (25 \leq L1 \leq 30))$. As valve *V1* is supposed to be blocked in the closed position, the control actions associated with valve *V1*, such as *Open(V1)* and *Close(V1)*, are removed and be made unavailable for generating the revised model. An illustration, shown in figure 4, explains the first step of the model generation. The acceptable control actions are *Close (P1)* and *Open (V2)*. Applying the acceptable control actions (i.e., *Close (P1)* and *Open (V2)*), the reachable states from initial state *1* are states *2, 3, 4* and *5.* The effects of these two commands are *Rise (L2)* and *Fall (L1)*. *Rise (L2)* brings the level *2* to *11* cm and triggers other commands in a new state. *Fall (L1)* decreases level *1* to make it equal to the increase in level *2* triggered by *Rise (L2)*. According to the current *L1* and its decrease, the new level *1* could now possibly belong to four different zones (i.e., *25, 25~30, 30* and *30~50*).

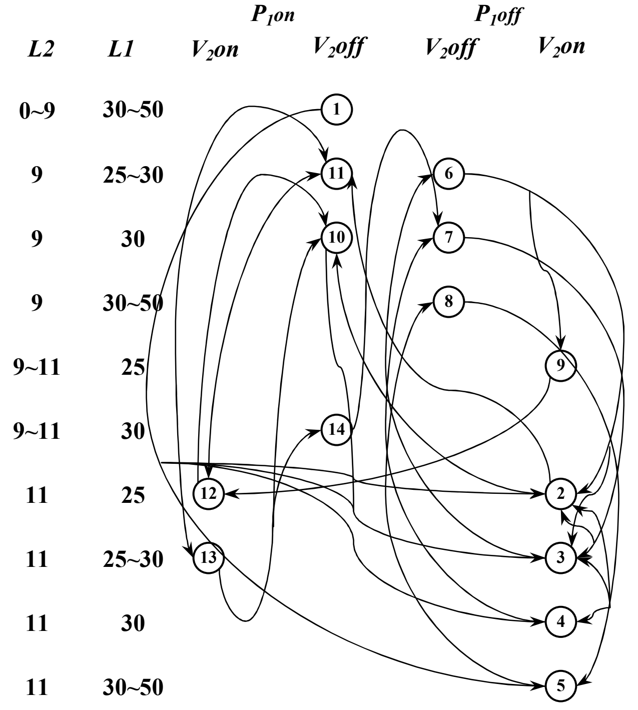

Fig. 3. The model generated using the knowledge contained in Example

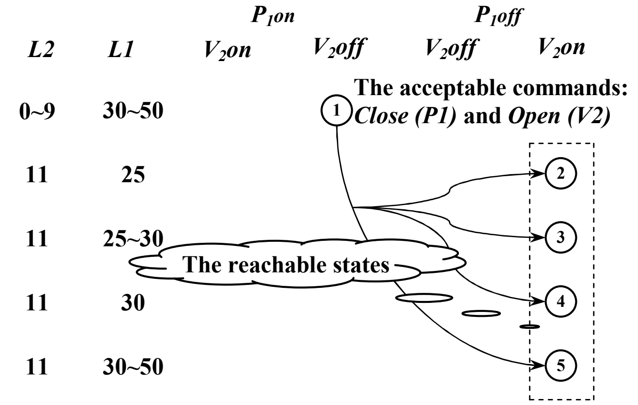

Fig. 4. An illustration of the first step of model generation

To perform the model checking, each state is labelled with the atomic propositions that are true in the state. The verified goal formula $AF\ (AG\ (9 \leq L2 \leq 11) \wedge EG\ (25 \leq L1 \leq 30))$ (here, *f* is used as the abbreviation for $9 \leq L2 \leq 11$, and *g* for $25 \leq L1 \leq 30$) is written in terms of the basic connectives (i.e., $\neg EG (EF (\neg f) \wedge \neg EG (g))$). First, the set of states that satisfy the atomic formulae are calculated, followed by those for the more complicated sub-formulae. Let $S (\psi)$ denote the set of all states labelled with the sub-formula ψ.

(1) $S (\neg f) = \{1\}$

(2) $S (g) = \{2, 3, 4, 6, 7, 9, 10, 11, 12, 13, 14\}$

(3) $S (EF (\neg f)) = \{1\}$

In order to calculate $S (EG (g))$, first, the states of $S (g)$ are labelled with $EG (g)$ and then the label $EG (g)$ is deleted from any state if none of its successors is labelled with $EG (g)$. This deletion procedure is repeated until there is no change. Thus the calculation terminates with:

(4) $S (EG (g)) = \{2, 3, 4, 6, 7, 9, 10, 11, 12, 13, 14\}$

(5) $S (\neg EG (g)) = \{1, 5, 8\}$

(6) $S (EF (\neg f) \wedge \neg EG (g)) = \{1\}$

When computing $S (EG (EF (\neg f) \wedge \neg EG (g)))$, only state 1 is labelled with $EG (EF (\neg f) \wedge \neg EG (g))$ as was done in the last step (6). Clearly, state 1 has no successor labelled with $EG (EF (\neg f) \wedge \neg EG (g))$. So, the model verification continues:

(7) $S (EG (EF (\neg f) \wedge \neg EG (g))) = \{\}$

Finally, the converse of the transition relation is used to label all states in $S (EG (EF (\neg f) \wedge \neg EG (g)))$. Step (7) produces a result of $\emptyset$, thus implying that

(8) $S (\neg EG (EF (\neg f) \wedge \neg EG (g))) = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14\}$

This result is very strong because it means that the checked goal holds true along every path from any state in the execution structure. For example, since the initial state 1 is contained in this set, it can be concluded that, in this generated execution structure, the reconfigurable control started at that initial moment and is guaranteed to achieve the original goal $(AG (9 \leq L2 \leq 11) \wedge EG (25 \leq L1 \leq 30))$ at some later time.

VI. Conclusion

In this paper, a formal framework for reconfigurable control, based on model checking has been proposed. This framework first generates a flexible model (i.e., an execution structure) according to the diagnosis, then defines a temporal specification language to deal with the problems due to infinite execution cycles and non-determinism, and finally provides the algorithms that will automatically verify whether the updated model satisfies the desired specification. An example is given to illustrate the entire reconfiguration procedure. The results of this illustration show that our framework is able to express reconfiguration requirements very well and provides powerful qualitative computation capabilities.

In future research, it would be interesting to attempt to reduce the impact of the state explosion problem. There have been several noteworthy works [10], [11], [12] and [13] which could inspire us to deal with this problem, including attempts to exploit abstractions, symmetries and compositionalities. Another interesting work is about the fault detection phase. It is noted that this modelling methodology includes enough knowledge to build the diagnoser as the discrete event systems, since it includes the specification of the possible effects of an action. We think that it can be used as starting point for fault detection procedures in discrete event systems framework [14], [15], [16] and [17].

References

[1] R. J. Patton, "Fault – Tolerant Control Systems: the 1997 Situation," *IFAC Symposium on Fault Detection Supervision and Safety for Technical Processes*, Vol. 3, pp. 1033-1054, Kingston Upon Hull, UK, 26-28 August 1997.

[2] Z. Gao , P. J. Antsaklis, "Stability of the pseudo-inverse method for reconfigurable control systems," *Int. J. of Control*, Vol. 53, No.3, pp. 717-729, 1991.

[3] J. Lunze, T. Steffen, "Control reconfiguration by means of a virtual actuator," in *IFAC Safeprocess 2003*, Washington, USA, 2003.

[4] W. D. Morse, K. A. Ossman, "Model-following reconfigurable flight control system for the AFTI/F-16," *J. Guid., Con. & Dyn.*, Vol.13, No.6, pp. 969-976, 1990.

[5] Y. Ochi, K. Kanai, "Design of restructurable flight control systems using feedback linearization," *J. of Guid.,Contr. & Dyn.*, Vol. 14, No.5, pp. 903-911, 1991.

[6] H. X. Hu, A.-L. Gehin , M. Bayart, "Model Aggregation for Reconfigurable Control Based on Generic Component Model," in *ICSSSM'06*, Troyes, France, 2006.

[7] M. Staroswiecki M., A.-L. Gehin, "Analysis of System Reconfigurability using Generic Component Models," in *Control'98*, Swansea, UK, 1998.

[8] E. A. Emerson, "Temporal and Modal Logic," *Handbook of theoretical computer science (vol. B): formal models and semantics.* pp. 995 – 1072. The MIT Press, 1991.

[9] E. M. Clarke, J. O. Grumberg,, A. Doron , A. Peled, *Model Checking*. The MIT Press, 2000.

[10] D. E. Long, "Model Checking, Abstraction, and Compositional Verification," PhD thesis, School of Computer Science, Carnegie Mellon University, July 1983.

[11] D. R. Dams, "Abstract Interpretation and Partition Refinement for Model Checking," PhD thesis, Institute for Programming Research and Algorithmics. Eindhoven University of Techonology, July, 1996.

[12] E. M. Clarke, O. Grumberg, D.E. Long, "Model Checking and Abstraction," ACM *Transactions on Programming Languages and Systems*, Vol. 16, no. 5, pp. 1512-1542, September 1994.

[13] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, L. J. Hwang, "Symbolic Model Checking: 10^{20} States and Beyond," *Information and Computation (Special issue for the best papers from LICS'90)*, Vol. 98, no. 2, pp. 142-170, June, 1992.

[14] M. Sampath, R. Sengupta, S. Lafortune, K. Sinnamohideen , D. Teneketzis, "Diagnosability of discrete-event systems," *IEEE Trans. Automatic Control*, Vol. 40, No. 9, pp. 1555–1575, 1995.

[15] M. Sampath, R. Sengupta, S. Lafortune, K. Sinnamohideen, K., D. Teneketzis, "Failure diagnosis using discrete-event models," *IEEE Trans. Contr. Syst. Technol.*, Vol. 4, No. 2, pp. 105–124, 1996

[16] G. Lamperti, M. Zanella *Diagnosis of Active Systems*, Kluwer Academic Publis, 2003.

[17] S. Zad, H. Raymond, H. Kwong, W. M. Wonham, "Fault diagnosis in discrete-event systems: Framework and model reduction," *IEEE Trans. Automatic Control*, Vol. 48, No. 7, pp. 1199–1212, 2003.

[18] Richard E. Fikes and Nils J. Nilsson, "STRIPS: A new approach to the application of theorem proving to problem solving," *Artificial Intelligence*, Vol. 2, Issues 3-4, pp. 189-208, Winter, 1971.