Systematic Generation of Dependable Change Coordination Plans for Automated Switching of Coordination Policies

Prasanta Bose
George Mason University
Fairfax, VA
pbose@mgfairfax.rr.com

Mark G. Matthews
The MITRE Corporation
McLean, VA
mmatthew@mitre.org

Abstract

Distributed information systems for decision support and e-commerce applications require coordination of multiple autonomous components and their services to accomplish a set of global goals. In such systems, a global and often distributed coordination policy actively governs the coordination among the distributed components. The policies are used to interpret messages or events (e.g., job assignments, changes in data) from the cooperating components, in order to generate tasks with sequencing constraints. The generated tasks and their sequencing are then used as a basis for coordination of the services provided and required by the components. This paper presents the SWAP architecture that addresses the problem of dynamic changes in the needs or context of the autonomous components via change in the coordination policy at run-time. We present a method in the context of SWAP for the systematic generation of change coordination plans supporting the automated switching of coordination policies.

1. Introduction

In cooperating information systems, multiple distributed and autonomous components must be coordinated to accomplish a set of global goals. In such systems, a global and often distributed coordination policy actively governs the coordination among the distributed components. The policies may be domain independent or domain dependent. They are used to interpret messages or events (e.g., job assignments, changes in data) from the cooperating components, in order to generate tasks with sequencing constraints. The generated tasks and their sequencing are then used as a basis for coordination of the services provided and required by the components. Current architectures for such systems are based on design-time analysis and selection of the coordination policies in effect in a system.

A major problem with such static policy based coordination is handling dynamic changes in needs and context of the autonomous components that require dynamic changes in the coordination policy. For systems with non-trivial coordination policies, most current approaches require the system to be shut down, reconfigured, and then restarted in order to change the policy. Other approaches require the system to be at a quiescent state prior to the run-time coordination policy change [2].

In this paper we present a method for the systematic generation of change coordination plans supporting the automated switching of coordination policies. We present this method in the context of SWAP [3]; an architecture that supports run-time coordination policy changes for the class of cooperating information systems.

In section 2, we provide background information on the SWAP architecture. In section 3, we present a method for the systematic generation of dependable change coordination plans. In section 4, we include a brief example from the multi-source database view maintenance domain. In sections 5 and 6, we discuss related work and present a summary.

2. Background

In [3], we present the SWAP architecture for automated switching of coordination policies in response to dynamically changing needs and context. A simplified model of the structural view of the SWAP architecture is depicted in Figure 1 below.

The SWAP configuration in Figure 1 supports two coordination policies, P1 and P2. At any point in time, a single coordination policy is active and governs the interaction between the application components (e.g., C1-C3). The active policy is used to interpret messages or events from source components, in order to generate messages or events for consumer components satisfying specific sequencing and data constraints. In order to support run-time coordination policy changes, SWAP uses tracking agents (e.g., TA1 and TA2) and a change coordination agent (CCA).

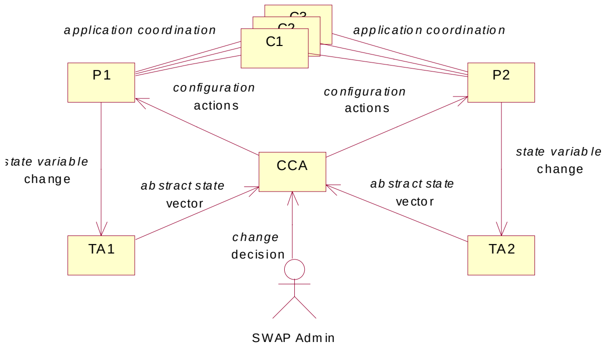

Figure 1. Simplified SWAP architecture

A major concern for mechanisms that support dynamic policy changes is ensuring safe and dependable transitions between coordination policies. This requires having knowledge of the processing state of the current coordination policy. The SWAP tracking agents monitor the coordination policy components and use specifications of the policies in order to track the policy execution state. In response to a switching decision from the SWAP administrator, the CCA uses the state information obtained from the tracking agents to coordinate the actions required to safely bring about the run-time change in a dependable manner. The CCA includes a set of change plans specifying when to switch and what initialization actions to take to bring about the switch.

For the methods to scale up as well as have change mechanisms that have soft real-time performance, SWAP relies on the use of abstract policy models for tracking as well as for generation of the change plans. The abstractions are generalization abstractions and are correctness preserving.

3. Systematic Generation of Dependable Change Coordination Plans

The focus of this paper is on the specification of safe and dependable change coordination plans, where a plan specifies the set of actions required to achieve a run-time switch from policy P_{FROM} to policy P_{TO} under a given context as captured by the tracking agents. This section presents a systematic method for generating change coordination plans. The method is built on the understanding that the coordination policy components maintain a set of key state variables that represent the application state and the tracking agents maintain abstractions of those key state variables. Furthermore, the key state variables represent queues associated with processing a job. Hence a necessary requirement for run-time change coordination between two polices is to transfer in-progress jobs from P_{FROM} to P_{TO}. However, simple transfer of state does not ensure that application specific safety and correctness properties are preserved during the change. Satisfying these properties requires searching for starting states in the target policy P_{TO} that meet these constraints.

Figure 2 presents a systematic method for generating a basic change plan which identifies the actions required to correctly initialize the state variables of policy P_{TO} based on the current values of the state variables of P_{FROM}. The method searches the state space (SA_{FR} and SA_{To}) of any two policies (P_{FROM} and P_{TO}) to generate a change plan ($CP_{FR\text{-}To}$) for handling all possible transitions from P_{FROM} to P_{TO}. The two primary steps of the method are 1) abstract event trace generation and 2) basic change plan generation.

```
GenerateBasicChangePlan(In:$SA_{FR}$,$SA_{To}$,Out:$CP_{FR-To}$)
$CP_{FR-To}$ = set of change plans
$TSA_{FR}$, $TSA_{To}$ = set of abstract traces
BEGIN:
1. GenerateAbstractEventTraces($SA_{FR}$)
2. GenerateAbstractEventTraces($SA_{TO}$)
3. Forall ts_fr $\in$ $TSA_{FR}$ where ts_fr.stable == true DO
4. Forall ts_to $\in$ $TSA_{TO}$ where ts_to.stable == true DO
5. Forall sv_fr $\in$ ts_fr DO
   5.1.Sv_c=ApplyCorrespondenceMapping(sv_fr);
   5.2. Forall sx $\in$ Sv_c DO
      5.2.1. If $\exists$ sv_to $\in$ ts_to such that subsumes(sv_to, sx) Then
CP=GenerateInitializationPlan(sv_fr, sv_to, $P_{FR}$, $P_{TO}$);
      5.2.2 Add (CP, $CP_{FR-TO}$); //without duplication
END;
```

Figure 2: Basic change plan generation method

3.1 Abstract Execution Trace Generation

In steps 1 and 2 of the GenerateBasicChangePlan method in Figure 2 we generate the search space for possible transitions between policies P_{FROM} and P_{TO} where the space is defined by the minimal set of abstract event traces of the policies. Figure 3 presents a systematic method for generating a "minimal set" of abstract event traces for the UML state machine specification [9] of a given coordination policy.

Step 1 of the method in Figure 3 generates the set of minimal "concrete" event traces. A concrete event trace consists of a sequence of state vectors for a given policy where the state vectors contain the real state variable data (i.e., not abstracted) maintained by the coordination policy components. The method for generating traces ensures minimality by generating execution traces of the state-machine specification such that no cyclic path is traversed more than once in any trace. The method ensures reachability by considering only those input events in the transitions that are plausible.

Step 2 of the method in Figure 3 applies the state abstraction mapping rules used by the tracking agents to abstractly track policy execution states. The mapping produces a set of abstract event traces TSA where an abstract event trace shows the set of abstract state vectors encountered by the tracking agents while tracking the processing of a particular workload scenario as captured by a corresponding concrete event trace.

```
GenerateAbstractEventTraces(In:SA, Out:TSA)
SA = state chart specification of source policy;
TS = set of traces of global state vectors for a policy;
TSA = set of traces of global abstract state vectors;
BEGIN:
1.  TS = GenerateAllMinimalRuns(SA) ;
2.  Forall  ti ∈ TS DO
    2.1 AbstractMap(ti, ta);
    2.2 Add(ta, TSA);
3.  Forall ta ∈ TSA  DO
    3.1 MarkStableStates(ta, ta' );
    3.2. Replace(ta, ta');
END;
```

Figure 3. Abstract event trace generation method

Step 3 of the method in Figure 3 evaluates the stability of each state vector in an abstract event trace. A state vector is considered unstable for run-time switching purposes if it is of a very short duration and hence, the state is likely to change before the change configuration agent has time to select a change plan and begin execution of the plan. We consider sequences of short duration states as atomic and do not support the dynamic switching of coordination policies during these atomic sequences.

3.2 Basic Change Plan Generation

In steps 3-5 of the GenerateBasicChangePlan method in Figure 2, we exhaustively search the abstract state space (TSA_{FR} and TSA_{TO}) of policies P_{FROM} and P_{TO} to generate a basic change plan CP_{FR-TO} for handling all possible transitions from P_{FROM} to P_{TO}.

For a given stable starting state sv_fr (specifying when to transition) in ts_fr, the inner loop (step 5 in Figure 2) finds those stable states sv_to (specifying where to transition) in ts_to that match the workload constraints in the source state. The matching operation involves: 1) Mapping the source state sv_fr to a corresponding state sx in the target policy based on a set of correspondence rules for mapping the abstract state variables of policy P_{FROM} to the abstract state variables of P_{TO}. 2) Checking for subsumption of sx by sv_to. The condition part of a basic change plan (a reactive rule) is then sv_fr and the action part is a set of configuration actions that initialize the state variables of the policy P_{TO} to conform to the state vector sv_to.

The method in Figure 2 deals solely with the configuration of the state variables of the coordination policies. The change plans generated by the method must be augmented with additional actions to ensure the preservation of application-specific safety and correctness properties during the switch.

4. Example

To illustrate, we consider an example in which the application components in Figure 1 are a set of autonomous databases in which a materialized view is incrementally maintained at one database from source relations contained within the other databases. In such an example, an incremental view maintenance policy is required to maintain consistency between the source data and the client view. We consider the case in which the policies P1 and P2 in Figure 1 are the Strobe and C-Strobe incremental view maintenance policies [10]. Assuming that we are given UML state machine [9] specifications for the Strobe and C-Strobe policy and tracking agents, the problem then is to execute the method in Figure 2 to generate a set of change coordination plans. The UML state machines for the C-Strobe policy and tracking agents are depicted in Figures 4 and 5.

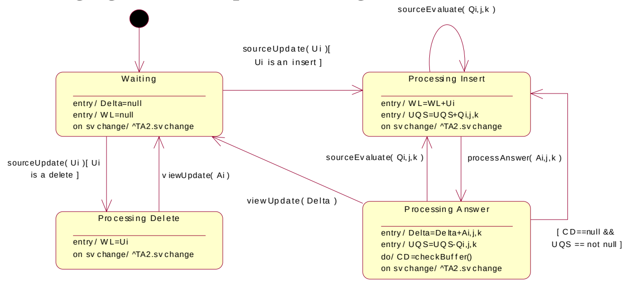

Figure 4. C-Strobe policy specification

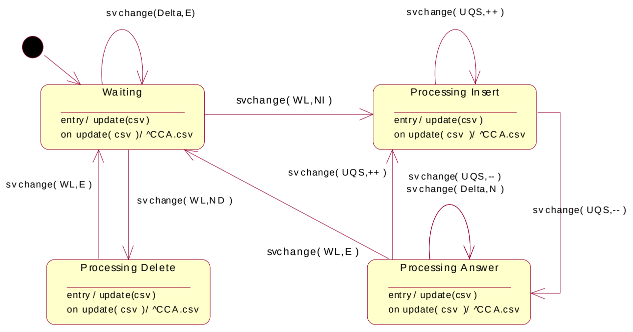

Figure 5. C-Strobe tracking agent specification

The GenerateAbstractEventTraces() method produces four event traces through the C-Strobe state machine. An example event trace is included in Table 1 and the corresponding abstract event trace is included in Table 2. Table 3 shows the output of the

GenerateBasicChangePlan() method for a switch from C-Strobe to Strobe. The plan shows the actions required to initialize the state variables such that state information pertaining to in-progress jobs is preserved during the switch.

Table 1. C-Strobe event trace (minimal run)

ID	Event Sequence	IB	WL	UQS	Delta	State
0	initial state (idle)	null	null	null	null	W
1	ACA.sourceUpdate(U1,insert,r1)	U1	null	null	null	W
2	CSA.sourceUpdate(U1,insert,r1)	null	null	null	null	T
3	WL=WL+U1	null	U1	null	null	PI (e)
4	Q1,1,0=V(U1)	null	U1	null	null	PI
5	A1,1,0=QPA.sourceEvaluate(Q1,1,0)	null	U1	null	null	T
6	UQS=UQS+Q1,1,0	null	U1	Q1,1,0	null	PI (e)
7	CSA.processAnswer(A1,1,0)	null	U1	Q1,1,0	null	T
8	UQS=UQS-Q1,1,0	null	U1	null	A1,1,0	PA (e)
9	Delta=Delta+A1,1,0	null	U1	Q1,1,0	A1,1,0	PA (e)
10	CD=checkBuffer()	null	U1	null	A1,1,0	PA (e)
11	CI=checkBuffer()	null	U1	null	A1,1,0	PA
12	viewUpdate(Delta)	null	U1	null	A1,1,0	PA
13	CMA.viewUpdate(Delta)	null	U1	null	A1,1,0	T
14	WL=null	null	null	null	A1,1,0	W (e)
15	Delta=null -- final state (idle)	null	null	null	null	W (e)

Table 2. Abstract C-Strobe event trace

ID	Event Sequence	WLa	UQSa in/out	DELTAa	State	stable
0	initial state (idle)	E	0/E	E	W	yes
1	ACA.sourceUpdate(U1,insert,r1)	E	0/E	E	W	yes
2	CSA.sourceUpdate(U1,insert,r1)	E	0/E	E	W	yes
3	WL=WL+U1	NI	0/E	E	T	T
4	Q1,1,0=V(U1)	NI	0/E	E	PI	no
5	A1,1,0=QPA.sourceEvaluate(Q1,1,0)	NI	0/E	E	PI	no
6	UQS=UQS+Q1,1,0	NI	1/N	E	T	T
7	CSA.processAnswer(A1,1,0)	NI	1/N	E	PI	yes
8	UQS=UQS-Q1,1,0	NI	0/E	E	T	T
9	Delta=Delta+A1,1,0	NI	0/E	N	PA	no
10	CD=checkBuffer()	NI	0/E	N	PA	no
11	CI=checkBuffer()	NI	0/E	N	PA	no
12	viewUpdate(Delta)	NI	0/E	N	PA	no
13	CMA.viewUpdate(Delta)	NI	0/E	N	PA	no
14	WL=null	E	0/E	N	T	T
15	Delta=null -- final state (idle)	E	0/E	E	W	T

Table 3. Basic change plan (C-Strobe to Strobe)

PFR	sv_fr	PTO	sv_to	initialization plan
C-Strobe	(E,E,E,W)	Strobe	(E,E,E,W)	none
C-Strobe	(NI,N,E,PI)	Strobe	(N,NFP,E,EV)	SSA.WL=CSA.WL, SSA.UQS=CSA.UQS, CSA.WL=null,CSA.UQS=null
C-Strobe	(NI,N,N,PI)	Strobe	(N,NFD,N,EV)	SSA.WL=CSA.WL, SSA.UQS=CSA.Q[WL], SSA.AL=processAnswer(Delta[1]), CSA.WL=null, CSA.UQS=null,CSA.Delta=null
C-Strobe	(NI,N,N,PA)	Strobe	(N,NFD,N,EV)	same as above

5. Related Work

The SWAP work on change coordination is related to the recent work in the area of specifying and analyzing dynamic software architectures [1, 5] and on using architectural specifications to plan and analyze changes in the run-time system [4, 7]. An approach to run-time software evolution based on exploiting an explicit architectural model of the system is presented in [6]. The SWAP change coordination method presented in this paper exploits domain specific knowledge to identify the necessary components and their role in the change coordination process. The SWAP work also exploits existing UML [8, 9] standards for object modeling to do architecture modeling.

6. Summary

The focus of this paper is on the specification of safe and dependable change coordination plans, where a plan specifies the set of actions required to achieve a run-time switch between two coordination policies under a given context. We present a systematic method for generating change coordination plans for the change coordination agent within the SWAP architecture. The method identifies the set of actions required to transfer state information pertaining to in-progress jobs from the current coordination policy P_{FROM} to another coordination policy P_{TO}. Future work is focused on developing an automated tool kit to support the generation of domain-specific instance architectures conforming to the generic SWAP architecture.

7. References

[1] R. J. Allen, R. Douence, and D. Garlan, “Specifying and Analyzing Dynamic Software Architectures,” Proceedings of the 1998 Conference on Fundamental Approaches to Software Engineering (FASE '98), March 1998.

[2] P. Bose and M. G. Matthews, “Coordination of View Maintenance Policy Adaptation Decisions: A Negotiation Based Reasoning Approach,” Proceedings of the International Workshop on Self-Adaptive Software (IWSAS 2000), April 17-19, 2000.

[3] P. Bose and M. G. Matthews, “An Architecture for Achieving Dynamic Change in Coordination Policies,” Proceedings of the 4th International Software Architecture Workshop (ISAW4), June 2000.

[4] D. Garlan and M. Shaw, “Software Architectures: Perspectives on an Emerging Discipline,” Addison Wesley Publishers, 1996.

[5] J. Magee, and J. Kramer, “Dynamic Structure in Software Architectures,” Fourth SIGSOFT Symposium on the Foundations of Software Engineering, San Francisco, October 1996.

[6] P. Oreizy, N. Medvidovic, and R. N. Taylor, “Architecture-based Runtime Evolution,” Proceedings of the International Conference on Software Engineering (ICSE), 1998.

[7] P. Oreizy et. al., “An Architecture Based Approach to Self-Adaptive Software,” IEEE Intelligent Systems, 1999.

[8] J. Robbins, N. Medvidovic, D. Redmiles, and D. Rosenbloom, “Integrating Architecture Description Languages with a Standard Design Method,” Second EDCS Cross Cluster Meeting, Austin, Texas, 1998.

[9] J. Rumbaugh, I. Jacobsen, and G. Booch, “The Unified Modeling Language Reference Manual,” Addison Wesley, 1999.

[10] Y. Zhuge, H. Garcia-Molina, and J. Wiener, “The Strobe Algorithms for Multi-Source Warehouse Consistency,” Proceedings of the International Conference on Parallel and Distributed Information Systems, December 1996.