Automation Controller for Operational IT Management

Sven Graupner
Enterprise Systems and Software Laboratory
HP Laboratories Palo Alto
HPL-2007-144
August 23, 2007*

IT Management, automation, workflow engine, operational management, ITSM, ITIL

Controllers in computer systems have mainly been explored for automating regulative tasks such as admission control or resource supply control. The majority of IT management tasks, however, relies on discrete management states and coordinated transitions between those states.

The paper shows how the concept of a feedback system can also be applied to automate operational management tasks. The paper introduces the concept and a realization of an IT Management Automation Controller, which operates on discrete management states expressed as a pair of models for desired and observed state. Models are represented as a special form of Place-Transition Nets (PTN or Petri Nets). Controller logic directly executes PTN in order to achieve and maintain alignment between desired and observed state in a managed domain. In contrast to workflow systems, PTN combine the description of state and actions in one model (graph).

Three operational database management tasks have been implemented as a proof of concept in a blade server automation infrastructure using the Management Automation Controller.

* Internal Accession Date Only
Published in IM 2007, 21-25 May 2007, Munich, Germany

Approved for External Publication

© Copyright 2007 IEEE

Automation Controller for Operational IT Management

Sven Graupner
HP Labs, 1501 Page Mill Rd, Palo Alto, CA 94304, USA
sven.graupner@hp.com

***Abstract* — Controllers in computer systems have mainly been explored for automating regulative tasks such as admission control or resource supply control. The majority of IT management tasks, however, relies on discrete management states and coordinated transitions between those states.**

The paper shows how the concept of a feedback system can also be applied to automate operational management tasks. The paper introduces the concept and a realization of an IT Management Automation Controller, which operates on discrete management states expressed as a pair of models for desired and observed state. Models are represented as a special form of Place-Transition Nets (PTN or Petri Nets). Controller logic directly executes PTN in order to achieve and maintain alignment between desired and observed state in a managed domain. In contrast to workflow systems, PTN combine the description of state and actions in one model (graph).

Three operational database management tasks have been implemented as a proof of concept in a blade server automation infrastructure using the Management Automation Controller.

Keywords: IT management, automation, controller.

I. Introduction

Enterprises streamline their IT environments by unifying, consolidating and centralizing IT processes, platforms and organizations in order to improve efficiency. Reducing the diversity in IT not only simplifies management, it also supports the introduction of systematic processes such as ITSM [1] and automation by creating more elements and processes of the same kind. Unification and consolidation reduces cost and leads to better aligned IT processes and organizations.

II. Automation in IT Management

The reality of IT management is dominated by a legacy of management systems which have been designed as tools for human operators, not for automation and self-management. Today, IT management can be seen as at a stage of mechanization where operators use management systems as tools to carry out management tasks. Management systems are designed as tools for operators and facilitate operations through consoles. Some tools, and lately more and more tools, also provide API to allow programmatic control and scripting for integration in process automation chains.

The 1st Stage of Automation: Scripts and Workflows. Automation at this stage is characterized by management tools that can be accessed through API or command lines enabling scripts and workflows to describe action sequences of repeatable management tasks. The operator initiates the script or workflow execution as opposed to actions individually.

The 2nd Stage of Automation: ECA Policies. Initiation of action sequences can be triggered by conditions reported as events from the managed system. Definitions of Event-Condition-Action ECA triples are also often referred to as ECA polices. As events are reported from the managed environment, they pass through a sequence of conditions, and for each condition evaluating to true, the associated action sequence is executed. ECA policies are widely used IT management.

First and second stage automation have no knowledge about the changes executions cause in the managed environment. An action sequence runs once when initiated. There is no inherent ability to detect whether the goal which caused the execution actually has been achieved in the managed environment or not.

To some extent, conditions in ECA policies can be seen as representations of a desired state such as thresholds that should not be passed. However, an ECA system relies on external events to trigger evaluation for executing actions, which is the difference to a controller that autonomously evaluates conditions and triggers actions in order to maintain a managed environment aligned to its desired state.

The 3rd Stage of Automation: Controllers. A controller has a description (model) about a desired status of its controlled domain. It also has a reflection (model) of the current status that is observed from the controlled domain. Both models, which are called the *Desired State Model (DSM)* and the *Observed State Model (OSM)*, are constantly evaluated by the controller. Corrective actions are deducted and executed as differences occur between the two models.

Intended change in the controlled domain is achieved by changing the desired state model, either manually by an operator or programmatically by another system or controller. Unintended change can occur any time the system that is reflected back into the observed state model such as in case of a failure. Both kinds of changes may initiate actions in the controller in order to maintain alignment between observed and desired state. Controllers may not be able to achieve alignment under all conditions. Those cases need to be detected and reported to a superior instance as uncorrectable conditions.

III. Controllers in IT Management

In general, a controller adjusts conditions of a controlled (or managed) element or domain by altering control knobs as a function of measured parameters and controller settings. Measured parameters can be interpreted as a form of observed

state. Controller settings can be seen as a form of desired state. Controller settings or desired state represents a goal according to which the controller aligns the controlled element by adjusting its control knobs according to the result of the evaluation of the control function. A large body of literature exists on feedback control in computing systems [2].

Controllers have been introduced to computer systems in a variety of ways:

Regulative Controllers operate based on numeric input for measurements and settings to their control functions, which produces numeric output for control knobs in the controlled environment. Examples are admission controllers, which throttle incoming workload when it surpasses processing capacity [3], or flex controllers, which expand or shrink resource supply based on workload [4]. Controller settings can only be altered from outside, not by the controller itself.

Adaptive Controllers can alter (tune) the control settings or even the control function as result of reasoning upon observed behavior in the past and deriving predictions for the future [5][6]. Adaptive admission controllers have been shown in [7]. Adaptive flex controllers have been presented in [8].

Autonomic Manager is a basic concept of Autonomic Computing [9]. It defines a closed loop with stages: monitor, analyze, plan, execute for a managed element or domain. A number of controllers have been implemented based on this concept, mainly regulative and adaptive controllers [10].

IT Management Automation Controller which is discussed in this paper is similar to a regulative controller. In contrast, it is not based on numeric control parameters, settings and function, it uses two discrete state models associated with a managed domain, the desired state model and the observed state model. The control function represents discrete state logic that evaluates the two models and produces a sequence of actions based on differences. An IT Management Automation Controller also meets the general criteria of the Autonomic Manager concept. It is a specific form directed to IT task automation and composition of automated IT management process chains.

IV. IT MANAGEMENT AUTOMATION CONTROLLER

HP has developed a concept of an IT Management Automation Controller [11], which is shown in Figure 1. with following components:

- DSM Interface through which the desired state model is accessed (read/write);
- OSM Interface through which the observed state model is accessed (read only, subscriptions to change events);
- Controller function (logic) which consists of:
    - Differencer logic which compares the desired and the observed state model;
    - Action sequencer logic which derives actions from the difference;
- Observer Connector through which the observed state model is updated from the managed environment (polled by the controller and/or event-based from the environment);
- Actuator Connector through which actions are passed into the managed environment for execution.

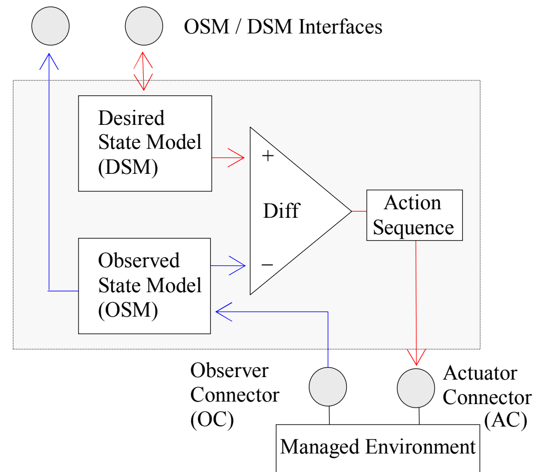

Figure 1. IT Management Automation Controller.

Control information flows into the controller in form of desired state definitions and changes to those definitions through the DSM interface. It also flows into the controller as changes to the observed state model through the observed connector. The controller's internal control loop aims to maintain the alignment between observed and desired state by deriving and issuing actions through its actuator connector. If this cannot be achieved, superior instances can subscribe to event types at the observed state model interface to be notified when those conditions occur.

A. Controller Implementation Toolkit

A toolkit [12] was developed for implementing controllers based on web services management standards. This toolkit was used for building the controllers for the database use cases described later. Controller interfaces employ web service management standards [13]. Web services management standards WSRF [14] and WSDM [15] were initially employed using the open source WSRF implementation from Globus GT4 [16] as basis for the toolkit supplemented with WSDM schema. The controller toolkit is currently being refactored to support the more recent WS-Management [17] standard. All web services management standards provide similar operations to access XML representations of models as well as event notifications.

B. Model Representation

Web services management standards achieve interoperability at the interface level. In regard to models, they only require that models are or can be rendered in XML. They do not impose a specific modeling framework. Interoperability at the model exchange level requires additional agreement. In the current controller realization, models are defined as XML schema and are proprietary. Compliance with the recently emerging modeling framework Service Modeling Language (SML) [18] is desirable and will be factored into controller models as its common and core model definitions mature.

V. PROBLEM STATEMENT

The modeling framework determines the expressiveness of models. It also determines the mechanisms that are required for interpretation (the controller logic in this case).

Using sole declarative models entirely hides the logic for interpretation inside the controller. A declarative model only describes a desired or observed state (data). It does neither describe how this state should be interpreted, nor how it came to this state and what should happen in that state. Current model-driven approaches to IT management favor the use of declarative models [19].

While this seems desirable at a fist glance, it has a number of shortcomings:

- Logic is built into controllers, typically hard-coded and cannot be customized.
- Logic is not modeled, hence remains unclear and hard to trace.
- Since logic depends on the structure and semantics of models, changes to those will likely break the controller logic requiring code replacement in all controller instances.
- Controllers depend on interactions with the managed environment. Sole declarative models do not provide means to describe those interactions and dependencies.
- Controller composition and automated IT management process chains require coordination among controllers. Again, when logic is built into the code of the controller, it cannot be customized making automated IT management process chains difficult to build and maintain.

Using declarative models in combination with built-in controller logic may be desirable for lower-level resource controllers with a fixed behavior. It is not sufficient for higher-ordered controllers that operate at a level of automated IT management process chains that require customization and adaptation in a customer environment.

A. Alternatives

To overcome the problem of separation of the model from its interpretation (logic), models can be supplemented with the interpretation logic to avoid hard-coded logic in controllers. Some modeling frameworks support the representation of interpretation rules. An example is the Resource Description Framework (RDF). A rules engine like the Jena Rules engine could execute the model interpretation rules that are part of the model. However, dynamic behavior is hard to integrate in rule engines.

Workflows are typically used to describe configurable, dynamic behavior across systems and execute on it. However, it is difficult to represent state in workflows, such as observed or desired state of a managed element. Once again, it leads to the separation between "state models" and "execution logic", although the execution logic is now configurable in a workflow engine and not built into the controller.

Another aspect with workflow languages such as BPEL is that they are designed for business transactions, which may suit higher-level, more transactional IT management processes, but is not a good match for the asynchronous and partially unpredictable behavior that occurs in a dynamic management environment such as asynchronous events, race conditions or critical sections.

A balance needs to be found that brings all those aspects together: the representation of desired and observed management states, the description of dynamic interactions and dependencies with other controllers, and the representation of interpretation logic in a form which allows to execute on a generic engine as opposed to built-in code in controllers.

VI. APPROACH: PTN

Place-Transition Nets provide a good approximation:

- State can be represented in a Place-Transition Net,
- Interactions with the managed environment leading to state changes can be expressed,
- Dynamic behavior, coordination and synchronization with other controllers can be represented,
- Synchronous and asynchronous interactions can be modeled,
- Place-Transition Nets can be executed (interpreted) by a generic execution engine.

Place-Transition Nets are well proven in domains such as manufacturing and supply chains. They have also been used in telecommunications for modeling and verifying protocols. A large body of experience and knowledge exists, from formal techniques for proving liveliness or reachability to simulation environments. However, despite favorable properties, PTN have not widely been leveraged in IT management and automation, which may partially be due to a lack of tooling and experience with PTN in the domain of IT management.

The realization of the IT management automation controller presented in this paper applies Place-Transition Nets as the modeling framework for representing the models for desired and observed states and for executing controller logic by interpreting PTN. A generic Place-Transition Net execution engine was built and included in controllers replacing their hard-coded logic. All interactions with the managed environment as well as with other controllers are driven by interpreting PTN.

A. Place-Transition Nets

Place-Transition Nets (PTN) or Petri Nets were first introduced in [20]. A Petri Net is defined as a 6-tuple (S, T, F, M_0, W, K) where S is a set of places and T is a set of transitions. F is a set of arcs between either a place and a transition or a transition and a place: $F \subseteq (S \times T) \cup (T \times S)$. A token is a construct that represents state in a place.

A distribution of tokens over the places in a net is called a marking. M_0 is the initial marking, $M_0: S \rightarrow N$ with each place $s \in S$ having $n \in N$ initial tokens. $W : F \rightarrow N$ is a set of arc weights $n \in N$ assigned to each arc $f \in F$ denoting how many tokens are consumed from a place by a transition and how many tokens are produced by a transition and added to a subsequent place. $K : S \rightarrow N$ is a set of capacity restrictions which assigns to each place $s \in S$ some positive number $n \in N$ denoting the maximum number of tokens that can occupy that

place. A net in which each of its places has some capacity k is known as a k-bounded Petri Net.

Places may contain any number of tokens up to the capacity restriction $k \in K$. A marking is altered $m_i \rightarrow t_{i,j} \rightarrow m_j$, $m_i \in M$, $m_j \in M$ when transition $t_{i,j} \in T$ fires. Firing a transition is an atomic operation.

Transitions *may* fire, when they are enabled. Transitions are enabled when they have at least the amount of tokens in each input place specified by the inbound weight of the transition (default is 1). When a transition fires, it consumes the weight amount of tokens from each inbound place and adds the amount of tokens specified by the outbound weight to each outbound place (default is 1).

These fundamental properties of Petri Nets allow reasoning on properties such as reachability, liveness or boundedness.

Execution of Petri Nets is nondeterministic. Multiple transitions can be enabled at the same time, any one of which can fire in any order or simultaneously. Transitions may not fire immediately when they become enabled or may not fire at all. Since firing is non-deterministic, Petri Nets are suited for modeling asynchronous and concurrent behavior of distributed systems [21].

However, some assumptions must be made in regard to non-determinism for the practical use of PTN for management automation. Furthermore, a combination of three extensions Colored Petri Nets (CP-Nets), Hierarchical Petri Nets and Timed Petri Nets is used.

B. Colored Petri Nets (CP-Nets)

In a basic Petri Net, tokens are indistinguishable ("black") and themselves stateless. Only their assignment to a place at a time determines the state (marking) in the network.

A number of examples in the domains of network protocols and manufacturing supply chains are shown in [22] where distinguishable items travel through a network as tokens following the PTN rules. Those items (represented as tokens) must carry own state ("color") in order to be distinguishable. Prof. Kurt Jensen from the University of Aarhus has developed Colored Petri Nets [23] by introducing following extensions:

- assign state (a value) to tokens that is defined by a simple or complex type,
- assign a type to places determining the type of tokens it can hold,
- allow multi-sets of tokens of same type and value by specifying coefficients, and to
- assign expressions (functions) to arcs that can be bound to token values and evaluated when tokens pass through transitions during firing.

Transitions in a CP-Net thus do not only alter the marking of the overall net and bring tokens to other places. Evaluation of arc expressions also allows altering the state within tokens when they pass through a transition. Those functions can alter token state.

Tokens do not share their states. States of multiple tokens can be combined as result of evaluating arc expressions when they are part of the same transaction and hence part of the same evaluation process. Altering states in tokens by evaluating arc expressions allows "programming" in a CP-Net. Tools have been developed for CP-Nets that are widely used, such as CPNTools [24].

C. Hierarchical Petri Nets

The idea behind hierarchical Petri Nets is to introduce scope, reusable building blocks and a modular structure in larger nets. Each place can be expanded into a (sub-) net into which tokens flow via inbound transitions, internally travel through the subnet and finally return or produce tokens in the surrounding net. Nets and places within nets can be made self-similar such that they can be composed hierarchically [25].

Inbound and outbound arcs to a (subnet-) place also define the interface to an underlying net. The structure of this net can remain hidden as long as the interface is known. Properties of Hierarchical Petri Nets and examples of reusable subnets (such as for critical sections or the reader-writer problem) are discussed in [26].

D. Timed Petri Nets

Petri Nets are non-deterministic in terms of when enabled transitions fire or if they fire at all. Again for practical reasons, Timed Petri Nets allow to define an interval within which an enabled transition must fire. The lower bound of the interval defines the minimal and the upper bound the maximal time an enabled transition must or can wait to fire.

E. Combination of Colored, Hierarchical, Timed Petri Nets

A combination of the three Petri Net extensions has been chosen as foundation for the PTN used in the Management Automation Controller. In addition, following assumptions are made, which are explained later in the text:

- Two types of places are introduced: regular places and connector places.
- Two types of tokens are introduced: regular and activity tokens.
- Two special transition rules are introduced called bonding and detaching.
- The ambiguity of a conflict ("confusion") in a PTN is resolved by labeling outbound arcs from places with disjunctive values and introducing a choice field as part of a tokens data type. In case of a conflict, the outbound arc with a matching choice label determines the next enabled transition.
- CP-Net multi-sets are not allowed.
- The default firing interval for transitions is [min=0, max=0], which means that transitions fire immediately as soon as they become enabled. The firing order of multiple simultaneously enabled transitions is arbitrary (undefined). The firing interval can be redefined for transitions.

F. Workflow Patterns with Petri Nets

Figure 2. shows common workflow patterns in terms of PTN. Case (a) shows a simple sequence. Since the default weight of arcs is 1, the token in place s_1 enables transition $t_{1,2}$. Firing $t_{1,2}$ brings the token to place s_2 by reducing the number of tokens in s_1 by 1 and increasing it by 1 in s_2. Case (b) is similar, except that 1 token is added to both places s_2 and s_3.

This means that the one token from s_1 becomes duplicated in places s_2 and s_3. Both tokens in s_2 and s_3 are independent, which semantically corresponds to forking a process.

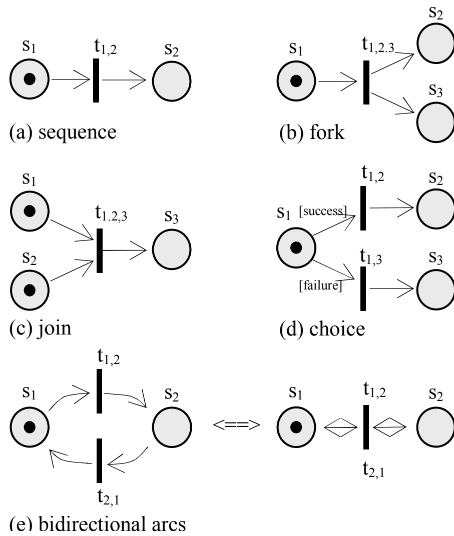

Figure 2. Basic workflow patterns as Petri Nets.

In case (c), transition $t_{1,2,3}$ is only enabled when both inbound places have at least one token each. Following the normal transition rule, 1 token is removed from each inbound place s_1 and s_2 and 1 token is added to s_3. This means, two tokens from s_2 and s_3 join at this transition. Two independently traveling tokens are synchronized.

The literature refers to case (d) as conflict or as "confusion" because both transitions $t_{1,2}$ and $t_{1,3}$ are enabled. Only one transition can fire since the one enabling token cannot be reduced twice by two firing transitions. Classic Petri Nets define this case as non-deterministic choice for selecting the firing transaction. One common approach to turn this case into a deterministic choice is to label outbound arcs (such as with "success" or "failure" in the figure) and determine the firing transition by computing a result against which the labels are compared. This results into the known branching pattern.

Case (e) shows a convention which is often used in Petri Nets to abbreviate bidirectional arcs. Both notations are semantically equivalent. Note that the bidirectional transition actually represents two transitions.

VII. Interpretation for the IT Management Automation Controller

In context of the Management Automation Controller, two main domains are modeled as PTN: one is the model of Desired State (DS) and one is the model of Observed State (OS) for a managed environment.

A *place* represents a desired or observed state in the managed environment. Examples of such states are: [system is down], [server is down], [application is running], or [maintenance is in progress]. States of a typical lifecycle diagrams correspond to places in a PTN. (A notation is used in the following for describing [states] and <transitions>).

A *transition* represents a change between those states. When the prior state was [server is down], and the subsequent state in the model is [server is up], then the transitions between the two states is <boot>. Since booting of a server is a longer term operation, it by itself can be modeled as a state: [server is booting]. It is good practice to model rather "short" or "timeless" indications as transitions such as <ignite server boot>, followed by the [server is booting] state (place), followed by a transition <bootstrap finished successfully> before entering the [server is up] state.

Figure 3. shows the expansion of a regular PTN transition into a transition state s_2, which is a regular PTN place. The transition $[s_{1=\mathrm{DOWN}}] \rightarrow \langle t_{1,2=\mathrm{BOOT}} \rangle \rightarrow [s_{2=\mathrm{UP}}]$ expands to $[s_{1=\mathrm{DOWN}}] \rightarrow \langle t_{1,2=\mathrm{IGNITE\ BOOT}} \rangle \rightarrow [s_{2=\mathrm{BOOTING}}] \rightarrow \langle t_{2,3=\mathrm{BOOT\ FINISHED}} \rangle \rightarrow [s_{3=\mathrm{UP}}]$.

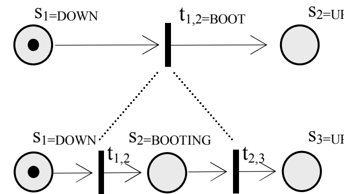

Figure 3. Expansion of a transition into a "transition state".

A *token* represents a managed element. The place in which a token resides defines the (either desired or observed) status of the associated managed element. Since tokens can carry own state, multiple managed entities (e.g. multiple servers) can be represented and transition in the same PTN, each independent from the others. Each managed element is represented by a pair of tokens, one representing its desired and one representing its observed state in the two PTN models, respectively.

The equivalent of a token in the workflow language BPEL would be a BPEL message. However, messages in workflow languages are meant to be received and processed according to the workflow definition.

A. Representing Desired and Observed State Models as PTN

Figure 4. shows a simple lifecycle model for class of servers as pair of PTN for desired and observed states. A pair of tokens (*md, mo*) represents the managed element, which is a particular server instance. Position of token *md* in a place in the desired state model represents the desired status of the managed element. Position of token *mo* in a place in the observed state model represents the observed status of the element in the managed environment.

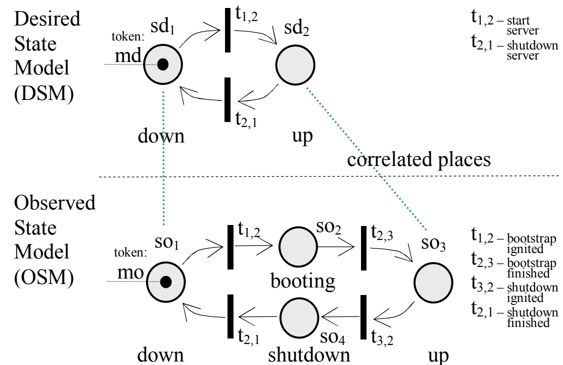

Figure 4. Simple lifecycle model for a server expressed as Petri Nets

(The transitions in Figure 4. are not yet controlled.)

A token *md* in place sd_1 in the desired state model (as shown in the figure) means that the server is supposed to be up. The token in sd_2 would mean that the server is desired to be down. A token *mo* in place so_1 in the observed state model indicates that the server is observed as down (as shown in the figure). A token in so_2 would mean the server would be in the process of booting. The token in so_3 would mean that the server would be observed as up, and the token in so_4 would indicate the server is shutting down.

There are correlations between certain places in the desired and the observed state models such as sd_1 and so_1 [down] and sd_2 and so_3 [up]. Those correlated places represent alignment between desired and observed state when the two tokens (*md, mo*) reside in those correlated places. Correlated places represent the same management status, either as desired or observed. Correlated places are defined as one-to-one relationships between pairs of places from the desired state model: $S_{DS} = \{ sd_i \}, i=1...n$ and from the observed state model $S_{OS} = \{ so_j \}, j=1...m, j \geq i$.

A set of correlated places C is defined as set $C = \{ (sd_i, so_j) \}$ with $\forall\, sd_i \in S_{DS} : sd_i \rightarrow so_j$ and $so_j \in S_{OS} : sd_j \rightarrow so_i$. For each place in a desired state model, there must be a correlated place in the observed state model. There may be more places in the observed state model that represent intermediate stages of a managed element (such as booting). The correlated places for the example in Figure 4. are: $C = \{ (sd_1, so_1), (sd_2, so_3) \}$.

A managed element is *aligned* when its pair of tokens (*md, mo*) is residing in a pair of correlated places. The subset of markings that represent alignment between desired and observed states is $M_{ALIGN} : md \rightarrow sd_i, mo \rightarrow so_j$ with $(sd_i, so_j) \in C$. Any other marking represents non-alignment between desired and observed state for the managed element.

B. Deriving Actions from Desired State Changes

A state of alignment can only change when either new desired state is defined or when a change occurs in the managed environment that reflects back to a change in observed state. Both changes lead to non-alignment since at least one token must transition from the correlated pair of places to another place, and one place can only be in one correlation according to the definition of C.

In case desired state is changed (intended change), the controller must determine a sequence of actions that brings the managed environment into new alignment. In the example, when desired state of a server is changed from [down] to [up], the boot process must be ignited and completed before the observed state can indicate that the server is up.

Aligned state: $c_{1=DOWN} = (sd_1, so_1)$.

1.) $md \rightarrow sd_2$ with $t_{1,2}$ firing in DSM leading to:

2.) $mo \rightarrow so_2$ with $t_{1,2}$ firing in OSM (igniting the server boot and booting the server),

3.) $mo \rightarrow so_3$ with $t_{2,3}$ firing in OSM (server boot completed).

Aligned state: $c_{2=UP} = (sd_2, so_3)$.

Server shutdown follows the same pattern. It will later be shown how error conditions can be taken into account and eventual corrective actions can be derived from error states. Errors and failures are examples of unintended changes that may occur in the observed state model.

All activity of the controller depends on firing transitions in DSM and/or OSM. In order to control firing, DSM and OSM need to be extended by connector places and activity tokens.

C. Connector Places

Connector places supplement DSM and OSM to provide the interfaces with the environment. Tokens can be generated or consumed in connector places as effects of interactions with the environment. Those interactions occur with the managed environment, with a user interacting through a console or with other controllers. Connector places may be source connector places or terminal connector places. The union of source and terminal connector places represents the *interface* of the PTN.

A source connector is a place that interacts with the environment and creates new activity tokens as effect of this interaction. A source connector has only outbound arcs. A terminal connector is a place that interacts with the environment when it receives an activity token and initiates an action. Activity tokens may be consumed during this interaction. Connector places may exist that are neither source nor terminal to represent intermediate stages. Connector places supplement PTN for DSM and OSM allowing so-called activity tokens to travel.

D. Activity Tokens

While tokens so far have been introduced to represent managed elements, activity tokens represent actions or state changes associated with managed elements. Activity tokens are the only cause of transitions for regular tokens in a DSM or OSM. Activity tokens are used to initiate and control the transitions in DSM and OSM nets. Activity tokens are associated with a specific managed element (and the representing token) to which the interaction applies.

While regular tokens (representing managed elements) can only travel through regular (non-connector) places, activity tokens may travel both under two special transition rules:

- *Join transition (bonding):* an activity token enables a join transition only for the associated (managed element) token and bonds with it during the transition. It remains bonded until it is detached from the regular token.
- *Fork transition (detaching):* if a bonded token arrives at a fork transition where following places include both connector places and non-connector places, activity tokens detach from carrying regular tokens and pass along the arc(s) to the connector place(s), while the regular token passes along the other arc(s) to non-connector places.

E. Deriving Actions from Desired State Changes

Figure 5. shows the desired state model from Figure 4. supplemented with connector places $sc_{[1,2,3,4]}$ in a state before and after transition $t_{1,2}$ has fired.

An activity token is shown in a source connector place sc_1 before $t_{1,2}$ has fired. This activity token might have been created as effect of a user interaction to change the desired state from [down] to [up]. The occurrence of the activity token in sc_1 enables and fires transition $t_{1,2}$ (under the assumption that the

activity is associated with the managed element represented by the token in sd_1).

(Non-connector places are shown with grey shading in the following figures. Connector places are without shading.)

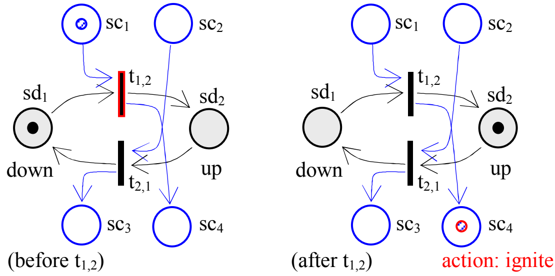

Figure 5. Desired state model before and after firing transition $t_{1,2}$.

The join rule applies to transition $t_{1,2}$ (multiple inbound arcs into $t_{1,2}$), which means that the activity token in sc_1 bonds with the one in sd_1. Transition $t_{1,2}$ is also a fork transition (multiple outbound arcs from $t_{1,2}$ leading to connector and non-connector places) such that the bonded tokens immediately separate. The activity token transitions into sc_4 while the token of the managed element transitions into sd_2 (new desired state [up]).

The arrival of the activity token in terminal connector place sc_4 can trigger an action in the managed environment to ignite the boot process.. Connector states sc_2 and sc_3 have the reverse effect when desired state is changed from [up] to [down].

Source connector places in DSM such as sc_1 and sc_2 provide the control elements for altering the desired state for a managed element. Terminal connector places in DSM such as sc_3 and sc_4 represent actions that are initiated for a managed element when an activity token arrives.

F. Reflecting Observed State Changes

Observed state changes as effect of reported changes from the managed environment. The same concept of connector places and activity tokens is applied. Source connector places are associated with sensors in the managed environment creating activity tokens when change is observed for a managed element. Activity tokens bond with the tokens of the managed elements for which changes were observed. Transitions of bonded tokens then lead to changes in the observed state model.

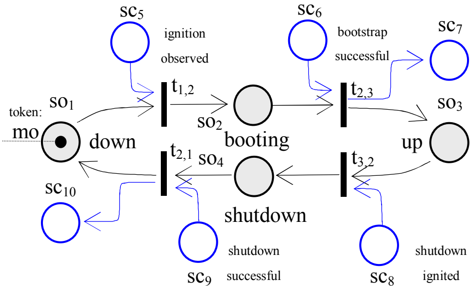

Figure 6. Observed state model with connector places.

Figure 6. shows the observed state model from Figure 4. supplemented with connector places for responding to changes in the managed environment. The token of a managed element resides in place so_1 ([down]). When the boot sequence is initiated (as effect of an activity token arriving in place sc_4 in Figure 5. and triggering ignition), this ignition can be observed in the managed environment and reported as an activity token arriving in sc_5, which enables transition $t_{1,2}$ and, during firing, bonds the activity token to token *mo*. The bonded token then resides in place so_2 ([booting]). When the boot sequence was completed successfully, an activity token is reported to so_6 enabling $t_{2,3}$. Based on the rules for bonded tokens, the token separates from the activity token during $t_{2,3}$, bringing the regular token to so_3 ([up]) and the activity token to sc_7 where it is consumed. Place so_3 is a place that is correlated with place sd_2 in the desired state model. The state of the managed element is now aligned with its desired state.

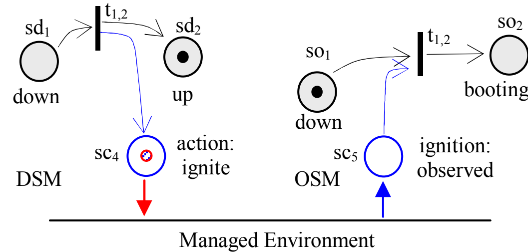

Figure 7. Linkage between initiating change and observing it.

The nets for DSM and OSM are indirectly connected through connector places. When a terminal connector place in the desired state model receives an activity token, an effect in the managed environment is triggered, which is reported back as another activity token arriving in a connector place in the observed state model. Occurrence of those activity tokens then can enable transitions in the observed state model. Figure 7. shows this indirect linkage between DSM and OSM. A direct linkage between connector places in DSM and OSM can also be established by connecting places through a direct transition.

G. Deriving Actions from Observed State Changes

In addition to intended changes in the managed environment which are derived from changes in the desired state model, error, failures and other conditions may occur in the managed environment any time. When those are reported, they also lead to the creation of activity tokens in connector places. The observed state model must take these conditions into account and must be designed accordingly.

For instance, the boot process of a server may end with a failure or may not complete within an expected time. An additional error place is introduced in the observed state model (so_5 in Figure 8.) to reflect those conditions.

The actual status of the server is unknown at this point. As long as the desired state still in [up], the observed state Petri Net may be designed in a way that it includes a sequence of corrective actions by attempting to reboot the server by:

1.) power cycle the server (bringing it into a defined state [down]) and

2.) re-igniting the boot sequence.

Figure 8. shows the PTN which handles these cases. Two new places so_5 [error] and so_6 [power cycle] have been introduced as well as five new connector tokens sc_{11} [boot error observed], sc_{12} [timeout], sc_{13} [retry], sc_{14} [initiate power cycle] and sc_{15} [power cycle completed]. The figure shows the token in place so_2 [booting], which has three possible outcomes

represented by three transitions $t_{2,3}$, $t_{2a,5}$ and $t_{2b,5}$ which are enabled by activity tokens arriving in places sc_6 [bootstrap successful], sc_{11} [boot error observed] or sc_{12} (timeout). Place sc_{12} receives an activity token after $t_{1,2}$ has fired, which starts the timer as side effect.

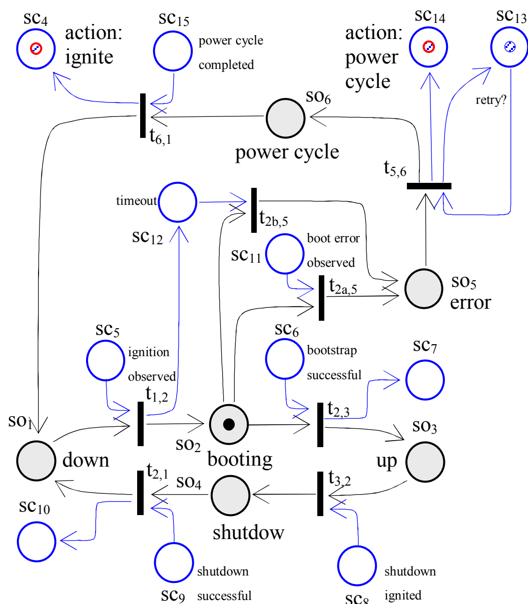

Figure 8. Extended observed state model with error correction.

In case of error, the token in place so_2 [booting] bonds with an activity token arriving from connector places sc_{11} [boot error observed] or sc_{12} [timeout] and transitions to place so_5 [error]. An activity token in place sc_{13} indicates that the controller should retry the boot cycle leading to the transitions to place so_6 [power cycle]. At this transition, activity tokens are separated and placed into sc_{13} (to maintain the retry marking) and sc_{14}, which is a terminal connector place that starts the power cycle. The end of the power cycle is indicated by an activity token arriving in place sc_{15}, which enables and fires transition $t_{6,1}$. This transition separates the activity token into sc_4 which ignites the bootstrap (see Figure 5.). The token now resides in the initial place so_1 [down] and the process repeats.

Whether or not the retry cycle will be performed in case of error depends on the marking of sc_{13}. This marking can be made dependent on whether or not the desired state for the server is still [up] or other conditions (not shown in the figure).

VIII. Controller Composition

Multiple controllers will interact in an automated IT management process chain, each responsible for a specific task or managed domain. Coordination among controllers is needed. Higher-ordered controllers mainly perform coordination tasks. They contain the composition models that span across underlying controllers and constitute an automated IT management process chain.

The self-similar structure of the IT management automation controller allows the composition of controllers as shown in Figure 9. Actions initiated by the upper controller are applied as desired state changes to underlying controllers. And reversely, observed state in underlying controllers constitutes the observed state of a higher-ordered controller. The interaction points among controllers are:

- higher Actuator Connector to lower DSM,
- lower OSM to higher Observer Connector to higher OSM.

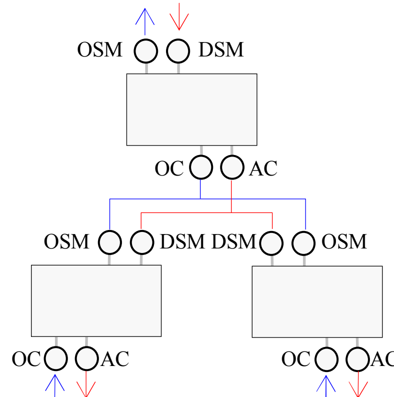

Figure 9. Composition of IT Management Automation Controllers.

Source and terminal connector places are the “interfaces” at the model level. Connector places are accessed through DSM or OSM Interfaces. Activity tokens arriving in terminal connector places can cause inter-controller activity. Activity tokens arriving in source connector tokens can then cause changes in PTN. Events can be issued to subscribers when the OSM or the DSM changes, which also lead to the creation of activity tokens in source connector places in PTN.

All those interactions are mediated through "get", “put”, "transfer", and "push/subscribe" operations defined for web service management standards [17].

IX. PTN Execution Engine

A PTN Execution Engine was implemented (in Java) that interprets DSM and OSM. It forms the core part of the IT Management Automation Controller. PTN Schema have been defined for DSM and OSM models which allows to represent PTN models in XML. Places are static XML fragments which can be addressed by xpath expressions in models. Tokens are represented as dynamic XML fragments that are associated with one place at a time.

The engine operates on the XML (SAX) trees of the DSM and OSM PTN models and interacts with the web service endpoint interfaces provided by the controller toolkit. Operating directly on XML tree representations of models also ensures that model state exposed through the DSM and OSM interfaces is always up to date. The engine is triggered when a new activity token arrives through one of the controller interfaces. Only arrival of activity tokens can alter model state.

Java class names are associated with terminal connector places, which are instantiated when activity tokens arrive. Updates or events associated with DSM or OSM are directly transformed into the creation of activity tokens in the addressed places along with the invocation of the engine.

X. Automation Use Cases

This section describes a controller-based automation scenario that was built as joint effort between a team from HP

and one from Oracle using the controller toolkit. The goal was to demonstrate model-driven automation for selected IT task automation use cases. The testbed consisted of HP blade servers (eight servers of type BL20p, dual Pentium III, 3.2GHz, 4GB), HP SAN disk array with two fiber channel switches, and a HP ProCurve 2848 LAN network.

The automation use cases demonstrated coordinated lifecycle and auto-correction capabilities for error situations that would require human attention during operation in a traditional system. The three automation use cases were:

1.) Automated provisioning and coordinated lifecycle control of an Oracle database on blade servers.

2.) A storage auto-correction capability by automatically configuring and attaching new disks from the SAN to servers when Oracle Enterprise Manager predicted storage shortage due to growing table sizes in the database.

3.) Response-time auto-correction during operation by flexing additional blades into the database pool when response times increased above a threshold due to load increases.

All use cases required the direct interaction between the two management systems from HP and Oracle. Neither system could achieve them alone. All interactions between systems were normalized as model exchanges between controllers.

To actuate actual changes in the managed environment, two management systems were employed: HP blade server automation software and Oracle Enterprise Manager. Both systems had to cooperate in order to solve the automation use cases, which would have required the point integration between the two systems in a traditional approach. Instead, both systems were wrapped into controllers using the controller toolkit.

Three controllers were created with models:

- HP blade automation controller,
- Oracle Enterprise Manager controller, and an additional
- Coordinator controller.

The first two controllers were implemented as wrappers around HP blade automation infrastructure and Enterprise Manager from Oracle. The third controller was created for coordinating the two other controllers. It coordinated activities and composed them into one management service achieving all three use cases.

The first use case allowed for basic provisioning of blade servers, disks and networks using HP blade automation infrastructure. It deployed Oracle 10g on bare servers and configured it for management through Oracle Enterprise Manager. Two template models were available for instantiating the database representing three different configurations ("sizes") of the Oracle database deployment: small (one blade server), medium (two servers) and large (4 servers). Templates were chosen from the coordinator controller based on user input. This specification was based on the number of users, data set size and the transaction rate supported by a configuration. After template selection, the coordinator controller interacted with the underlying controllers to establish the needed hardware infrastructure using the capabilities of HP's blade server automation software and, once this had been achieved, to initiate the configuration of Oracle.

The second use case employed Oracle Enterprise Manager's ability to predict shortage of storage in a growing database and triggering correction by notifying the HP blade automation controller to attach another disk from the SAN storage array. After completion, the Oracle controller was initiated to reconfigure the database in order to utilize the additional disk.

The third use case allowed to auto-correct server capacity when slowing database response time was indicated by Oracle Enterprise Manager, triggering a server flex-up operation to the HP blade automation controller. After completion, database instances configured into the database.

These use cases demonstrated automation scenarios achieving self-correcting behavior. They also demonstrated how controllers can be used to wrap legacy management systems into a controller framework, normalizing their interactions using common controller interfaces and model exchange through interfaces. Use of PTN allowed describing and executing the coordination needed between controllers providing an example for an automated IT management process chain.

XI. Related Approaches

While automation has made substantial progress on the business side of IT, such as in business process automation [27], automation in IT management has been lagging behind. On the business side of IT, enterprise software such SAP is widely been used to automate the processing and management of enterprise information. Tools such as ARIS [28] are used to design automated business processes. In IT management, in contrast, people still carry out management processes from higher-ordered planning stages to the lowest levels of managing machines, networks and storage. Management tools are used that support those tasks. Tools signal and report conditions to a human operator, who then is in charge to interpret those signals and eventually respond by making a change in the system, which is again mediated through a tool. The loop is not closed in IT management; the operator's attention is permanently required.

Workflow systems are predominantly used in IT management automation. In contrast to workflow languages, which describe sequences of parallel or sequential actions, Petri Nets primarily represent state. State changes occur in effect of transitions, which are also described in a Petri Net. State in a workflow language is always external to the actual workflow description. It typically occurs in form of a message that is processed along the workflow statements (e.g. a purchase order, which is the message, traveling through a purchase order workflow, which is a graph of actions).

Cfengine was developed at University College in Oslo [29]. Its primary function is to provide automated configuration and maintenance of computers, from a policy specification. It emerged from the need to control the accumulation of complex shell scripts used in the automation of key system maintenance. In a heterogeneous environment, shell scripts are hard to maintain: shell commands have differing syntax across different operating systems, and the locations and names of key files differ. The non-uniformity of Unix was a major problem. Cfengine defined a new language which unified the

heterogeneity underneath. The aim was to absorb frequently used coding paradigms into a declarative, domain-specific language that would offer self-documenting configuration. Cfengine has an agent-based infrastructure through which scripts can be distributed and executed on machines.

While Cfengine allows to abstract and to unify scripts used in system management, it does not posses the capabilities of a controller. Cfengine needs to be activated by an administrator in order to perform management tasks on remote systems.

XII. SUMMARY

The paper presented an IT Management Automation Controller which adopts the concept of a feedback system to automated IT management for operational tasks such as lifecycle management. State of a managed environment is represented in terms of a pair of models for desired and observed states and transitions between those states. A specialized form of a Place-Transition-Net (PTN or Petri Net) is used to represent the static aspects (states) as well as dynamic aspects (coordination) in one model. This overcomes the problems that result from separating models from interpretation and execution logic that is often found in model-based management approaches.

A PTN execution engine was built that directly executes PTN and forms the core of the controller logic. This allows the controller logic to be "generic" and driven by configurable PTN models as opposed to hard-coded and built into the controller. Web services management standards are used to create uniform interfaces to controllers and that allow the composition of controllers. It was shown how error correction can be factored into PTN. It was also shown how coordination between controllers in an automated IT management process chain can be achieved.

Three automation tasks for a database system have been implemented using the controller.

REFERENCES

[1] *IT Service Management, The ITIL and ITSM Directory*, http://www.itil-itsm world.com.

[2] Hellerstein, J.L., Diao, Y., Parekh, S.S., Tilbury, D.: *Feedback Control of Computing Systems*, John Wiley & Sons, New York, 2004.

[3] Hellerstein, J.L., Gandhi, N., Parekh, S.S.: *Managing the Performance of Lotus Notes: A Control Theoretic Approach*, International CMG Conference, 397-408, 2001.

[4] Wang, Z., Zhu, X., Singhal, S.: *Utilization and SLO-Based Control for Dynamic Sizing of Resource Partitions*, Distributed Systems: Operations and Management Workshop (DSOM 2005), Barcelona, Spain, October 24-26, 2005.

[5] Astrom, K.J., Wittenmark, B.: *Adaptive Control (2nd Edition)*, Prentice Hall, 1994.

[6] Xu, W., Zhu, X., Singhal, S., Wang, Z.: *Predictive Control for Dynamic Resource Allocation in Enterprise Data Centers*, 2006 IEEE/IFIP Network Operations and Management Symposium (NOMS 2006), Vancouver, Canada, April 3-7, 2006.

[7] Welsh, M., Culler, D.: *Adaptive Overload Control for Busy Internet Servers*, 4th USENIX Symposium on Internet Technologies and Systems (USITS 2003), Seattle, March 2003.

[8] Liu, X., Zhu, X., Singhal, S., Arlitt, M.: *Adaptive Entitlement Control of Resource Containers on Shared Servers*, IFIP/IEEE International Symposium on Integrated Network Management (IM 2005), Nice, France, May 2005.

[9] Kephart, J., Chess, D.M.: *The Vision of Autonomic Computing*, IEEE Computer 36(1), 41-50, 2003. http://researchweb.watson.ibm.com/ autonomic.

[10] Smith, B.F., Kreitz, J., Wilson, M.M.: *Autonomic IMS and IMS Tools*, The Mainstream The IBM eServer zSeries and S/390 Software Newsletter, Issue 8, April 12, 2004.

[11] Coleman, D., Cook, N., Eidt, E., Fleck, J., Graupner, S., Mukerji, J., Singhal, S., Thompson, C.: *Specification of the Service Delivery Controller (SDC)*, Hewlett-Packard, July 2005.

[12] Graupner, S., Cook, N., Coleman, D., Nitzsche, T.: *Management Middleware for Enterprise Grids*, 6th IEEE International Symposium on Cluster Computing and the Grid (CCGrid 2006), Singapore, May, 2006.

[13] Sahai, A., Graupner, S.: *Web Services in the Enterprise: Concepts, Standards and Management*, Springer Verlag, ISBN 0-387-23374-1, 310 Seiten, 2004.

[14] OASIS TC and Global Grid Forum: *The Web Services Resource Framework (WSRF) v1.2*, April 2006, http://www.oasis-open.org/specs/index.php#wsrfv1.2.

[15] OASIS: *WSDM: Management Using Web Services (MUWS 1.0) and Management Of Web Services (MOWS 1.0)*, March 2005. http://www.oasis-open.org/specs.

[16] *Globus Toolkit GT4*, http://www.globus.org.

[17] *Web Services for Management (WS-Management) v1.0*, April 2006, http://www.dmtf.org/standards/wsman.

[18] *Service Modeling Language Specification v0.5*, Draft Specification, July 2006. http://go.microsoft.com/fwlink/?LinkId=70293.

[19] Thompson, C., Coleman, D.: *Model Based Automation and Management for the Adaptive Enterprise*, 12th Annual Workshop of HP OpenView University Association, Porto, Portugal, July 10-13, 2005.

[20] Petri, C.A.: *Kommunikation mit Automaten*, Ph.D. Dissertation, University of Bonn, Germany 1962.

[21] Petterson, J.L.: *Petri Net Theory and the Modeling of Systems*, Prentice-Hall, Englewood Cliffs, 1981.

[22] Jensen, K.: *An Introduction to the Practical Use of Coloured Petri Nets*, In: W. Reisig and G. Rozenberg (eds.): Lectures on Petri Nets II: Applications, Lecture Notes in Computer Science vol. 1492, pages 237-292, Springer-Verlag 1998.

[23] K. Jensen: *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use*, Volumes 1-3, Monographs in Theoretical Computer Science, Springer-Verlag, ISBN: 3-540-60943-1, 2nd corrected printing, 1997.

[24] *CPN Tools*, http://wiki.daimi.au.dk/cpntools/cpntools.wiki.

[25] Huber, P., Jensen, K., Shapiro, R.M.: *Hierarchies in Colored Petri Nets*, In: G. Rozenberg (ed.): Advances in Petri Nets 1990, Lecture Notes in Computer Science Vol. 483, pages 313-341, Springer-Verlag, 1991.

[26] Choo, Y.: Hierarchical Nets: *A Structured Petri Net Approach to Concurrency*, Technical Report CaltechCSTR:1982.5044-tr-82, California Institute of Technology, 1982.

[27] Scheer, A.W., Abolhassan, F., Jost, W., Kirchmer, M. (Ed.): *Business Process Automation*, ISBN 3540207945, Springer Verlag, 2004.

[28] IDS Scheer: *Aris*, http://www.ids-scheer.com.

[29] Cfengine, http://www.cfengine.org.