H-0254 (H0612-005) December 13, 2006
Computer Science

IBM Research Report

Policy Verification for System Automation with Model Checking: A Case Study

Emmanuel Zarpas, Cindy Eisner, Sivan Tal
IBM Research Division
Haifa Research Laboratory
Mt. Carmel 31905
Haifa, Israel

Research Division
Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich

Policy Verification for System Automation with Model Checking: A Case Study

Emmanuel Zarpas, Cindy Eisner, and Sivan Tal
IBM Haifa Research Lab.
{zarpas, eisner, sivant}@il.ibm.com

Abstract

System automation through policy-based management allows IT administrators to define high-level policies for various management tasks, such as networked systems and applications for business environments, network planning, problem detection, and quality of service provisions. Policies can be understood as specifications; therefore they can be translated more or less easily into formal languages and then be verified by formal techniques such as model checking. In this paper, we focus on formal verification of real-life industrial policies of the Tivoli System Automation for Multi-Platform (TSA). We use PSL to model the system and describe the desired behavior and the RuleBase PE model checker to verify it.

1. Introduction

Every day, IT systems become more complex, heterogeneous, and interconnected, and therefore pose greater challenges to IT administrators. System automation through policy-based management allows IT administrators to define high-level policies for various management tasks, such as networked systems and applications for business environments, network planning, problem detection, and quality of service provisions. This approach (e.g., [1] and [15]) to system management allows us to separate the rules that govern behavioral choices of the system from the functionality provided by that system. In a very general way, policies are plans of an organization to achieve its objectives. A policy can be understood as a high-level specification of the system to be automated by TSA. It is, therefore, natural to translate it to a formal language and then to verify it.

In this paper, we focus on the formal verification of policies with model checking [5], [6]. More precisely, we present model checking verification through a case study for IBM Tivoli System Automation for Multi-Platform [18]. Our work follows a very concrete approach. We translate real-life industrial policies into PSL [21] and then verify the system with the RuleBase PE model checker [22]. In this way, we can handle properties that are difficult to check with testing and simulation. While Sinz *et al*. [10], [11] used an extension of Propositional Dynamic Logic [13] to detect loops in the TSA rule-evaluation engine; our approach focuses on verification of the higher level policies using PSL.

The rest of the paper is organized as follows. In Section 2 we define the PSL fragment we use in this paper. Section 3 discusses our general methodology, and Section 4 provides background information on TSA policies. Section 5 then describes how we model policies with the PSL modeling language and Section 6 covers the PSL properties we checked with RuleBase. We then present a case study in Section 7 and conclude with Section 8.

2. Preliminaries

We use the specification language PSL [4], recently standardized as IEEE 1850™-2005 [21]. The temporal layer of PSL contains the temporal logics LTL and CTL, regular expressions and hardware-specific features such as clocks and the *abort* operator. Due to lack of space, we have chosen to provide the semantics of only the fragment of PSL that we use in this paper, corresponding to LTL [7] for the assumptions used to model the policy and CTL [2] for the properties to be verified.

Both CTL and LTL are subsets of the temporal logic CTL* [3], which is defined as follows:

- Every atomic proposition is a formula.
- If f and g are formulas, then so are $\neg f$ and $f \wedge g$.
- If f is a formula, then $E f$ is also a formula.

- If f and g are formulas, then $f\ U\ g$ and $X\ f$ are also formulas.

Additional operators can be viewed as abbreviations of the above, as follows:

- $f \vee g = \neg(\neg f \wedge \neg g)$
- $F\ g = true\ U\ g$
- $G\ f = \neg(true\ U\ \neg f)$
- $f\ V\ g = \neg(\neg f\ U\ \neg g)$
- $A\ f = \neg E\ \neg f$

The semantics of a CTL* formula is defined with respect to a model M. A model is a quadruple (S, S_0, R, L), where S is a finite set of states, $S_0 \subseteq S$ is a set of initial states, $R \subseteq S \times S$ is the transition relation, and L is the valuation, a function mapping each state with a set of atomic propositions true in that state. We require that there is at least one transition from every state.

A computation path π of a model M is an infinite sequence of states $\pi = (\pi_0, \pi_1, \pi_2, \ldots)$ such that $R(\pi_i, \pi_{i+1})$ is true for every i. Given a computation path π, we will denote by π^i the computation path starting from the i-th state in π. More formally:

$\pi^i = (\pi_i, \pi_{i+1}, \pi_{2i+2}, \ldots)$.

The semantics of CTL* is then as follows:

- $(M, \pi) \models p \Leftrightarrow p \in L(\pi_0)$, where p is an atomic proposition.
- $(M, \pi) \models \neg f \Leftrightarrow (M, \pi) \not\models f$.
- $(M, \pi) \models f_1 \wedge f_2 \Leftrightarrow (M, \pi) \models f_1$ and $(M, \pi) \models f_2$.
- $(M, \pi) \models E\ f \Leftrightarrow$ for some computation path π' in M, starting from π_0, $(M, \pi') \models f$.
- $(M, \pi) \models X f \Leftrightarrow (M, \pi^1) \models f$.
- $(M, \pi) \models f_1\ U\ f_2 \Leftrightarrow \exists\, n \geq 0$ such that $(M, \pi^n) \models f_2$, and for all i such that $0 \leq i <$ n, we have $(M, \pi^i) \models f_1$.

We say that $M \models f$ iff for every computation path π in M, such that $\pi_0 \in S_0$, we have $(M, \pi) \models f$.

CTL is a subset of CTL* in which each temporal operator (F, G, U, V, and X) must be immediately preceded by a path quantifier (A or E).

LTL is a subset of CTL* in which there are no path quantifiers.

3. Method

Policies can be understood as rules that must be followed in the pursuit of a set of objectives. While the policy changes per system, the objectives for each system are the same. For instance, the policy might specify that node A must always be shut down before node B is shut down, while an objective might be that some desired state (e.g. all nodes are online) is reachable.

Our goal is to show whether, for a given policy, there exists a program that obeys it while achieving its objectives. We assume that the automated system does not interact with the policy. For instance, if a system tells node A to shut down, we assume that node A does not have the ability to object, and we also assume that node B does not have the ability to prevent the shutdown of node A. Thus, our system is *closed* [8], [14] and it is sufficient to show that there exists a model obeying the policy that achieves the objectives (one of the objectives will be that the model is non-empty). Formally, let P be the policy, represented as a set of temporal logic formulas, and let F be a set of temporal logic formulas representing the objectives. Then it is sufficient to show that there exists model M such that $M \models P$ and $M \models F$. We use the RuleBase model checker [22] which implements the PSL directives *assume* and *assert* in the following manner:

assume: Given a model M and a linear formula ϕ, assume ϕ creates a model M_ϕ such that $M_\phi \models \phi$ by removing states, transitions, and computation paths from M. The set of assumptions $\Phi = \{assume\ \phi_1, assume\ \phi_2, \ldots, assume\ \phi_n\}$ creates M_Φ and is equivalent to assume $\phi_1 \wedge \phi_2 \wedge \ldots \wedge \phi_n$.

assert: Given a model M, a set of assumptions Φ, and a linear or branching formula ψ, *assert* ψ checks whether $M_\Phi \models \psi$.

We run the model checker using the model M as described in Section 5. Now, we assume our policy P and assert our objectives F. If all of the objectives hold, we have shown that there exists M' such that $M' \models P$ and $M' \models F$, as needed.

4. TSA Policy Description

TSA is an application designed to provide a single point of control for a full range of system management functions. It manages the availability of applications running on Linux systems or clusters. Although TSA has several features, this paper focuses on its policy-based automation. TSA allows users to configure high availability systems through the use of policies that define relationships among the various components. Once the relationships are established, TSA assumes responsibility for managing the applications on the specified nodes as configured. This reduces implementation time and the need for complex coding of applications. In this paper, we do not describe the verification of TSA software or of any particular implementation of a TSA policy. Rather, we verify the

policy itself by performing sanity checks such as conflict, dead-lock and loop detection on TSA policies. A TSA policy is, roughly speaking, a collection of relationships that describes the automated behavior to be enforced by TSA. TSA describes temporal relationships (e.g., A should start after B) or topological relationships (e.g., A is co-located with B) between resources that should be enforced by the system. The building blocks of TSA policies are resources, which can be any piece of hardware or software in the TSA management scope, located on several nodes of the system. There are three types of resources in the TSA policy language: fixed resources (`Resource`), floating resources (`MoveGroup`), and references to a resource outside the management scope of TSA (`ResourceReference`). Resources can be grouped using the constructs `ResourceGroup` or `Equivalency`, so that they are easier to handle. TSA policies are described with XML syntax. The following section includes a brief summary of the TSA policy description language; see [19] for a detailed description. A resource is described using the keyword `Resource`, and specifying the name, node, type and parameters -- see Figure 1 for an example.

```
<Resource
name="XI_J2EE_LOP_achalm47_AS"
class="IBM.Application"
node="achalm47">
  <ClassAttributesReference>
          <IBM.ApplicationAttributes
name="IBM.Application.A1"/>
  </ClassAttributesReference>
 </Resource>
```

Figure 1 Resource example. This resource is an application, whose parameters are described in IBM.Application.A1.

Keyword `MoveGroup` is used for floating resources. A move group is defined by its name, type and parameters and its constituent resources. The constituent resources act as resources of the same type and parameter as the move group, however only one of them can be online at any given time.

Keyword `ResourceGroup` is used to group resources together, so they can be handled in an easier way. A resource group is defined first by its name, type and its members, then by constraints on its members (for instance, the members of a co-located resource group have to run on the same node). In addition the desired state of the resource group is specified. See Figure 2 for an example of a resource group.

```
<ResourceGroup name="SA-samba-rg"
class="IBM.ResourceGroup">
  <DesiredState>Online</DesiredState>
<Members>
<MoveGroup name="SA-samba-server"
class="IBM.Application"
selectFromPolicy="Ordered"
mandatory="true" />
<MoveGroup name="SA-samba-data-work"
class="IBM.Application"
selectFromPolicy="Ordered"
mandatory="true" />
<MoveGroup name="SA-samba-ip-1"
class="IBM.ServiceIP"
selectFromPolicy="Ordered"
mandatory="true" />
</Members>
<MemberLocation>Collocated</MemberLoc
ation>
<Priority>0</Priority>
<AllowedNode>ALL</AllowedNode>
</ResourceGroup>
```

Figure 2 Resource group example. This resource group encompasses three move groups and is co-located (i.e. the three move groups have to run on the same node) as specified by keyword `Collocated`. The members of the resource group can run on any node (as long as they all run on the same one) as specified by keyword `AllowedNode`. Keywords `Priority` and `selectFromPolicy` are used by TSA to manage conflicts.

The keyword `Equivalency` is another way to group resources. It is used to describe a collection of resources, all providing the same functionality. An equivalency consists of a set of fixed resources from the same resource class. For example, network adapters might be defined as equivalency resources. If one adapter goes offline, another network adapter can take over the processing from the offline adapter. These resources are given explicitly or implicitly. An equivalency is described by its name, type, its members and the minimum number of members that need to be online in order for the equivalency to be online.

The keyword `Relationship` is used to describe the relationships between resources in a cluster. The source (a resource, a resource group or a move group) and target (a resource, resource group, resource reference, move group, or equivalency) are defined either explicitly or implicitly. See Figure 3 for an example of a relationship.

```
<Relationship name="SA-samba-server-
on-data-work">
<Source>
  <MoveGroup name="SA-samba-server"
class="IBM.Application"/>
</Source>
```

```
<Type>DependsOnAny</Type>
<Target>
  <MoveGroup name="SA-samba-data-
work" class="IBM.Application" />
</Target>
</Relationship>
```

Figure 3 Relationship example. Using the keyword `DependsOnAny`, this relationship states that in order for move group SA-samba server to run, SA-samba-data-work should be running.

Additional types or conditions not yet shown can be specified for some location relationships. The possible types/conditions are: A `StartAfter` B, A `StopAfter` B, A `DependsOnAny` B, A `DependsOn` B, A `ForcedDownBy` B, A `Collocated` B, A `Anticollocated` B, A `IsStartable` B, A `Collocated/IfOnline` B, A `Collocated/IfNotOnline` B, A `Collocated/IfOffline` B, A `Collocated/IfNotOffline` B, and `AntiCollocated/*`. Details on the semantics are given in Section 5.5.

5. Modeling

This section describes how we model TSA policies. Fixed and floating resources are modeled as state machines in the GDL flavor of the PSL modeling layer [22] and the relationships are modeled as PSL assumptions. Time in the systems we are modeling is continuous, while PSL time is discrete. We deal with this by allowing events to happen at a non-deterministic time and by considering the atomic unit of time to be the minimum possible time between two events in the system.

5.1 Resources

The TSA description provides the name and node of a resource. The attribute class and information specified by the `ClassAttributesReference` keyword appearing in Figure 1 are not relevant for the properties that need to be checked. A resource can have five states: Unknown, Online, Offline, FailedOffline, and StuckOnline. A resource state is Unknown when its state is not known by TSA for some reason; a resource is Online when it is running and Offline when it is not running. A resource is FailedOffline when it is down with a fatal failure and StuckOnline when it is running with a fatal failure. Possible transitions (where a transition takes one atomic unit of time) for the resource state are:

Unknown -> *Unknown* | *Online* | *Offline* | *FailedOffline* | *StuckOnline*
Online -> *Unknown* | *Online* | *Offline* | *FailedOffline* | *StuckOnline*
Offline -> *Unknown* | *Online* | *Offline* | *FailedOffline*
FailedOffline -> *FailedOffline*
StuckOnline -> *StuckOnline*

The amount of time a resource stays in a specific state is non-deterministic and independent of the behavior of other resources. Resources, resource groups, move groups and equivalencies are coded in the PSL modeling language as an array. The first part of the array codes the node and the second part codes the state. For simplicity's sake, in this paper we denote the node and the state of a resource "r" by r.node and r.state. Resource transitions are constrained by the relationship (see Section 5.5). The type and parameters specified by `ClassAttributesReference` are related to the implementation of the resource and therefore outside of the scope of this work -- they are not needed to prove the Section 6 properties. Constituent resources and resource references are modeled as resources.

2 Resource groups

A resource group is coded in the PSL modeling layer as an array, similar to the method used for resources. The node of a resource group is relevant only if the resource group is co-located, in which case it is the node of its members. If the resource group is co-located, the relevant co-location constraints between its members are added to the list of relationships (see Section 5.5). `AllowedNode` and `ExcludeNode` are coded in a straightforward fashion as a PSL constraint on the node fields of the resource group members. The priority related fields have no impact on the properties to be verified (they only give information to the TSA engine on the way to implement the policy). The state of a resource group RG is defined from the states of its members, as follows:

```
RG.state := case
  all members are Online             :
Online ;
  all members are Offline            :
Offline ;
  one member is StuckOnline          :
StuckOnline ;
  one member is FailedOffline        :
FailedOffline;
```

```
  else                                    :
Unknown ;
esac ;
```

5.3 Move groups

Only one of the move group members (constituent resources) can be Online or StuckOnline at a given time. Therefore if CR1,…,CRn, represent the constituent resources of a move group, we add the following PSL layer verification directive (the `assume` statement enables us to specify an invariant):

```
  assume always ((CR1.state in
  {Online, StuckOnline} or…or
  CRn.state in {Online, StuckOnline})
  -> (CR1.state in {Online,
  StuckOnline} xor … xor CRn.state in
  {Online, StuckOnline})) ;
```

The state of a move group MG is defined from the states of its members, as follows:

```
MG.state := case
  one of the members is online       :
Online ;
  all members are StuckOnline        :
StuckOnline ;
  all members are FailedOffline      :
FailedOffline;
  else                               :
Offline ;
esac ;
```

If a move group is online, its node is the node of its online constituent. Otherwise, its node is not relevant. As for fixed resources, type and `ClassAttributesReference` are not modeled.

5.4 Equivalencies

The state of an equivalency E is defined from the states of its members, as follows: An equivalency is Online if n of its members are Online (n being equal to the value of the field `MinimumNecessary`).

```
E.state := case
  n members are Online               :
Online ;
  all members is StuckOnline         :
StuckOnline ;
  all members is FailedOffline       :
FailedOffline;
  else                               :
Offline ;
esac ;
```

5.5 Relationships

Relationships are modeled as constraints using the PSL verification layer directive `assume` (as in Section 5.3 assume statement allows us to specify an invariant). This allows us to provide a formal description of TSA policy relationships that are only informally described in [18] and [19].

A StartAfter B means that A must start after B starts. More precisely, when A starts, B should already be online. This is translated to the following PSL verification directive:

```
assume always(rose(A.state=Online)
-> (B.state=Online &
!rose(B.state=Online))) ;
```

This means the following property should be an invariant of the model: when A goes online, B should be online but did not go online at the same moment A did (*always p* is PSL syntax for the LTL *G p*). This is more complex than expected. Translation to PSL allows a clearer and non-ambiguous description of the relationships.

A StopAfter B means that A must stop after B does, i.e., when A stops, B is already offline:

```
assume always (fell(A.state=Online)
-> (B.state in {Offline,
FailedOffline} & !rose(B.state in
{Offline, FailedOffline}))) ;
```

A DependsOnAny B means that A cannot be online if B is not online:

```
assume always (A.state=Online ->
B.state=Online) ;
```

The difference between *A DependsOnAny B* and *A StartAfter B* is that with the former A and B can go online at the same time, and B should stay online as long as A is online.

A DependsOn B means that A DependsOnAny B and A collocated B:

```
assume always (A.state=Online ->
(B.state=Online & A.node=B.node)) ;
```

A ForcedDownBy B means that when B is down, it forces A to be down:

```
assume always(rose(B.state in
{Offline, FailedOffline}) -> next
(A.state in {Offline,
FailedOffline})) ;
```

We assume that A is forced down in one atomic unit of time after B (*next p* is PSL syntax for LTL *X p*). This restriction reduces the complexity but is very limiting because there is no guarantee that TSA can bring down A in only one atomic unit of time. However, this relationship can also be modeled in a more general way for a given k:

```
assume always(rose(B.state in
{Offline, FailedOffline}) ->
```

```
next_e[1..k](A.state in {Offline,
FailedOffline})) ;
```

In other words, when B goes into the Offline or FailedOffline state, then A should go Offline or FailedOffline with 1 to k atomic units of time (*next_e[1..k] p* is PSL syntax for *X*p | XXp | … | XX…X p i.e., between 1 and k X operators). Generally and without reference to any "clock", the relationship can be modeled as follows:

```
assume always(rose(B.state in
{Offline, FailedOffline}) ->
eventually! (A.state in {Offline,
FailedOffline})) ;
```

That is, if B goes into the Offline or FailedOffline state, then A should eventually go Offline or FailedOffline (*eventually! p* is PSL syntax for LTL *F p*).

A Collocated B means that if A is online, A and B are on the same node:

```
assume always ((A.state=Online) ->
A.node=B.node) ;
```

A Anti Collocated B means that if A is online, A and B are not on the same node:

```
assume always ((A.state=Online) ->
A.node!=B.node) ;
```

A IsStartable B means that A can only run on a node that B will be able to run on in the future (i.e., B is not FailedOffline on this node):

```
assume always((A.state=online) ->
!(B.state=FailedOffline &
A.node=B.node));
```

A Collocated/IfOnline B means that if A is online and B is online or StuckOnline, then A and B are on the same node:

```
assume always ((A.state=online) ->
(B.state in {Online, StuckOnline} -
> A.node=B.node)) ;
```

AntiCollocated/IfOnline, Collocated/IfNotOnline, AntiCollocated/IfNotOnline, Collocated/IfOffline, AntiCollocated/IfOffline, Collocated/IfNotOffline and AntiCollocated/IfNotOffline are defined in a similar way.

5.6 Parameters and Initial State

Parameters (IBM.ApplicationAttributes, IBM.ServiceIPAttributes, IBM.TestAttributes, and IBM.TieBreaker) are not included in our model. They describe the characteristics of the resources needed for the automation. These characteristics are totally independent of the relationships; therefore, we don't need to model them to prove properties on the relationships.

TSA policies don't specify the system's initial state. It is possible to either leave the initial state free (i.e., assign a non-deterministic value to each resource) or to assign a fixed value such as Offline or Unknown. The first option implies that every state is reachable, which may be too general. Therefore the second option is a safer choice.

6. Verification

Once the model is built, we can check PSL properties against it in order to perform:

- Conflict detection and see if it is possible to enforce all relationships,
- Validation of the policy specified to ensure it is consistent with the capabilities of the system
- Deadlock detection
- Loop detection.

We don't check how the system managed by TSA behaves; rather, we check properties on the policy controlling its behavior. For example, we check that the policy is not over-constrained in ways that prevent the system from running satisfactorily, we check that the system can reach the desired state, and we identify whether there exists a single point of failure with regard to these properties. The following PSL properties should hold for every policy:

1. `assert EF nominal_state ;`
2. `assert AG EX true ;`
3. `assert AG (desired_state1 -> EF desired_state2) ;`
4. `assert AG (desired_state1 -> EX desired_state1) ;`

where `nominal_state` is true when all resource groups are in the desired states specified in the policy, and `desired_state1` and `desired_state2` are chosen non-deterministically from all the desired states of the system. A desired state of the system is a state in which each resource group is in a known state, and not failed or stuck. Thus there are two "good" values per resource group – Online and Offline, and 2^n possible values of `desired_state1` and `desired_state2`.

Property 1 means the system can reach the nominal state specified by the policy. Property 2 means the system can always follow the policy; i.e., there is no truncated path. Property 3 means that while running and in a desired state, the system can reach any other desired state (for instance, if resource group X is

offline, all other resource groups are online, and it is desired to bring resource group X online and take groups Y and Z offline, that can be done). Property 4 means that once the system reaches a desirable state, it can stay there forever (this can be seen as some sort of termination property; it ensures, for instance, that no loop will prevent the system to stay as long as needed in the desired state). RuleBase will automatically check that the model is not empty. These properties are rather different to the properties commonly used in hardware verification like for example in [16]. The most commonly used properties used for hardware verification are safety properties and non-LTL properties are rather uncommon.

In addition, we can look for a single point of failure by checking the previous properties with any resources in a terminal state that fails offline. For instance, for Property 1, we can check that the nominal state can be reached if resource r fails:

```
assert AG(r.state=FailedOffline ->
EF nominal_state)  ;
```

This would check that nominal state can be reached even if some resourced r failed.

The properties we have shown so far should hold for every policy, and thus checking them can be completely automated. In addition, it is possible to perform policy-specific checks using RuleBase. Although it allows thorough verification, these kinds of properties must be manually coded and thus require that the user have some knowledge of PSL. However, if the user is willing to write PSL properties, the verification is no more difficult than push-button conflict detection. For instance, the following is a hand-coded property for the policy described in Section 7:

```
assert (!(XI_ABAP_LOP_node1=Online
| XI_ABAP_LOP_node2=Online) until
!(XJ_JEE_LOP_node1=Online |
XJ_JEE_LOP_node2=Online)) ;
```

i.e. if the ABAP stack did not start yet on either of the two nodes, then the JEE stack cannot have started either on either of the nodes.

We built an *ad hoc* translator that semi-automatically translates the XML TSA policy into a model (described in the previous section) and extracts definitions needed for the automated properties. Then, we checked these properties with the RuleBase PE model checker. We verified several real-life TSA policies, one of which is described in the next section.

7. Case Study

In this section, we describe the verification of a TSA policy for SAP and the experiments we used to check the scalability of our approach.

7.1 SAP System

Our work describes a TSA policy for a highly available SAP system with a J2EE application server. For more details, see [12]. SAP is widely acclaimed ERP software from SAP AG. We consider a minimum hardware setup as consisting of a two-node TSA domain. The two nodes are either physical machines or dynamic logical partitions (LPAR) with each LPAR running on a different physical machine. The machines must be connected via a network, for example, Gigabit Ethernet. Also, each machine needs access to a shared database and SAP data, for example, as provided by a Storage Area Network **(**SAN) attached disk subsystem, which is attached to each node via a fiber channel.

As required in a two-node TSA domain, we dedicated a third, very small, FAStT-disk as the TSA quorum disk. This disk is not shown in the following figure and is exclusively used as a quorum disk, functioning as a tie-breaker in the tested two-node domain.

Each machine/LPAR must be capable of running the basic SAP program resources that TSA for Multi-Platform make highly available via 'switch-over' groups. These program resources include the NFS server, the database server, and at least one application server instance (e.g., the Advanced Business Application Programming (ABAP) SAP Central Service (ASCS), and/or the SAP Central Service (SCS) instance).

A SAP system needs utility programs. If the SAP Router or SAP Web Dispatcher program is used, it must also be highly available. In addition, TSA MP should automate the SAP Operating System COLlector (SAPOSCOL) program.

Figure 4 illustrates a sample two-node TSA domain. It shows all basic program resources and their corresponding switch-over groups for a SAP system running an 'add-in' application server (such as SAP XI). The utility programs are not shown. Each add-in application server consists of a J2EE Central/Dialog Instance and an ABAP Dialog Instance; therefore, we also need the ASCS and the SCS instances in parallel.

The Highly Available sample policy for SAP version 4.0 is required to automate the setup. This version enhances ABAP-only SAP systems to J2EE-only SAP systems and extends to the most complicated SAP

systems that cover both ABAP and J2EE stacks. From the TSA MP viewpoint, J2EE-only SAP systems are very similar to ABAP-only SAP systems. This is different from SAP systems that support ABAP and J2EE stacks simultaneously, such as SAP XI. An add-in system runs two SAP Central Service instances, an ASCS and a J2EE SCS in parallel, and the add-in application server. An add-in application server is physically one instance (running both ABAP and J2EE stacks) with two logical parts: the ABAP application server instance and the J2EE application server instance. In other words, within a TSA domain, one add-in application server instance is automated as two logical application server instances. However, there is a tight relationship between the two logical application servers, in that the J2EE instance always starts after the ABAP instance. This StartAfter relationship guarantees that starting the J2EE instance immediately triggers the start of the ABAP instance. On the other hand, stopping the J2EE application server does not stop any of the add-in server processes; it only stops the monitoring 'java GetWebPage' Java program, which does a primitive health check of the J2EE application server.

The High Availability SAP policy has two nodes, where each node has three resources (XI_J2EE_LOP, SAP_SYS_SAPOSCOL_EXE, and XI_ABAP_LOP), forty-four constituent resources, and two referenced resources (network interfaces). These resources are grouped into fourteen resource groups, twenty-two move groups, and six equivalencies. There are forty explicit relationships (not including the co-located relationships from the resource groups).

The translation into the PSL modeling language and RuleBase backend processing provides us with a final model of about 250 variables. RuleBase solves any properties defined in the previous section in less than one minute with a BDD-based engine. Previously, the SAP policy was thoroughly tuned and tested; therefore we did not find bugs. However, we found the bugs we artificially inserted without difficulty. In addition, we were able to identify single points of failure of this policy.

7.2 Scalability

To test the scalability of our approach, we combined two large policies for TSA for Multi-Platform (SAP and STK policies) to create a very large policy. We made each of the SAP resource group dependents of the "goal" of the STK policy using the relevant relationships. An average TSA for Multi-Platform policy has about half the size of either SAP or STK policies, so the combined policies is roughly speaking four times the size of the average policy. The result was a policy whose translation into the PSL modeling layer and RuleBase backend processing provides us with a final model of about 460 variables. RuleBase was able to verify any of the previously defined properties in less than one minute with classical backward BDD-based model-checking. Therefore, we are confident that our approach can tackle most of the TSA for Multi-Platform policies, as SAP and STK policies are large by TSA standards.

For huge policies that would take too much time to process we can use abstractions in order to get a more manageable model. For instance, there are five different resource states; therefore, the state of every resource has to be coded on three bits (i.e., three state variables). In order to be able to code resource state on 2 bits, we can omit the FailedOffline and StuckOnline terminal states for checking most properties. For example, for the property `EF nominal_state`, if the desired states specified in the policy are non-terminal states (which is likely, since the terminal states are failure states), then if the property holds for the model with terminal states, it holds against the model without terminal states. This simple abstraction, allows for example to shrink the final model for the composition of SAP and STK policies by more than 20%.

8. First-order headings

We presented a model checking approach for policy verification using the specific case study of TSA. TSA relationships are translated in a straightforward manner into PSL properties and then checked with RuleBase. We conducted experiments on real-life industrial policies to validate our approach.

We plan to develop a fully automated verification solution for TSA for Multi-Platform. This will allow faster tuning and complete verification of TSA policies, and foster an increase in productivity. In addition, we are working on the modeling and formal verification of TSA for z/OS policies [20]. Contrary to expectations, this is not straightforward. TSA for z/OS has different syntax and more importantly, semantics, from the TSA for Multi-Platform, and typical TSA policies for z/OS encompass a number of variables that are one to two orders of magnitude larger than typical TSA MP policies. That said, the complexity of z/OS policies makes their verification even more critical.

Acknowledgments. The authors wish to thank Oliver Andersen, Enrico Joedecke, Markus Mueller, and Thomas Lumpp for their help and explanations about TSA for Multi-Platform, and Dana Fisman, Sharon Keidar-Barner, Avigail Orni and Sitvanit Ruah for helpful discussions.

9. References

1. Dakshi Agrawal, Seraphin Calo, James Giles, Kang-won Lee, and Dinesh Verma. Policy Management of Networked Systems and Applications. In *Proc. of Ninth IFIP/IEEE International Symposium on Integrated Network Management (IM 2005)*, IFIP/IEEE 2005.
2. E.M. Clarke and E.A. Emerson, "Design and synthesis of synchronization skeletons using Branching Time Temporal Logic". In *Proc. of Workshop on Logics of Programs, LNCS 131*, Springer, 1981.
3. E.A. Emerson and J.Y. Halpern, "'Sometimes' and 'Not Never' Revisited: On Branching versus Linear Time Temporal Logic", *Journal of the Association for Computing Machinery, Vol. 33, No. 1*, January 1986.
4. Cindy Eisner, Dana Fisman. *A Practical Introduction to PSL*. Springer, August 2005.
5. Sandeep K. Shukla and Rajesh K. Gupta. A Model Checking Approach to Evaluating System Level Dynamic Power Management Policies for Embedded Systems. *Sixth IEEE International High-Level Design Validation and Test Workshop (HLDVT'01)*, IEEE Computer Society Press, 2001.
6. S. Jha and T. Reps. Analysis of SPKI/SDSI certificates using model checking. In *Computer Security Foundations Workshop (CSFW)*, June 2002.
7. A. Pnueli. A Temporal Logic of Concurrent Programs. In *Theoretical Computer Science*, Vol 13, 1981.
8. A. Pnueli and R. Rosner. On the synthesis of a reactive module. *POPL '89: Proc. of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM Press, 1989.
9. Andreas Schaad, Volkmar Lotz, and Karsten Sohr. A model-checking approach to analysing organisational controls in a loan origination process. *Proc. of the Eleventh ACM Symposium on Access Control Models and Technologies*, 2006.
10. Carsten Sinz, Thomas Lumpp, Jürgen M. Schneider, and Wolfgang Küchlin. Detection of dynamic execution errors in IBM system automation's rule-based expert system. *Information & Software Technology 44(14),* 2002.
11. Carsten Sinz, Wolfgang Küchlin, and Thomas Lumpp: Towards a Verification of the Rule-Based Expert System of the IBM SA for OS/390 Automation Manager. *Asia-Pacific Conference on Quality Software, APAQS IEEE Computer Society 2001.*
12. Volker Schölles. *Setup and Policy to make a SAP system with J2EE Application server (like XI or EP) highly available with TSA MP*, 2006.
13. R. S. Street. Propositional Dynamic Logic of Looping and Converse is Elementary Decidable. *Information and control, 54(1/2)*, 1982.
14. Moshe Y. Vardi. An Automata-Theoretic Approach to Fair Realizability and Synthesis. *In Proc. Of Computer Aided Verification, 7th International Conference (CAV), LNCS 939*, Springer, 1995.
15. S. Wright, R. Chadha, and G. Lapiotis, (eds): *Special issue on Policy based Networking*, *IEEE Networking 16*, 2002.
16. Emmanuel Zarpas. A Case Study: Formal Verification of Processor Critical Properties, *Correct Hardware Design and Verification Methods: CHARME 2005, LNCS 3725*, Springer 2005.
17. Nan Zhang, Mark D. Ryan and Dimitar Guelev, Evaluating Access Control Policies Through Model Checking. *Eighth Information Security Conference (ISC'05). LNCS 3650*, Springer, 2005.
18. *IBM Tivoli System Automation for Multi-platforms, Guide and Reference, version 1.2,* IBM, 2004.
19. *IBM Tivoli System Automation for Multi-platforms, Base Component Reference, version 2.1.1*, 2006.
20. *Tivoli System Automation for z/OS, Defining Automation Policy*, July 2006.
21. *IEEE Standard for Property Specification Language (PSL) IEEE Std. 1850-2005*, 2005.
22. *RuleBase User Parallel Edition, User Manual*, June 2006.

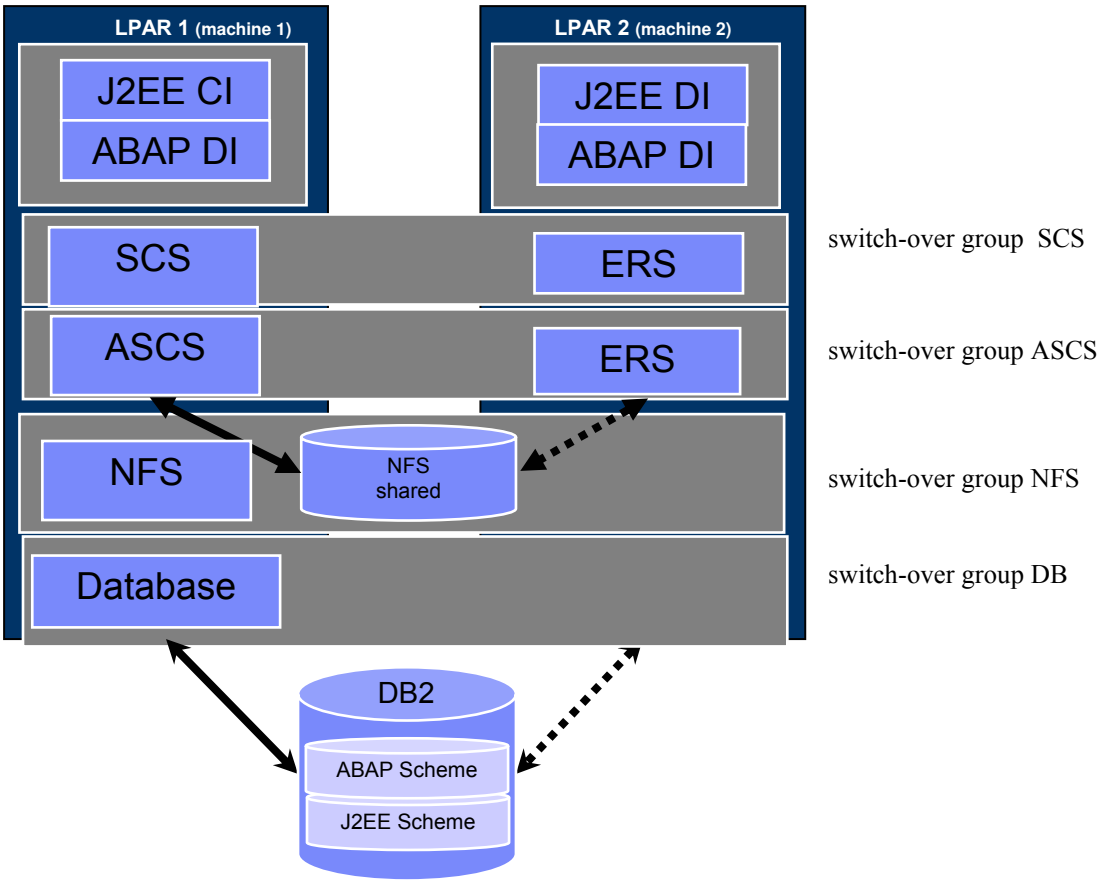

Figure 4 Two node domain setup for SAP system