Towards a Verification of the Rule-Based Expert System of the IBM SA for OS/390 Automation Manager

Carsten Sinz and Wolfgang Küchlin
University of Tübingen, WSI
Symbolic Computation Group
72076 Tübingen, Germany
http://www-sr.informatik.uni-tuebingen.de

Thomas Lumpp
IBM Germany Development
zSeries System Management
71032 Böblingen, Germany
Thomas.Lumpp@de.ibm.com

Abstract

We formally verify consistency aspects of the rule-based expert system of IBM's System Automation software for IBM's e-server zSeries. Starting with a formalization of the expert system in propositional dynamic logic (PDL), we are able to encode termination and determinism properties. To circumvent direct proofs in PDL or its extension ΔPDL, we further translate versions of the occurring decision problems to propositional logic, where we can apply advanced SAT and BDD techniques. In our experiments we could reveal some inconsistencies and, after correcting them, we successfully verified a non-looping property for a part of the expert system.

1. Introduction

The use of knowledge bases as components within safety or business critical systems has become more and more widespread during the 90s [And92], and has attracted renewed attention in agent-based intelligent web applications [GB00]. A very common technique to store knowledge in these systems is via rules. This form of expressing knowledge has—amongst others—the advantage that it employs a representation that resembles the way experts tend to express most of their problem solving techniques, namely by situation-action rules [HR85]. A precise semantics of the rules and a suitable verification methodology is highly desirable, as there is a strong potential for errors during the generation and maintenance of rules [NK91]. There is, however, no generally accepted formalism for the verification of rule-based systems, so a lot of different techniques have been proposed [NPLP87, AT92, RSC97], and verification of real-world industrial applications has remained rare.

In our paper, we investigate the rule-based expert system of IBM's System Automation (SA) solution for OS/390.[1] This system is used by major companies of practically all industrial sectors to automate the operation of high-availability applications on their S/390 and zSeries mainframe computers. Moreover, IBM's zSeries e-servers are becoming increasingly important as UNIX/Linux superservers for e-business applications.

Our main goal is the detection of *infinite computations* (or *loops*) in the rule-based central control instance of SA called *Automation Manager*. The presence of such infinite computations which are caused by erroneous rules may lead the Automation Manager to false decisions, or to oscillate between different computation states, disabling the overall functionality of SA for the mainframe or even for the whole cluster of mainframes (*Parallel Sysplex*).

The verification approach we take consists of the following steps: Starting with the necessary formal description of the rule-system, for which we have chosen propositional dynamic logic, we encode some consistency criteria in an extension of this logic, ΔPDL. This leaves us with proof obligations for either a ΔPDL model checker or theorem prover. As we expected our proof tasks to reach or exceed the limit of such provers, we have chosen another approach by translating our problems—or partially restricted versions of our problems—to propositional logic and then applying state-of-the-art SAT-checkers [Zha97] and BDD implementations [Som98] that have already shown their success in neighboring fields. While this step could theoretically be considered as the final one in a verification, we want to stress that in practice this was not the case. Lots of errors were falsely reported, due to an incompletely specified rule system. Implicit assumptions on possible computation states thus had to be made explicit to allow separation of genuine and spurious errors.

After having taken all these steps on a development re-

[1]For more information on SA see `http://www.s390.ibm.com/products/sa/v21info.html`

lease of SA close to shipment, we could actually find some residual errors that had remained even after months of professional testing. All of our detected deficiencies were confirmed by simulation on a zSeries test system, and could be eliminated.

2. IBM's System Automation for OS/390

Mission critical computer systems have to be up and running reliably. Often these systems are engaged with complex application environments, and thus demand high skills and considerable knowledge from the operating personnel in the computer centers. Computer failures can provoke considerable financial losses. For instance, a one hour down time period in a computer center of a bank can easily cause costs of up to ten million dollars. In these highly critical environments, IBM's zSeries e-servers are frequently used to provide extremely high availability.

The basic idea behind SA is to fully automate a computer center and to increase the availability and reliability of business applications. It allows to define complex software environments in which applications are automatically started, stopped, and supervised during runtime. In the case of an application or system failure, SA can react immediately and solve the problem by restarting the application or, if this is not possible any more, restart it on another system in the cluster. SA provides functionality like *grouping* which allows to automate a collection of applications as one logical unit. Furthermore, *dependency management* allows the definition of start and stop relationships, for example start A before B. Both grouping and dependency management are provided for an entire cluster of S/390 or zSeries mainframes. Of course, SA provides further functionality.

For a better understanding of SA, let us consider a simplified flight reservation system that can be used by hundreds of users in parallel. Such an application consists of various functional components: a database that stores information about flight schedules, reservations, and billing; a transaction management system which guarantees overall data consistency; a network infrastructure with different protocol stacks and firewalls; a web server environment for the user interface; and possibly further system dependent components. To model this application in SA, we define a top level group "flight reservation" which has each of the functional components as a member. Since the functional components themselves consist of various applications these are also each defined as groups.

Besides, most of the applications have start and stop dependencies to other applications. In our example, parts of the transaction management system may depend on the underlying database to work properly. With SA, one can define that a start of the transaction management is only performed when the required database is running (*start dependency*). Therefore SA would start the database first in order to start the transaction management system. Starting the database, however, may in turn depend on system specific applications having been started before. Similar relations can hold when stopping applications (*stop dependency*). For example, it should not occur that the database is stopped before the transaction system is brought down. So moving an application with complicated start and stop dependencies from one system to another (for example in case of a malfunction) can be quite an elaborate task. Moreover, applications that can or should not work simultaneously on the same system can generate conflicting requirements.

2.1 Outline of the SA for OS/390 Software Architecture

The SA software consists of two logical parts: a manager and an agent. There is only one active manager, but additional backup managers can be defined. Agents are located on each system in the cluster. The manager acts as a Sysplex-wide decision maker. It determines for all applications when they have to be started or stopped, and observes their activity. When making its decisions the manager takes grouping and dependency restrictions into account.

The manager does not perform application starts and stops by itself. Instead, it sends a start or stop command to the agent on the system where the application is located. This agent receives the order and does the actual start- or stop-processing. Furthermore the agent informs the manager about the current status of all applications on the system where it is running. Thus the manager possesses information about the complete status of the cluster. The manager does its internal processing based on abstract (proxy) resources. The implementation of the manager is done in a generic way: Most parts of the manager are implemented as a rule-based expert system. For example, the start and stop dependency behavior is defined by a collection of specific rules in the expert system. In the rest of this paper we will focus on this rule-based expert system.

2.2 The Automation Manager's Rule-Based Expert System

The Automation Manager is implemented as a rule-based expert system. Its core consists of an automation engine which besides rule execution can interpret a few hundred different instructions which are used to create and manipulate abstract resources within the manager. For each definition of an automation entity (application, group, etc.) an abstract *resource* is generated by the manager. Groups of rules are then associated with each resource. These rules, called *correlation rules*, are taken from a basic rule set of a couple hundred of rules contained in SA, and their variables

```
correlation set/status/compound/satisfactory :
when        status/compound NOT E {Satisfactory}
        AND status/startable E {Yes}
        AND ( ( status/observed E {Available, WasAvailable}
                AND status/desired E {Available}
                AND status/automation E {Idle, Internal}
                AND correlation/external/stop/failed E {false}
              )
              OR
              ( status/observed E {SoftDown, StandBy}
                AND status/desired E {Unavailable}
                AND status/automation E {Idle, Internal}
              )
            )
then    SetVariable status/compound = Satisfactory
        RecordVariableHistory status/compound
```

Figure 1. Example of a Correlation Rule

are instantiated for their resource. All correlation rules are of the form

correlation `<name>:`
when `<formula>`
then `<action list>`

where `formula` is a finite domain formula with atomic propositions of the form

`<var> E { <val`$_1$`>, ... ,<val`$_n$`> }`
`<var> NOT E { <val`$_1$`>, ... ,<val`$_n$`> }`

and the usual Boolean connectives `AND`, `OR` and `NOT`. Variable names may contain alpha-numerical characters and the slash. `E` denotes set membership. The only actions in the **then**-part we are interested in are assignment statements of the form `SetVariable <var> = <val`$_i$`>`. Other actions in the SA system are mainly used for event logging and to present messages to the user. Only one `SetVariable`-action is present in each rule's action list. Figure 1 shows a typical correlation rule.

To compute, for example, the compound state of a resource, rules are evaluated according to the following scheme: As soon as a variable changes its value, the automation manager re-evaluates the rules to reflect this change: the rules are tested one by one, whether the formula of the rule's **when**-part evaluates to true under the current variable assignment. If this is the case, the action part is executed, which may result in further variable changes. The whole process stops when no more variable changes occur. The order in which rules are evaluated is not specified—with the exception of a fairness condition stating that no rules are missed out.

Changes on the variables' values may occur for two reasons: (i) by a "spontaneous" change of volatile (transient, observed) external variables not controlled by the correlation rule system or (ii) by execution of `SetVariable`-actions in the **then**-part of a rule. We therefore partition the set V of variables contained in the correlation rules into two disjoint sets: a set of computed state variables V_S and a set of observed external variables V_O, such that $V = V_S \uplus V_O$. V_S comprises exactly those variables that occur in a rule's action part, i.e. variables that may be changed by rule execution. The values of externally controlled, observed variables are delivered to the rule system either by the resource's automation agent or by the central Automation Manager itself.

3. Formalization of Correlation Rules and Consistency Properties

For a formalization of the correlation rules and the computation done by the Automation Manager, we have selected PDL. There are many reasons for our choice. First, correlation rules can easily be translated to PDL, and the resulting formulae are quite comprehensible. Furthermore, the employed rule-based computation contains an indeterminism in that the exact order of rule evaluation is not specified; PDL allows easy formulation of and reasoning about indeterministic programs. Communication between resources is not the key issue here, so the formalization language need not reflect this aspect. For the specification of the correlation rules we only need constructs from PDL, whereas formalization of the termination property of the Automation Manager requires an extension of ordinary propositional dynamic logic. We employ ΔPDL, which adds a divergence operator Δ to PDL to enable the notion of infinite computation. ΔPDL was introduced by Streett

[Str82], a similar extension can be found in the work of Harel and Sherman [HS82].

PDL allows reasoning about programs (denoted by $\alpha, \beta, \dots$) and their properties, and thus contains language constructs for programs as well as for propositional formulae. Atomic propositions ($P, Q, R, \dots$) can be combined to compound PDL formulae ($F, G, \dots$) using the Boolean connectives $\neg$, $\vee$ and $\wedge$, composite programs are composed out of atomic programs using three different connectives: $\alpha;\beta$ denotes program sequencing, $\alpha \cup \beta$ nondeterministic choice, and α^* a finite, nondeterministic number of repetitions of program α. For a formula F the program $F?$ denotes the test for property F, i.e. $F?$ proceeds if F is true, and otherwise fails. The modal formulae $[\alpha]F$ and $\langle\alpha\rangle F$ have the informal meaning "all terminating executions of program α lead to a situation in which F holds" respectively "there is a (terminating) program run of α after which F is true". ΔPDL adds the construct $\Delta\alpha$ to the language expressing that the program α^* can diverge, i. e. enter a non-halting computation. We refer the reader to Harel's introductory chapter on PDL [Har84] for a thorough elaboration.

3.1 Encoding of the Correlation Rules and the Status Computation

Encoding of correlation rules and formalization of the Automation Manager program is accomplished in four steps: First, we encode the variable's finite domains in Boolean logic; then we translate the rule's actions and their semantics to PDL; afterwards we are able to give PDL encodings of complete correlation rules; and finally we give a formal description of program executions of the rule-based Automation Manager.

3.1.1 Finite Domains

Each variable v occurring in a correlation rule can take a value of a finite domain D_v depending on the variable. For our PDL encoding, we first need to decompose the finite domains into Boolean propositions. We therefore introduce new propositional variables $P_{v,d}$ for each possible value $d \in D_v$ of each variable v, expressing the fact that variable v takes value d. We then need additional restrictions, expressing that each finite domain variable takes exactly one of its possible values. Supposing a set V of correlation rule variables, we thus get an additional propositional restriction R_V:

$$\bigwedge_{v \in V} \left(\bigvee_{d \in D_v} P_{v,d} \;\wedge \bigwedge_{\substack{d_1, d_2 \in D_v \\ d_1 \neq d_2}} \neg(P_{v,d_1} \wedge P_{v,d_2}) \right)$$

Formulae similar to R_V also occur in the context of propositional encoding of planning problems, where they are referred to as *linear encodings* [KMS96].

3.1.2 Atomic Programs

The atomic programs of our formalization are assignment programs, denoted by $\alpha_{v,d}$, where $\alpha_{v,d}$ assigns value $d \in D_v$ to variable $v \in V_S$. Each assignment program is, of course, deterministic and after its execution the variable has the indicated value. Other computed variables are not affected. Therefore the following PDL properties hold for each program $\alpha_{v,d}$ and all propositions p:

1. $[\alpha_{v,d}]p \Leftrightarrow \langle\alpha_{v,d}\rangle p$
2. $[\alpha_{v,d}]P_{v,d}$
3. $P_{w,d'} \Rightarrow [\alpha_{v,d}]P_{w,d'}$ for all $w \in V_S, w \neq v$ and $d' \in D_w$.

We will denote the conjunction of these propositions for all atomic programs by R_α.

3.1.3 Correlation Rules

In the following we assume for each variable-value pair (v, d) at most one rule with an action setting variable v to d in its **then**-part. If this is not the case, the **when**-parts of rules with common actions can be merged disjunctively. To encode a correlation rule, its **when**-part is recursively translated into a Boolean logic formula using transformation τ, which is defined for the base case by

$$\begin{aligned} \tau(v \; \mathtt{E} \; \{d_0, \dots, d_j\}) &= P_{v,d_0} \vee \dots \vee P_{v,d_j} \\ \tau(v \; \mathtt{NOT} \; \mathtt{E} \; \{d_0, \dots, d_j\}) &= \neg P_{v,d_0} \wedge \dots \wedge \neg P_{v,d_j} \quad , \end{aligned}$$

and extended to complex formulae in the obvious way. For the **then**-part we only have to consider actions setting variables, which are translated by τ to their corresponding atomic PDL programs:

$$\tau(\mathtt{SetVariable} \; v = d) = \alpha_{v,d} \quad .$$

Given a rule's translated **when**-part $F_{v,d}$ and its translated **then**-part $\alpha_{v,d}$, we get as PDL program $R_{v,d}$ for that rule:

$$R_{v,d} \; := \; (F_{v,d} \wedge \neg P_{v,d})?; \alpha_{v,d} \quad ,$$

expressing that the action of the **then**-part is executed, provided the **when**-part holds and the variable is not already set to the intended value. The additional restriction $\neg P_{v,d}$ prevents rule executions that do not produce any change of variable values.

3.1.4 Automation Manager

We are now able to formally specify the computations performed by the Automation Manager program. As there is no rule evaluation order, the program just selects any rule, evaluates its formula, executes the action part and starts over again. The single-step Automation Manager program S and the Automation Manager program AM therefore look like this:

$$\begin{aligned} \mathrm{S} &= \bigcup_{v \in V_S, d \in D_v} R_{v,d} \\ \mathrm{AM} &= \mathrm{S}^*; \Big(\bigwedge_{v \in V_S, d \in D_v} (F_{v,d} \Rightarrow P_{v,d}) \Big)? \end{aligned}$$

For each SA resource a program of the above kind is generated. Each Automation Manager program runs until no further rules can be applied (reflected by the last test in the Automation Manager program AM), and is restarted as soon as an observed external variable $v_o \in V_O$ changes its value.

3.2 Consistency Properties of the Correlation Rule System

The computation relation generated by the correlation rules should be functional and terminating. For example, a status computation should not result in different values depending on the exact order of rule application, and it should produce a result in a finite number of computation steps. Moreover, we are faced with the situation that there are external variables (observation variables) that may change their values during computation. Thus, to produce sensible statuses, short computations are needed, and observed external variables should not change frequently. For our consistency properties we therefore assume all external observed variables to be fixed.

We now turn to the formalization of the two consistency criteria termination and functionality. As above, we denote by AM, respectively S, the part of the Automation Manager program that deals with full, respectively single step, computations. In the following, formula PRE encodes common preconditions for all consistency criteria. This includes the finite domain restrictions R_V, the atomic program specifications R_α, and the fixing of all observation variables during computation. We therefore define

$$\mathrm{PRE} := R_V \wedge R_\alpha \wedge \bigwedge_{\substack{v \in V_O, w \in V_S \\ d \in D_v, d' \in D_w}} (P_{v,d} \Rightarrow [\alpha_{w,d'}] P_{v,d}) \ .$$

The following ΔPDL formula, provided it is valid, guarantees that there is no divergent computation: [2]

$$\mathrm{PRE} \Rightarrow \neg \Delta \mathrm{S} \ . \qquad (1)$$

[2]Note that this property cannot be expressed in ordinary PDL [Str82].

To ensure functionality for a computation starting in some state, we need a final result that is unique. So, if there is a terminating computation sequence of the Automation Manager all other computations have to end in the same state:

$$\mathrm{PRE} \Rightarrow \Big(\langle \mathrm{AM} \rangle p \Leftrightarrow [\mathrm{AM}] p \Big) \ . \qquad (2)$$

There are other consistency criteria (for example confluence) that could be checked, too. We will not elaborate on this, but instead concentrate on the termination property in the following.

As termination is defined as the absence of an infinite sequence of consecutive computation states ($\neg \Delta \alpha$), we have to fix more precisely the notion of a state. A *state* s is an assignment to the propositional variables, i.e. a function $V \to \{0, 1\}$. A state s is said to be *proper* if it correctly reflects the finite domain restriction R_V, i.e. if s is a model of R_V, or, equivalently in symbols, $s \models R_V$. A pair of states (s_0, s_1) is called an $\alpha_{v,d}$*-transition* (denoted by $s_0 \xrightarrow{v=d} s_1$) if execution of program $\alpha_{v,d}$ leads from s_0 to s_1, i.e. s_0 and s_1 are proper states, with s_0 and s_1 differing only on the set $\{P_{v,d'} \mid d' \in D_v\}$, and $s_1(P_{v,d}) = 1$.

As the number of states is finite, all non-terminating computations are caused by loops in the program transition graph. For example, the 2-loop

$$s_0 \xrightarrow{v=d_1} s_1 \xrightarrow{v=d_0} s_0 \qquad (3)$$

generates an infinite computation oscillating between the states s_0 and s_1. As another example consider the 4-loop

$$s_{00} \xrightarrow{v_0=d_1^0} s_{01} \xrightarrow{v_1=d_1^1} s_{11} \xrightarrow{v_0=d_0^0} s_{10} \xrightarrow{v_1=d_0^1} s_{00}$$

that is not decomposable into two simpler 2-loops. Showing termination of the Automation Manager program can thus be accomplished by proving the absence of n-loops for all $n \geq 2$. Note, that the case $n = 2$ in particular covers those situations where the loops are due to an overlap of the **when**-parts of two rules for the same variable, i.e. when $s_0, s_1 \models F_{v,d_0} \wedge F_{v,d_1}$. It is thus of special importance.

To prove the non-existence of loops—as well as the other consistency criteria—directly within the ΔPDL formalism, we can in principle distinguish two main approaches: either by model checking or by theorem proving. For the first approach, a Kripke structure has to be created based on the elementary properties R_α of the atomic assignment programs and on the validity of the propositions $P_{v,d}$, considering the restrictions R_V. This step builds a structure that fulfills the general precondition PRE. Then it is checked, whether or not the ΔPDL consistency criteria (without preconditions) are fulfilled in the generated model. In the theorem proving formalism, we try to derive the consistency criteria directly from the preconditions.

We have chosen yet another way, which translates the PDL proof obligations into purely propositional logic formulae. We thus enable the application of advanced propositional SAT-checkers.

3.3 Conversion to Propositional Satisfiability

Conversion to a purely propositional formalism requires handling of different states within one formula. We use restrictions to achieve this goal.

The *proper restriction* $F|_{v=d}$ of a propositional formula F is defined as the homomorphic extension of the function

$$P_{v',d'}|_{v=d} = \begin{cases} \top & \text{if } v = v', d = d', \\ \bot & \text{if } v = v', d \neq d', \\ P_{v',d'} & \text{if } v \neq v'. \end{cases}$$

The following lemma is easily shown and allows the formulation of propositional properties concerning multiple computation states.

Lemma 3.1 *Let* $s_0 \xrightarrow{v=d} s_1$. *Then* $s_1 \models F$ *iff* $s_0 \models F|_{v=d}$.

At first, we want to consider 2-loops. All 2-loops are of the form $s_0 \xrightarrow{v=d_1} s_1 \xrightarrow{v=d_0} s_0$. Thus, the absence of 2-loops is expressed in accordance with (1) by

$$\text{PRE} \;\Rightarrow\; \neg\exists s_0, s_1, v, d_0, d_1 (s_0 \xrightarrow{v=d_1} s_1 \xrightarrow{v=d_0} s_0). \quad (4)$$

The two $\alpha_{v,d_{0/1}}$-transitions can be performed provided the following holds (by definition of α-transitions and the Automation Manager program):

$$s_0, s_1 \models R_V \qquad s_1 \models P_{v,d_1} \qquad s_0 \models P_{v,d_0}$$

$$s_0 \models F_{v,d_1} \wedge \neg P_{v,d_1} \qquad s_1 \models F_{v,d_0} \wedge \neg P_{v,d_0}$$

According to Lemma 3.1 this is equivalent to

$$s_0 \models R_V \wedge P_{v,d_0} \wedge F_{v,d_1} \wedge \neg P_{v,d_1} \wedge (R_V \wedge F_{v,d_0})|_{v=d_1},$$

which can be further simplified to

$$s_0 \models R_V \wedge P_{v,d_0} \wedge F_{v,d_1} \wedge F_{v,d_0}|_{v=d_1}.$$

As a propositional formula for the absence of 2-loops we therefore get from (4) by dropping the now superfluous semantics of atomic programs from PRE and after further simplification:

$$R_V \;\Rightarrow\; \neg\big(P_{v,d_0} \wedge F_{v,d_1} \wedge F_{v,d_0}|_{v=d_1}\big), \quad (5)$$

which has to be valid for all v, d_0, and d_1. Similarly, the absence of 3-loops is reflected by the validity of

$$R_V \;\Rightarrow\; \neg\big(P_{v,d_0} \wedge F_{v,d_1} \wedge F_{v,d_2}|_{v=d_1} \wedge F_{v,d_0}|_{v=d_2}\big)$$

for all v, d_0, d_1, and d_2. The extension to n-loops involving only one variable v is obvious. The general case of n-loops is more complicated, however, due to different types of loops involving the modification of multiple finite domain variables. This general case is not considered here.

Not all loops detected by this method really do occur. For example, in the formula above, s_0 has to be a reachable state of the computation. Moreover, rule evaluation in the Automation Manager program is subject to a fairness condition, stating that a rule is eventually executed provided its **when**-part is satisfied.[3] We modeled the fairness condition by considering only starting states s_0 in which all non-affected variables are already computed. I.e. we demand for a loop involving variable v that all rules containing other variables $w \neq v$ in their action part cannot be executed.

For our experiments we therefore checked the extended formula E instead of Formula (5) where E is defined as

$$R_V \;\wedge \bigwedge_{\substack{w \in V_S, w \neq v \\ d_w \in D_w}} (F_{w,d_w} \Rightarrow P_{w,d_w}) \quad \Rightarrow$$
$$\neg\big(P_{v,d_0} \wedge F_{v,d_1} \wedge F_{v,d_0}|_{v=d_1}\big) \;. \quad (6)$$

4. Verification Techniques

We will now describe the techniques we used to prove the propositional formulae of the last section. We also show how the counter-models that appear in case a proposition could not be proved can be made more intelligible.

Davis-Putnam-Style Prover

We used a Davis-Putnam-style prover to show the unsatisfiability of the negations of the aforementioned formulae. The prover is our own implementation (see [Kai00] for a more detailed description) which, in contrast to other DP implementations, does not require the input to be in conjunctive normal form. Moreover, it allows the direct specification of *n-out-of-m*-constructs that frequently occur in practical applications, in our case in the translation of the finite domain restriction R_V to Boolean logic. So instead of formula R_V which is quadratic in the domain sizes D_v, we just have to deal with a linear-size formula in our extended language.

BDD-based Approach

We also experimented with a conversion of the formulae to binary decision diagrams (BDDs), where we used Somenzi's CUDD package [Som98]. One reason to use

[3]The actual condition in the Automation Manager is even stronger in that rules are cyclically checked for execution.

BDDs was to get a manageable representation of all counter-models in case the propositions did not hold.

To further simplify the counter-model representation we applied an existential abstraction over variables not occurring in the currently investigated rules, i.e. we retained only variables occurring in F_{v,d_0} or F_{v,d_1} of Formula (6) in the counter-models and removed all others. More formally, we generated a quantified Boolean formula $\exists \vec{X} F$ where F is Formula (6) and $\vec{X}$ contains exactly those variables not appearing in F_{v,d_0} and F_{v,d_1}, i.e. $\vec{X} = V \setminus (\text{Var}(F_{v,d_0}) \cup \text{Var}(F_{v,d_1}))$.

Implicit Assumptions on Observation Variables

Not all combinations of possible values for observation variables do really occur. But which of them are possible and which are not is not laid down in the Automation Manager's expert system. For our verification task we thus added some further restrictions to the set R_V reflecting cases that do not occur in practice. These cases were specified by SA experts from IBM after an investigation of the counter-models.

5 Experimental Results and Experiences

We conducted experiments with a subset of the rules of the Automation Manager and exemplarily investigated the 41 rules for the compound status computation. The compound status indicates the overall status of a resource, depending on its automation goal, the actual state and on other resources' states. It can take any of seven different values, so we had to perform 21 proofs of Formula (6) to show the absence of 2-loops. Instead of proving these formulae directly, we tested their negations for unsatisfiability.

We used our DP-style prover implementation to check the generated formulae. As our implementation allows special *select-n-out-of-m*-constructs [Kai00], formula sizes could be kept small. Proofs or counter-models for all formulae were found in under a second. Initially, seven of the 21 instances were satisfiable, each indicating a possible non-terminating computation. However, further examination showed that most of these cases cannot occur in practice. The reason for these false error readings lies in an incomplete formalization in the rule system. E. g., implicit assumptions on which states are reachable have to be made explicit in order to achieve practically usable results. Thus, we added the abovementioned further restrictions on observation variables, which brought down the number of inconsistencies to three. For these three cases we generated BDDs of the 2-loop-formulae. The times to build up the BDDs again were under a second. Whereas SAT-checkers can only deliver counter-models of the termination property, BDDs allow a direct representation of the condition under which the loop occurs and thus enable further processing of the result. We made use of the BDD representation by applying existential abstraction to variables not occurring in the rules' **when**-parts. This helped greatly to find out in which situations a loop occurs and thus facilitated correction of the rules.

All of our detected 2-loops were reproduced by emulation on a zSeries test system and resulted in a modification of the final product. The final version thus contained no 2-loop-errors. So, by identifying real defects, we could further increase the reliability of the Automation Manager.

6 Conclusion, Related and Future Work

By formalizing the SA Automation Manager's rule-based expert system we could prove a restricted non-looping property for a part of the rule system. After encoding the rules and consistency properties in ΔPDL and converting them to SAT, our approach led us to a set of propositional properties that current SAT-checking techniques can easily handle. We also consider it an important observation that in practice rule systems may be incompletely specified and that formalization requires to make implicit assumptions explicit in order to avoid meaningless results.

Spreeuwenberg *et al.* present a tool to verify knowledge bases built with Computer Associate's Aion system [GB00]. They also treat real-life applications, for example for the Postbank Nederland BV's assessment knowledge base [SGB00]. Representative of many other similar projects, we want to mention Hörl and Aichernig [HA99] who formalized and verified a set of test cases for an air traffic voice communication system.

As an interesting task for the future we see an integrated verification approach for both the high-level dependency conditions on resources and the low-level Automation Manager's rule-system. As the high-level conditions can be edited by SA users, verification cannot remain a step in the product development cycle, but becomes part of the users' administration work, with all the induced demands this entails on the verification process such as user-friendliness or fully automatic proofs.

References

[And92] E. Andert. Automated knowledge-base validation. In *Proceedings of the AAAI Workshop on Verification and Validation of Expert Systems*, pages 122–127, July 1992.

[AT92] R. Agarwal and M. Tanniru. A Petri-net based approach for verifying the integrity of production systems. *Int'l J. Man-Machine Studies*, 36:447–468, 1992.

[GB00] S. Garone and N. Buck. *Capturing, Reusing, and Applying Knowledge for Competitive Advantage: Computer Associate's Aion*. International Data Corporation, 2000. IDC White Paper.

[HA99] J. Hörl and B. K. Aichernig. Formal specification of a voice communication system used in air traffic control: An industrial application of light-weight formal methods using VDM^{++}. In *FM'99 - Formal Methods, Vol. II*, volume 1249 of *Lecture Notes in Computer Science*, pages 1868–1868. Springer, 1999.

[Har84] D. Harel. Dynamic logic. In D. Gabbay and F. Guenthner, editors, *Handbook of Philosophical Logic*, volume II: Extensions of Classical Logic, pages 507–544. Kluwer, 1984.

[HR85] F. Hayes-Roth. Rule based systems. *Comm. ACM*, 28(9):921–932, 1985.

[HS82] D. Harel and R. Sherman. Looping vs. repeating in dynamic logic. *Information and Control*, 55(1-3):175–192, 1982.

[Kai00] A. Kaiser. A SAT-based propositional prover for consistency checking of automotive product data, 2000. Unpublished manuscript.

[KMS96] H. Kautz, D. McAllester, and B. Selman. Encoding plans in propositional logic. In *Proc. Fifth International Conference on Principles of Knowledge Representation and Reasoning (KR'96)*, pages 374–384, Cambridge, MA, November 1996. Morgan Kaufmann.

[NK91] D. L. Nazareth and M. H. Kennedy. Verification of rule-based knowledge using directed graphs. *Knowledge Acquisition*, 3:339–360, 1991.

[NPLP87] T. A. Nguyen, W. A. Perkins, T. J. Laffey, and D. Pecora. Knowledge-base verification. *AI Magazine*, 8(2):69–75, 1987.

[RSC97] M. Ramaswamy, S. Sarkar, and Y.-S. Chen. Using directed hypergraphs to verify rule-based expert systems. *IEEE Transactions on Knowledge and Data Engineering*, 9(2), 1997.

[SGB00] S. Spreeuwenberg, R. Gerrits, and M. Boekenoogen. VALENS: A Knowledge Based Tool to Validate and Verify an Aion Knowledge Base. In *ECAI 2000, 14th European Conference on Artificial Intelligence*, pages 731–735. IOS Press, 2000.

[Som98] F. Somenzi. *CUDD: CU Decision Diagram Package, Release 2.3.0*. University of Colorado, Boulder, 1998. Available at http://vlsi.colorado.edu/˜fabio.

[Str82] R. S. Streett. Propositional dynamic logic of looping and converse is elementarily decidable. *Information and Control*, 54(1/2):121–141, 1982.

[Zha97] H. Zhang. SATO: An efficient propositional prover. In *CADE'97: 14th International Conference on Automated Deduction*, volume 1249 of *Lecture Notes in Computer Science*. Springer Verlag, 1997.