ELSEVIER

Information and Software Technology 44 (2002) 857–873

INFORMATION AND SOFTWARE TECHNOLOGY

www.elsevier.com/locate/infsof

Detection of dynamic execution errors in IBM system automation's rule-based expert system☆

Carsten Sinz[a,*], Thomas Lumpp[b], Jürgen Schneider[b], Wolfgang Küchlin[a]

[a] *Symbolic Computation Group, Department of Computer Science, University of Tübingen, 72076 Tübingen, Germany*
[b] *IBM Germany Development, eServer System Management, 71032 Böblingen, Germany*

Abstract

We formally verify aspects of the rule-based expert system of IBM's system automation software for IBM's zSeries mainframes. Starting with a formalization of the expert system in propositional dynamic logic (PDL), we encode termination and determinism properties in PDL and its extension ΔPDL. We then translate our decision problems to propositional logic and apply advanced SAT techniques for automated proofs. In order to locate real program bugs for each failed proof attempt, we apply extra formalization steps and represent propositional error formulae in concise normal form as binary decision diagrams. In our experiments, we revealed residual non-termination bugs in a tested program version close to shipment, and, after correcting them, we formally verified the absence of this class of bugs in the production code. © 2002 Elsevier Science B.V. All rights reserved.

Keywords: Formal methods; Expert systems; Verification; Validation; Propositional dynamic logic; SAT checking; IBM system automation

1. Introduction

The use of knowledge bases as components within safety or business critical systems has become more and more widespread during the 1990s [2], and has attracted renewed attention in agent-based intelligent Web applications [9]. 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 [14].

However, there is some potential for errors during the generation and maintenance of the rules [24]. For example, rule systems lack common structuring elements such as those of object oriented languages, and they fall outside common programming technology. On the other hand their simplicity and level of abstraction facilitates formal verification. There is, however, no generally accepted formalism for the verification of rule-based systems, so many different techniques have been proposed [1,23,25,28], and the verification of real-world industrial applications is still 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.[2] IBM mainframes are typically employed in clusters, called *Parallel Sysplex*, for enhanced reliability. SA's technology is intended to be adapted to further platforms in near future.

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 faulty 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 entire Parallel Sysplex.

Common software engineering terminology distinguishes between *validation* and *verification* [34]. The former is concerned with meeting customer expectations in building the right system, the latter is concerned with building the system right, according to its specification.

☆ A preliminary version of this paper was presented at the Second Asia–Pacific Conference on Quality Software (APAQS 2001) [32].

* Corresponding author.

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

[2] zSeries and z/OS are the successors of the S/390 mainframes and their operating system OS/390, respectively.

PII: S0950-5849(02)00120-9

Formal verification of a program *P* involves proving, using mathematical arguments, that *P* is consistent with its (formal) specification. This can only work if the semantics of the programming language are formally defined, and the program is formally specified in a notation consistent with the verification techniques [34].

In our case, the rules are of a **when–then** form. The **when**-part consists of a formula of a finite domain propositional logic, and the **then**-part manipulates a global system state by setting the value of a finite domain variable. Our goal is to formally validate some consistency properties of a given rule set, mainly termination.

The verification approach we take consists of the following steps: Starting with the necessary formal description of the actions 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. We have chosen yet 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 [42] and BDD implementations [33] that have already shown their success in neighboring fields. The purpose of the interim PDL step is to help derive a correct formal model of the dynamics of the rule system and of the validation requirements.

In theory, the automated proofs could be considered the final step in a verification. In practice, the continuous discovery of errors during development is even more important than one final verification, so for each error an intelligible description is needed [6]. Initially, a number of non-genuine errors were reported, due to an incompletely specified rule system. Implicit assumptions on possible computation states thus had to be made explicit to allow the separation of genuine and spurious errors. Also, the Boolean formulas describing the error conditions had to be converted into human readable concise normal forms, using BDDs, in order to locate the errors in the source.

Working with a development release of SA close to shipment, we could actually locate some residual faults that had remained even after conventional professional testing and that had also survived all code reviews due to the complexity of the rules' **when**-parts. All of these deficiencies, which were detected through failed verification attempts, were subsequently confirmed by simulation on a zSeries test system, and could be eliminated prior to product roll-out. We then verified that the final product does not contain any more looping defects of this class.

The remainder of this paper is organized as follows. In Section 2, we give a description of IBM's system automation and of the form of its rules. In Section 3, we derive our formalization of the rule language; this is the theoretical core of the paper. In Section 4, we describe our verification techniques and tools. Section 5 contains our experimental results. In Section 6, we give a summary account of our industrial experience. Section 7 discusses related work, and the paper closes with a conclusion in Section 8.

2. IBM's system automation for OS/390

Mission critical computer systems have to be up and running reliably. Often these systems are employed in complex application environments, and thus demand high skills and considerable knowledge from the operating personnel in the computer centers. Computer failures, especially in the financial industry, can cause considerable losses. For instance, a one hour downtime period in a computer center of a bank can cause costs of up to ten million dollars [36]. In these highly critical environments, IBM's Parallel Sysplex clusters of zSeries mainframes are frequently employed to provide extremely high availability. The IBM zSeries provides an availability of 99.999% within a z/OS Parallel Sysplex environment, or less than 5.3 minutes downtime per year.

The basic idea behind IBM's SA is to fully automate a computer center and thus to reduce the complexity for the operators 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, if necessary 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, such as 'start A before B'. Both grouping and dependency management are provided across an entire Parallel Sysplex. Of course, SA provides further functionality beyond the scope of this paper.

As an example, let us consider a 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. In our example, parts of the transaction management system may depend on the underlying database to work properly. Hence, we have a *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

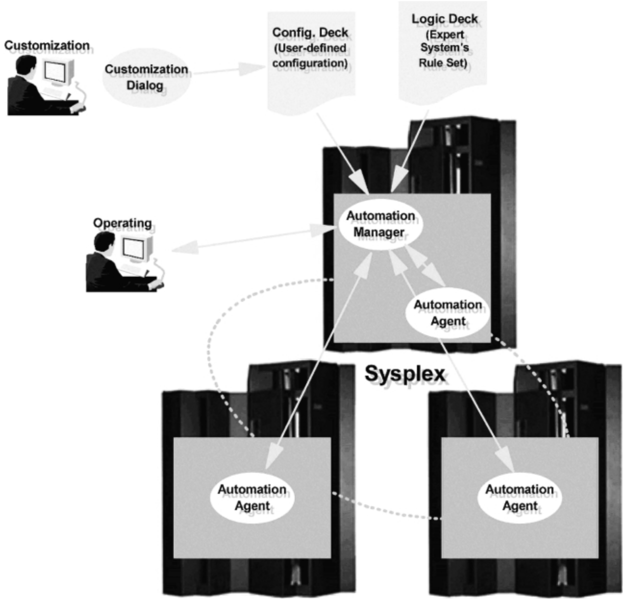

Fig. 1. General operation scheme of IBM SA.

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 in the cluster to another one (for example in case of a malfunction) can be quite an elaborate task. Moreover, applications that cannot or should not be collocated on the same system can generate conflicting requirements.

Taking into consideration that a customer setup can contain several thousand applications with a similar number of dependencies, it is clear that this cannot be controlled manually anymore.

2.1. Outline of the SA for OS/390 software architecture

IBM SA consists of two logical parts, *Automation Managers* (AM) and *Automation Agents* (AA). There is only one active Automation Manager at a time, the Primary Automation Manager (PAM). Additional Secondary Automation Managers (SAM) can be defined to prevent a single point of failure. These idle in the background, and one can take over in error situations with minimal interruption. The Automation Agents are located on each virtual system in a Parallel Sysplex cluster. Up to sixteen virtual systems with different operating systems may run on each physical system in the cluster.

The principles of the SA architecture are illustrated in Fig. 1. The Automation Manager is the central control instance and acts as a Sysplex-wide decision maker. It receives monitoring information from all automation agents consisting of application states, system state, and other data. Further input sources are a user-defined *Automation Policy* which describes the operating scenario with all its applications and their dependencies, and system operators who manually start and stop applications.

Based on monitoring information and the user-defined Automation Policy (see below), the Automation Manager possesses a complete picture of the status of the Sysplex. This enables the Automation Manager to derive a decision for each application whether it has to be started, stopped, or left in the state it is. The Automation Manager does not perform application starts and stops itself, but sends commands to the agent of the system on which the application currently is located. The agent receives the order and performs the actual start- or stop-processing. Furthermore, the agent delivers monitoring information to the manager.

To achieve a high level of availability, both the Manager–Agent communication and the Automation Manager's internal processing is implemented on top of IBM's transaction-based middleware MQSeries.

2.2. Automation Manager architecture

The Automation Manager receives monitoring information from the Automation Agents and uses this information to compute different status values; it sends commands, called *orders*, to the Automation Agents to control their assigned applications; and it contains the expert system controlling the Sysplex behavior. The automation manager internally represents real entities, like applications or systems, and virtual entities like application groups (aggregations of multiple applications) as abstract *resources*. As the expert system depends on the resources present in the cluster and their dependencies, it has to be user-adaptable to different scenarios.

The expert system representing the whole cluster is composed of a set of local expert systems for each resource. To model dependencies between resources, these expert systems communicate via special variables which have their values automatically exchanged by the Automation Manager. The rule set of each local expert system is composed of a multitude of predefined special-purpose rule sets, called *triggers*. The whole bunch of predefined triggers containing a few hundred rules is called the *Logic Deck*.

To achieve a high degree of adaptability, the Automation Manager is implemented as a virtual machine with its own instruction set of a few hundred instructions, specialized on abstract resource management. There are instructions to specify resources and their dependencies, or to change variables' values; other instructions start the evaluation and application of the rules. The user specifies the resources and their dependencies in a so-called *Automation Policy*. At initialization time, the Automation Manager therefore loads a configuration file containing the Automation Policy. This file contains virtual machine instructions for each application, group, and system. Moreover, it includes instructions that generate relationships between those abstract resources, for example to reflect start or stop dependencies. Abstract

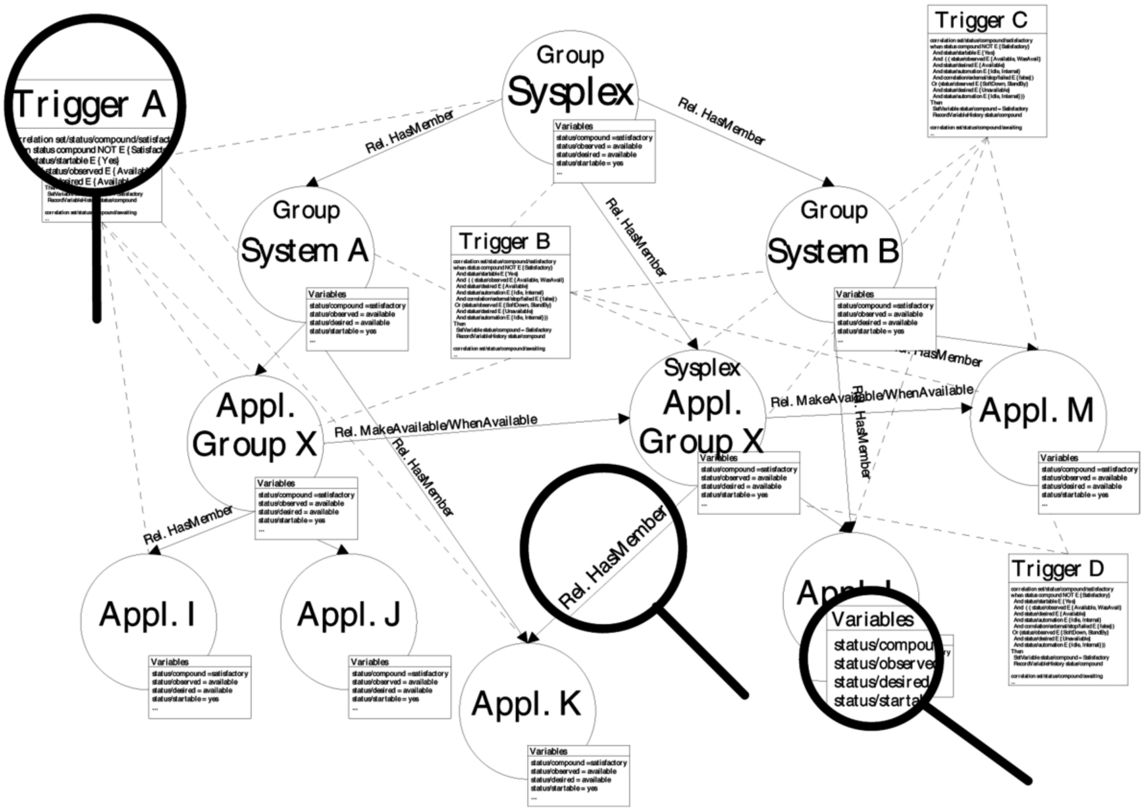

Fig. 2. Resource structure representation in the Automation Manager.

resources and their relationships internally build up a graph which is called the *Resource Structure* (see Fig. 2).

Each abstract resource maintains its own set of state variables, containing variables for status information, configuration specific information, or local variables used for internal processing. The rules are shared between different resources via triggers. A local expert system, contained in the Resource Structure above, is shown in Fig. 3.

Execution of the whole expert system now works as follows. After the virtual machine is initialized and the resource structure is built up, the system waits for changes of its variables. These may occur for either internal or external reasons. The latter may happen due to a variable update provided by the Automation Agent, or by a human operator directly interacting with SA. The former is triggered by other variables of the same resource altering their values, or by a change of a dependent variable of another resource. When such a change occurs, all rules are re-evaluated. This process is described below in greater detail.

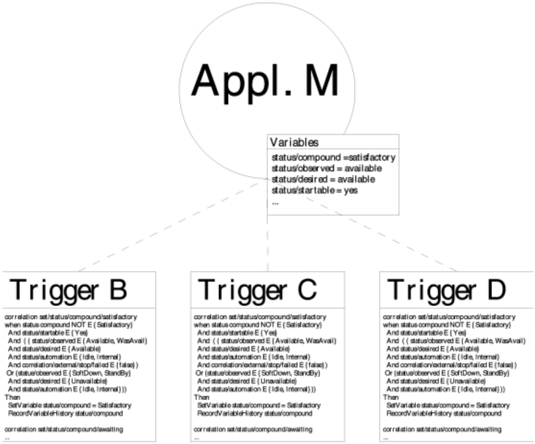
Appl. M
Variables
status/compound =satisfactory
status/observed = available
status/desired = available
status/startable = yes
...
Trigger B
Trigger C
Trigger D
correlation set/status/compound/satisfactory
when status compound NOT E {Satisfactory}
And status/startable E {Yes}
And ((status/observed E {Available, WasAvail}
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
correlation set/status/compound/awaiting
...

Fig. 3. Local expert system of a resource.

The verification we present here is not concerned with the verification of a certain scenario represented by some Resource Structure, but with the whole set of predefined rules, the Logic Deck, itself.

2.3. The Automation Manager's rule base

All correlation rules are of the form

correlation $\langle\texttt{name}\rangle$:
when $\langle\texttt{formula}\rangle$
then $\langle\texttt{action list}\rangle$

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

$\langle\texttt{var}\rangle$ E $\{\langle\texttt{val}_1\rangle,\ldots,\langle\texttt{val}_n\rangle\}$
$\langle\texttt{var}\rangle$ NOT E $\{\langle\texttt{val}_1\rangle,\ldots,\langle\texttt{val}_n\rangle\}$

```
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
```

Fig. 4. Example of a correlation rule.

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` $\langle \texttt{var} \rangle = \langle \texttt{val}_i \rangle$. Other actions in the SA system are mainly used for event logging and to present messages to the user. We assume that only one `SetVariable`-action is present in each rule's action list. This is not enforced by the instruction language of the Automation Manager, but turned out to be the case for the rules we encountered. Fig. 4 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 an abstract resource instance's variable changes its value, the automation has to re-evaluate to reflect this change. Therefore the manager takes all rules of the triggers that are linked to the instance into consideration: 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, repeating the process. The order in which rules are evaluated is only partially specified using priority schemes for rules. Thus, in our formalization we do not make any assumptions about the rule evaluation order and consider it completely unspecified.

Because resources have relationships like start or stop dependencies, a state change on one resource can lead to state changes on other resources. This is implemented with *relationship correlations*. The basic idea is to copy the value of a state variable from one instance to another, by which state variables from different resources behave identical and thus can be identified. Thus, by means of relationship correlations, communication between resources is realized. This also implies that changes on one resource can cause re-evaluations on others.

As seen above, 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

We have selected PDL as formalization language for the correlation rules and the computations done by the Automation Manager. There are several 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 the 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 [37], and a similar extension is due to Harel and Sherman [13].

PDL allows reasoning about programs (denoted by $\alpha, \beta, \ldots$) and their properties, and therefore contains language constructs for programs as well as for propositional formulae. Atomic propositions ($P, Q, R, \ldots$) can be combined to compound PDL formulae ($F, G, \ldots$) using the Boolean connectives $\neg, \vee$, and $\wedge$. Composite programs are composed out of atomic programs using three different

Table 1
ΔPDL symbols and their semantics (adapted from Ref. [12])

Symbol	Name	Semantics σ,τ
$\top$	Truth	$\sigma(\top) = \mathscr{S}$
$\perp$	Falsity	$\sigma(\perp) = \theta$
$\neg$	Negation	$\sigma(\neg F) = \mathscr{S} \backslash \sigma(F)$
$\vee$	Disjunction	$\sigma(F \vee G) = \sigma(F) \cup \sigma(G)$
$\wedge$	Conjunction	$\sigma(F \wedge G) = \sigma(F) \cap \sigma(G)$
$\langle\alpha\rangle$	Possible postcondition	$\sigma(\langle\alpha\rangle F) = \{s \in \mathscr{S} \mid \exists t.(s,t) \in \tau(\alpha) \wedge t \in \sigma(F)\}$
$[\alpha]$	Necessary postcondition	$\sigma([\alpha]F) = \{s \in \mathscr{S} \mid \forall t.(s,t) \in \tau(\alpha) \Rightarrow t \in \sigma(F)\}$
$\Delta\alpha$	Divergence of α^*	$\sigma(\Delta\alpha) = \{s_0 \in \mathscr{S} \mid \exists s_1, s_2, \ldots \forall i \geq 0.\ (s_i, s_{i+1}) \in \tau(\alpha)\}$
;	Consecutive execution	$\tau(\alpha, \beta) = \tau(\alpha) \cdot \tau(\beta) = \{(s,t) \mid \exists u.(s,u) \in \tau(\alpha) \wedge (u,t) \in \tau(\beta)\}$
$\cup$	Non-deterministic Choice	$\tau(\alpha \cap \beta) = \tau(\alpha) \cup \tau(\beta)$
*	Repetition	$\tau(\alpha^*) = \{(s,t) \mid \exists k \exists s_0 \ldots s_k . s_0 = s \wedge s_k = t \wedge (s_i, s_{i+1}) \in \tau(\alpha)\}$
F?	Test	$\tau(F?) = \{(s,s) \mid s \in \sigma(F)\}$

connectives: $\alpha;\beta$ denotes program sequencing, $\alpha \cup \beta$ non-deterministic choice, and α^* a finite, non-deterministic 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 fails otherwise. 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.

A summary of the syntactical components of PDL and their semantics is shown in Table 1. PDL semantics is based on the notions of computation states, transitions between them, and properties that hold in these states. Therefore, let $\mathscr{S}$ denote the set of all (not further specified) computation states, $\sigma : \mathscr{F} \rightarrow 2^{\mathscr{S}}$ is a valuation function, mapping a formula to the states in which it is valid, and $\tau : \mathscr{P} \rightarrow 2^{\mathscr{S} \times \mathscr{S}}$ is a transition function, mapping a program α to the pairs of states (s,t) such that execution of program α may lead from state s to state t. Let $\sigma_0 : \mathscr{F}_0 \rightarrow 2^{\mathscr{S}}$ resp. $\tau_0 : \mathscr{P}_0 \rightarrow 2^{\mathscr{S} \times \mathscr{S}}$ be the restrictions of σ resp. τ to atomic formulae resp. programs. Then σ (resp. τ) is uniquely determined by σ_0 (resp. τ_0) using the semantical definitions of Table 1. The triple $\mathscr{K} = (\mathscr{S}, \sigma_0, \tau_0)$ is called a Kripke frame and assigns meaning to PDL expressions. For a fixed $\mathscr{K}$ and a state $s \in \mathscr{S}$, we write $s \vDash F$ for $s \in \sigma(F)$, and say that s *satisfies* F. If $s \vDash F$ for all states $s \in \mathscr{S}$ and all Kripke frames $\mathscr{K}$ we say that F is *valid*, and write $\vDash F$. If $\mathscr{K}$ has to be made explicit, we also use the notations $(\mathscr{K}, s) \vDash F$ and $\mathscr{K} \vDash F$.

Some program constructs occurring frequently in conventional programming languages are expressed in PDL as:

$$\textbf{if } F \textbf{ then } \alpha \textbf{ else } \beta = (F?; \alpha) \cup (\neg F?; \beta)$$
$$\textbf{while } F \textbf{ do } \alpha = (F?; \alpha)^*; \neg F?$$
$$\textbf{repeat } \alpha \textbf{ until } F = \alpha; (\neg F?; \alpha)^*; F?$$

As another example consider Hoare's partial correctness assertion $\{F\}\alpha\{G\}$. It says that if program α is started in a state satisfying F, then, provided that α halts, it does so in a state where G holds. In PDL the equivalent to Hoare's assertion is $F \Rightarrow [\alpha]G$.

We refer the reader to Harel's introductory chapter on PDL [12] for a more complete elaboration.

3.1. Encoding of the correlation rules and the status computation

Encoding of correlation rules and the 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. For any set V of correlation rule variables, we thus get an additional propositional restriction RES_V:

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

Formulae similar to RES_V also occur in the context of propositional encodings of planning problems, where they are referred to as *linear encodings* [18].

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 variables in V_S 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) $P_{w,e} \Rightarrow [\alpha_{v,d}]P_{w,e}$ for all $w \in V_S$, $w \neq v$ and $e \in D_w$.

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

Using techniques from modal correspondence theory [39,40] we can derive properties of the program transition relation imposed by the restriction RES_α. Therefore, an admissible program transition relation $\tau(\alpha_{v,d})$ for an atomic program $\alpha_{v,d}$ must have the following properties:

(P1$_\mathrm{a}$) $\forall s_0 \exists s_1.(s_0, s_1) \in \tau(\alpha_{v,d})$
(P1$_\mathrm{b}$)
$\forall s_0 s_1.(s_0, s_1) \in \tau(\alpha_{v,d}) \wedge (s_0, s_2) \in \tau(\alpha_{v,d}) \Rightarrow s_1 = s_2$
(P2) $\forall s_0 s_1.(s_0, s_1) \in \tau(\alpha_{v,d}) \Rightarrow s_1 \in \sigma(P_{v,d})$
(P3)
$\forall s_0 s_1.s_0 \in \sigma(P_{w,e}) \wedge (s_0, s_1) \in \tau(\alpha_{v,d}) \Rightarrow s_1 \in \sigma(P_{w,e})$
for $w \neq v$, which correspond to the respective PDL formulae.

3.1.3. Correlation rules

In the following, we assume that for each variable-value pair (v, d) there is at most one rule $R(v, d)$ 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

$$\tau(v\ \mathtt{E}\{d_0, \ldots, d_j\}) = P_{v,d_0} \vee \cdots \vee P_{v,d_j},$$

$$\tau(v\ \mathtt{NOT\ E}\{d_0, \ldots, d_j\}) = \neg P_{v,d_0} \wedge \cdots \wedge \neg P_{v,d_j},$$

and extended to complex formulae in the obvious way. Thus, for each pair (v, d) we obtain a unique translation $F_{v,d}$ of the **when**-part of the associated rule $R(v, d)$. For the **then**-part we only have to consider actions setting variables, which are translated by τ to their corresponding atomic PDL programs:

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

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:

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

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, corresponding to loops of length 1.

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:

$$\mathrm{S} = \bigcup_{v \in V_S, d \in D_v} R_{v,d}, \qquad \mathrm{AM} = \mathrm{S}^*; \left(\bigwedge_{v \in V_S, d \in D_v} (F_{v,d} \Rightarrow P_{v,d}) \right)?$$

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. However, there are external variables (observation variables) that may change their values during computation. For our consistency properties we 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 RES_V, the atomic program specifications RES_α, and the fixing of all observation variables during computation. We therefore define

$$\mathrm{PRE} := \mathrm{RES}_\mathrm{V} \wedge \mathrm{RES}_\alpha \wedge \bigwedge_{v \in V_S, w \in V_O, d \in D_v, e \in D_w} (P_{w,e} \Rightarrow [\alpha_{v,d}]P_{w,e}).$$

The last part of PRE, fixing the observation variables, also has a first order predicate logic equivalent, which is obtained using correspondence theory as for RES_α above. We get

$(P4)$

$$\forall s_0 s_1.s_0 \in \sigma(P_{w,e}) \wedge (s_0, s_1) \in \tau(\alpha_{v,d}) \Rightarrow s_1 \in \sigma(P_{w,e})$$

for all $v \in V_S$, $w \in V_O$.

Now addressing consistency properties, the following

ΔPDL formula, provided it is valid, guarantees that there is no divergent computation:[3]

$$\text{PRE} \Rightarrow \neg \Delta \text{S}. \tag{1}$$

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:

$$\text{PRE} \Rightarrow (\langle \text{AM} \rangle p \Leftrightarrow [\text{AM}]p). \tag{2}$$

Confluence of the rule system, i.e. the property that all ambiguities about which rule should be applied next, eventually are irrelevant because they lead to the same computation state, is expressed as follows:

$$\text{PRE} \Rightarrow (\langle \text{S}^* \rangle [\text{S}^*]p \Rightarrow [\text{S}^*]\langle \text{S}^* \rangle p). \tag{3}$$

The corresponding first order formula in this case is

$$\begin{aligned} &\forall stu.(s,t) \in \tau(\text{S}^*) \wedge (s,u) \in \tau(\text{S}^*) \\ &\qquad \Rightarrow \exists v.(t,v) \\ &\qquad \in \tau(\text{S}^*) \wedge (u,v) \in \tau(\text{S}^*). \end{aligned}$$

Obviously, there are many more consistency criteria that we will, however, not elaborate on. Instead, we concentrate on the termination property.

As termination is defined as the absence of an infinite sequence of consecutive computation states, we have to make the notion of a state more precise. We will use a state space that is isomorphic to the exponential-sized (in the number of predicates) collapsed model. A *state* s is an assignment to the propositional variables $\text{PROP} = \{P_{v,d} | v \in V, d \in D_v\}$, i.e. a function

$$s : \text{PROP} \rightarrow \{0,1\}.$$

A state s is said to be *proper* if it correctly reflects the finite domain restriction RES_V, i.e. if s is a model of RES_V, or, equivalently in symbols, $s \vDash \text{RES}_\text{V}$. A pair of states (s_0, s_1) is called an *$R_{v,d}$-transition*, if execution of the rule that sets variable v to value d leads from s_0 to s_1.

Definition 1. Let $v \in V_S$, $d \in D_v$, and let $\mathscr{K} = (\mathscr{S}, \sigma_0, \tau_o)$ be a Kripke frame. A pair of states (s_0, s_1) is called an $R_{v,d}$-transition, denoted by $s_0 \overset{v=d}{\rightarrow} s_1$, when $(s_i, \mathscr{K}) \vDash \text{PRE}$ for $i \in \{0,1\}$ and $(s_0, s_1) \in \tau(R_{v,d})$.

Lemma 1. *Let* $s_0 \overset{v=d}{\rightarrow} s_1$. *Then the following holds*:[4]

(a) $s_0, s_1 \vDash \text{RES}_\text{V}$
(b) $s_0 \vDash F_{v,d} \wedge \neg P_{v,d}$
(c) $s_1 \vDash P_{v,d}$
(d) $s_0 \vDash P_{w,e} \Leftrightarrow s_1 \vDash P_{w,e}$ *for all* $w \neq v$, $e \in D_w$.

Lemma 1 clarifies some properties of $R_{v,d}$-transitions, but its main use will be later on in translating PDL formulae to propositional logic formulae.

Turning back to divergent computations, and noting that 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

$$s_0 \overset{v=d_1}{\rightarrow} s_1 \overset{v=d_0}{\rightarrow} s_0 \tag{4}$$

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

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

It involves two variables and cannot be decomposed 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 $s_0, s_1 \vDash F_{v,d_0} \wedge F_{v,d_1}$. It is thus of particular 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 RES_α of the atomic assignment programs and on the validity of the propositions $P_{v,d}$, considering the restrictions RES_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. This facilitates the application of advanced propositional SAT-checkers which have shown good performance on a number of industrial strength problems (see, for example, Refs. [3,20]).

3.3. Conversion to propositional satisfiability

Conversion to a purely propositional formalism requires handling 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 allows the formulation of propositional properties concerning multiple computation states.

[3] Note that this property cannot be expressed in ordinary PDL [37].

[4] Proofs of all lemmas can be found in Appendix A.

Lemma 2. *Let* $s_0 \overset{v=d}{\longrightarrow} s_1$. *Then* $s_1 \vDash F$ *iff* $s_0 \vDash F|_{v=d}$.

The Automation Manager program terminates if $\vDash$ PRE $\Rightarrow \neg\, \Delta S$. By definition of the semantics of ΔPDL and the single-step Automation Manager program S, this is equivalent to

$$(\mathscr{K}, s_0) \vDash$$

$$\text{PRE} \Rightarrow\ \neg\, \exists s_1, s_2 \ldots \forall i \geq 0. \bigvee_{v \in V_S, d \in D_V} (s_i, s_{i+1}) \in \tau(R_{v,d})$$

for all $\mathscr{K}, s_0$. By using Definition 1 we get for all s_0

$$s_0 \vDash \text{PRE} \Rightarrow \neg\, \exists s_1, s_2 \ldots \forall i \geq 0 \exists v_i d_i \cdot s_i \overset{v_i=d_i}{\longrightarrow} s_{i+1}. \tag{5}$$

We want to specialize on 2-loops now. According to Formula (5), absence of 2-loops is expressed by

$$s_0 \vDash \text{PRE} \Rightarrow \neg\, \exists s_1, v, d_0, d_1 \cdot s_0 \overset{v=d_1}{\longrightarrow} s_1 \overset{v=d_0}{\longrightarrow} s_0. \tag{6}$$

The two $R_{v,d_{0/1}}$-transitions can be performed provided the following holds (by Lemma 1):

$$s_0, s_1 \vDash \text{RES}_\text{V} \qquad s_1 \vDash P_{v,d_1} \qquad s_0 \vDash P_{v,d_0}$$

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

$$s_0 \vDash P_{w,e} \Leftrightarrow s_1 \vDash P_{w,e} \qquad \text{for all } w \neq v,\ e \in D_w$$

According to Lemma 2, this is equivalent to

$$s_0 \vDash \text{RES}_\text{V} \wedge P_{v,d_0} \wedge (F_{v,d_1} \wedge$$

$$\neg\, P_{v,d_1}) \wedge (\text{RES}_\text{V} \wedge P_{v,d_1} \wedge F_{v,d_0} \wedge \neg\, P_{v,d_0})|_{v=d_1},$$

which can be further simplified to

$$s_0 \vDash \text{RES}_\text{V} \wedge P_{v,d_0} \wedge F_{v,d_1} \wedge F_{v,d_0}|_{v=d_1}.$$

Substituting this formula back into Formula (6) we obtain, after dropping the now superfluous existential quantification over s_1,

$$s_0 \vDash \text{PRE} \Rightarrow \neg\, \exists v, d_0, d_1 \cdot \text{RES}_\text{V} \wedge P_{v,d_0} \wedge F_{v,d_1} \wedge F_{v,d_0}|_{v=d_1}.$$

As the properties of atomic programs are not needed any more now, we can replace PRE by RES_V. Simplification and moving the quantifiers to the front yields

$$\text{RES}_\text{V} \Rightarrow \neg\, (P_{v,d_0} \wedge F_{v,d_1} \wedge F_{v,d_0}|_{v=d_1}) \qquad \forall v, d_0, d_1.$$

The propositional formula expressing absence of 2-loops therefore reads

$$\text{RES}_\text{V} \Rightarrow \neg\, (P_{v,d_0} \wedge F_{v,d_1} \wedge F_{v,d_0}|_{v=d_1}), \tag{7}$$

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

$$\text{RES}_\text{V} \Rightarrow \neg\, (P_{v,d_0} \wedge F_{v,d_1} \wedge F_{v,d_2}|_{v=d_1} \wedge F_{v,d_0}|_{v=d_2})$$

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 due to different types of loops involving modification of multiple finite domain variables.

3.4. Generalization to n-loops

We now consider the general case of n-loops, assuming that the potential loop involves states $s_0, \ldots, s_{n-1}$. We then have $s_i \overset{v_i=d_i}{\longrightarrow} s_{i+1}$ for all i with $0 \leq i < n-1$, and $s_{n-1} \overset{v_{n-1}=d_{n-1}}{\longrightarrow} s_0$. As in the restricted cases above, by Lemma 1, the R_{v_i,d_i}-transitions can be performed provided the following holds:

$$s_i \vDash \text{RES}_\text{V} \quad \text{for } 0 \leq i < n \tag{8a}$$

$$s_0 \vDash P_{v_{n-1},d_{n-1}} \tag{8b}$$

$$s_i \vDash P_{v_{i-1},d_{i-1}} \quad \text{for } 0 < i < n \tag{8c}$$

$$s_i \vDash F_{v_i,d_i} \wedge \neg\, P_{v_i,d_i} \quad \text{for } 0 \leq i < n \tag{8d}$$

$$\begin{aligned} &s_i \vDash P_{w,e} \Leftrightarrow s_{i-1} \vDash P_{w,e} \\ &\quad \text{for } w \neq v_{i-1},\ e \in D_w \text{ and } 0 < i < n \end{aligned} \tag{8e}$$

$$s_0 \vDash P_{w,e} \Leftrightarrow s_{n-1} \vDash P_{w,e} \quad \text{for } w \neq v_{n-1},\ e \in D_w \tag{8f}$$

Of course, it would be possible to test for all possible n-loops whether the formulae above hold for the inclosed states s_i. But as there are $N_S = \prod_{v \in V} 2^{|D_v|}$ states (and still $N_P = \prod_{v \in V} |D_v|$ proper states), the number N_n^{SEQ} of possible n-loop sequences grows very quickly with $N_n^{\text{SEQ}} = (N_S)^n$ (resp. $(N_P)^n$). Moreover, most of the states trivially do not fulfill the above formulae, independent of the F_{v_i,d_i} involved. Therefore we try to consider only 'sensible' loops.

In the following, a sequence $\vec{a} = (a_0, \ldots, a_{n-1})$ of atomic programs $a_i = \alpha_{v_i,d_i}$ is called an *action sequence*. Each action sequence $\vec{a}$ is related to a whole set $T(\vec{a})$ of transition sequences by

$$T(\alpha_{v_0,d_0}, \ldots, \alpha_{v_{n-1},d_{n-1}})$$

$$= \{(s_0, \ldots, s_{n-1}) | s_i \overset{v_i=d_i}{\longrightarrow} s_{i+1}, s_{n-1} \overset{v_{n-1}=d_{n-1}}{\longrightarrow} s_0\}.$$

Note that it is the action sequence $\vec{a}$ that determines whether or not *all* of the associated transition sequences form a loop. Thus, in the following we characterize action sequences that may lead to loops.

Definition 2 (Loop Candidate Sequence). An action sequence $\vec{a} = (a_0, \ldots, a_{n-1})$ with $a_i = \alpha_{v_i,d_i}$ is a loop candidate sequence, provided that the following two criteria hold:

(1) There are either none or at least two actions for each variable: For all $a_i = \alpha_{v,d}$ there exist j and $d' \neq d$ such that $a_j = \alpha_{v,d'}$.

(2) Between two actions setting variable v to the same value d there must be another action involving variable v: If $a_i = a_j = \alpha_{v,d}$ and $i < j$ then there exist indices k, l such that
  (a) $i < k < j$ and $a_k = \alpha_{v,d'}$ for some d' and
  (b) either $l < i$ or $l > j$, and $a_k = \alpha_{v,d''}$ for some d''.

Definition 3 (Last Action Set). Given an action sequence $\vec{a}$, the *last action set* $L(\vec{a})$ contains those actions that are the last for that variable in the sequence:

$$L(\vec{a}) = \{a_i = \alpha_{v,d} | a_j = \alpha_{v,d'} \Rightarrow j \leq i\}.$$

Lemma 3. *Let* $\vec{a} = (a_0, \ldots, a_{n-1})$ *be a loop candidate sequence. Then*

$$\mathrm{LOOP}(\vec{a}) := \mathrm{RES_V} \wedge \bigwedge_{\alpha_{v,d} \in L(\vec{a})} P_{v,d} \wedge \bigwedge_{0 \leq i < n, a_i = \alpha_{v_i,d_i}} F_{v_i,d_i}|_{v_{i-1}=d_{i-1},\ldots,v_0=d_0} \tag{9}$$

is satisfiable iff there is a looping transition sequence in $T(\vec{a})$. *Moreover, the models of* LOOP($\vec{a}$) *are (starting) states of looping transition sequences.*

Lemma 4. *Let* $\vec{a}$ *be an action sequence. If* $T(\vec{a}) \neq \emptyset$, *then* $\vec{a}$ *is a loop candidate sequence.*

Theorem 1. LOOP($\vec{a}$) *is unsatisfiable for all loop candidate sequences* $\vec{a}$ *of length n iff the expert system contains no n-loops.*

The maximal length of a loop that we have to search for is only limited by the number of different computation states. Thus, we have to check for loops of length n up to $n_{\max} = \prod_{v \in V_S} |D_v|$. It is even possible to construct a rule system that loops but contains no loops of length $n < n_{\max}$. Consider, for example, the generalization of an m-bit counter, where the only loop involves 2^m states. In a practical setting, there may be an a priori limit on loop lengths that can be considerably smaller.

Compared to the n-fold product of the set of all states, the set of loop candidate sequences of length n can cause a substantial reduction on cases to be checked. As mentioned above, the number of proper state sequences of length n is determined by $N_n^{\mathrm{SEQ}} = (\prod_{v \in V} |D_v|)^n$. The number of action sequences of length n is (asymptotically) already much lower with $N_n^{rmASEQ} = (\sum_{v \in V_S} |D_v|)^n$. For a further analysis of the number of loop candidate sequences N_n^{LCS}, we make some simplifying assumptions. Let $d = \max_{v \in V_S} \{D_v\}$ be the maximal variable domain size, and $k = |V_S|$ be the number of state variables. Then the number of all loop candidate sequences of length n can be approximated by $(d{\cdot}k)^n$. The number of sequences violating criterion (1) of Definition 2 can be approximated by

$$N_1 = d{\cdot}k{\cdot}n{\cdot}(d{\cdot}(k-1)^{n-1},$$

and the number of sequences violating criterion (2) by

$$N_2 = d{\cdot}k{\cdot}n{\cdot}(n-3){\cdot}(d{\cdot}(k-1)^{(n-2)/2}{\cdot}(d{\cdot}k)^{(n-2)/2}.$$

Thus, $N_n^{\mathrm{LCS}} \approx (d{\cdot}k)^n - N_1 - N_2$.

What is not considered so far, but leads to a further improvement, is that only those loop candidate sequences have to be examined which do not contain another loop candidate sequence as a subsequence.

3.5. Rule evaluation order

Not all loops detected by the presented method inevitably have to occur in an actual implementation. For example, in the formulae above, the states of the loop have to be reachable states of the computation. With regard to observation variables, we are not allowed to suppose any restrictions on possible variable values, so all combinations have to be considered viable for them. But some computation states may not occur due to a pre-imposed rule evaluation order. So, some states may be unreachable, and do not have to be considered.

We modeled the evaluation order by assuming that all rules changing variables with higher priority have been computed already. We denote by $w > v$ that the evaluation priority of w is greater than that of v, i.e. all rules $R_{w,e}$ are evaluated before any rule $R_{v,d}$ setting variable v is considered. Then, to take into account rule evaluation priority, we can use the following extended formula instead of Formula (7) to check for 2-loops:

$$\mathrm{RES_V} \wedge \bigwedge_{w \in V_S, w > v, e \in D_w} (F_{w,e} \Rightarrow P_{w,e}) \Rightarrow \neg\,(P_{v,d_0} \wedge F_{v,d_1} \wedge F_{v,d_0}|_{v=d_1}). \tag{10}$$

This modification naturally extends to the case of 3-loops and n-loops. In our experiments, however, we only considered the case of 2-loops.

Note that rule evaluation order may counteract fairness of rule evaluation. That is, by fixing a certain rule evaluation order some rules may not be evaluated at all, because of either a loop in the computation of values for variables with higher priority, or because of two simultaneously activated rules, where due to the fixed evaluation order always the same one rule is selected. Assuming that no such loops occur during the computations of variables with higher priority, the first case vanishes and only the second case remains. Thus, fairness of rule selection has to be assured only for variables with the same priority.

4. Verification techniques

We will now describe the techniques we used to prove the propositional formulae of Section 3. 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 (DP) prover to show the unsatisfiability of the negations of the 2-loop formulae. The prover was developed in the Symbolic Computation group at the University of Tübingen in collaboration with A. Kaiser for checking industrial product documentation [16,20,31]. In contrast to other DP implementations [21,42], it is specifically geared towards industrial strength inputs. First because it does not require the (potentially very large) input to be in conjunctive normal form. Second, 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 $\mathrm{RES_V}$ to Boolean logic. More precisely, there is a set of n-ary *selection operators* S_1^n, such that $S_1^n(F_1,\dots,F_n)$ holds when exactly one of the formulae $F_1,\dots,F_n$ holds. Using this selection operator, and assuming that the variable domains are ordered, i.e. $D_v = (d_1^v,\dots,d_{|D_v|}^v)$, we can restate $\mathrm{RES_V}$ as

$$\bigwedge_{v\in V} S_1^{|D_v|}(P_{v,d_1^v},\dots,P_{v,d_{|D_v|}^v}).$$

So instead of the original formula $\mathrm{RES_V}$, which is quadratic in the domain sizes $|D_v|$, we just have to deal with a linear-size formula in the extended language. Third, it contains an explanation component that helps to pinpoint small unsatisfiable subsets of the input.

Today's SAT-solvers are often very efficient in solving encodings of real-world problems, where they are capable of handling problems containing up to several thousands of variables. On the other hand, as the SAT problem is NP complete, there are obviously much harder SAT instances even at smaller sizes. However, it turned out that they occur infrequently in practice [19]. Our experiments corroborate this observation.

BDD-based approach. Another technique to prove propositional formulae are binary decision diagrams (BDDs). BDDs uniquely represent Boolean functions as binary graphs, so that validity can be checked in constant time, once the BDD is built. BDDs were successfully employed in the realm of hardware verification [5] where one of the most common applications is to prove the equivalence of two hardware designs. So, in addition to DP solvers, we also experimented with BDDs, where we used Somenzi's CU Decision Diagram package [33].

The Davis–Putnam algorithm produces as output either 'satisfiable' or 'unsatisfiable', in the former case possibly accompanied by a list of models (examples). In contrast to this, BDD construction always results in a formula. So if the problem is satisfiable, i.e. not equivalent to $\perp$ (false), the resulting BDD encodes all satisfiable instances. This is a big advantage over DP-style SAT-solvers, as it allows further inspection of the resulting formula. Hence, if the outcome 'unsatisfiable' needs explanation, the minimal unsatisfiable subsets as computed by our DP implementation are needed; otherwise, we need a concise representation of the satisfiable cases, much as by BDDs. In our application of proving termination of the expert system, we were thus able to further simplify the counter-model representation of the cases in which the expert system starts oscillating between different computation states.

The simplification we applied to Formula (10) distinguishes between *main variables* and *side variables*. Main variables are those variables occurring in the rules R_{v,d_0} resp. R_{v,d_1}. These are of special importance, as these are variables of the rules making up the possible 2-loop. All variables occurring only in $\mathrm{RES_V}$ or in rules with higher priority, i.e. in $R_{w,e}$ with $w > v$, are side variables, which can be thought of as describing the environment in which the loop possibly occurs. In order not to confuse the user with additional variables irrelevant to the occurrence of the loop, we therefore applied existential abstraction over all side variables. Thus, we are only interested in assignments to main variables that can in some way be extended to all side variables, while still fulfilling the side conditions of Formula (10).

More formally, we generated a quantified Boolean formula $\exists\vec{X}.F$ where F is Formula (2-loop-exp-for) and $\vec{X}$ contains exactly those variables not appearing in R_{v,d_0} and R_{v,d_1}, i.e. $\vec{X} = V\setminus(\mathrm{Var}(R_{v,d_0})\cup\mathrm{Var}(R_{v,d_1}))$.

It would be interesting to compare the efficiency of the two approaches for generating proofs of consistency properties of expert systems in general, as the two methods have shown advantages in differing fields [38]. In our experiments, however, the formulae were too small to observe a significant discrepancy in efficiency.

Implicit assumptions on observation variables. Not all combinations of possible values for observation variables really do 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 $\mathrm{RES_V}$ 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

We conducted experiments with a subset of the rules of the Automation Manager's Logic Deck 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 the states of other resources. It can take any of seven different values, so we had to perform 21 proofs of Formula (10) 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 [16], formula sizes could be kept small. All formulae contained 72 propositional variables, 39 of them were state variables, 33 observation variables. The generated propositional logic formulae contained around 1500 atomic symbols. 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 of the rule system. Implicit assumptions on which states are reachable have to be made explicit in order to achieve practically useful results. Thus, we added the above-mentioned 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. For simplification, we then 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. Thus the final version is verified to contain no 2-loop-errors. So, by identifying real defects, we could further increase the reliability of the Automation Manager.

Fig. 5 further illustrates the results of our experiments. The upper part shows two correlation rules for which a 2-loop was detected. The generated propositional verification condition for these rules, according to Formula (10), contains encodings of both rules' **when**-parts (on the right hand side of the implication), as well as the remaining 34 rules (as part of the left hand side of the implication). We also added further restrictions on observation variables, for example,[5]

$$\mathtt{status\_startable\_no}$$

$$\Rightarrow \mathtt{status\_observed\_starting}.$$

Running our DP-style prover for this particular case generates a list of 301 models providing examples for which Formula (2-loop-exp-for) does not hold. One of these models consists of the following propositional variables set to true, and all others set to false:

```
_fd_extstopdelayed_true correlation_can_be_started
correlation_external_stop_failed correlation_may_send_orders
correlation_set_status_compound_degraded correlation_set_status_compound_problem
flag_automation_disabled flag_expect_extstop_yes flag_external_stop_always flag_hold_no
group_nature_basic request_desiredstatus_da request_desiredstatus_origin
status_automation_internal status_compound_automating status_desired_available
status_observed_degraded status_observed_null status_startable_yes test_0var_off
```

First correlation rule:

```
correlation set/status/compound/degraded :
when        status/compound NOT E {Degraded}
     AND status/automation E {Idle, Internal}
     AND ( ( status/desired E {Available}
               AND status/observed E { Degraded}
               AND status/startable E { Yes }
             )
             OR
             ( status/observed E {Starting, Stopping}
               AND correlation/group/IsAutomating NOT E { True}
               AND correlation/external/stop/failed E { False }
             )
         )
then  SetVariable status/compound = Degraded
```

Second correlation rule:

```
correlation set/status/compound/problem :
when        status/compound NOT E {Problem}
     AND ( status/observed E { Problem }
             OR status/automation E { Problem }
             OR ( status/automation E { Idle, Internal }
                  AND ( status/startable E { No }
                        OR status/observed E { Harddown } )
             )
             OR ( status/observed E { Available, Degraded, Problem,
                                       Starting, Stopping, WasAvail }
                  AND correlation/external/stop/failed E { True }
             )
         )
     AND test.0var E { Off }
then  SetVariable status/compound = Problem
```

Variable settings:

```
status/startable = Yes,
status/observed = Degraded,
status/desired = Available,
status/automation = Internal,
test.0var = Off,
correlation/external/stop/failed = True,
correlation/group/IsAutomating = False,
```

Fig. 5. Correlation rules and variable settings causing 2-loop.

Using the BDD representation of the 2-loop-formula and existentially quantifying over all variables that do no occur in the **when**-parts of the two correlation rules of Fig. 5, we arrive at one remaining case:

```
status_startable_yes
status_observed_degraded
status_desired_available
status_automation_internal
test_0var_off
correlation_external_stop_failed
```

The variable setting corresponding to this case is also shown at the bottom of Fig. 5. So when the variables are set as indicated, SA's rule system oscillates between the state where $\mathtt{status\_compound} = \mathtt{Degraded}$ and the state with $\mathtt{status\_compound} = \mathtt{Problem}$. SA experts at IBM corrected this loop by modifying the second correlation rule.

In the other two situations where 2-loops have shown up, three cases remained to consider (as opposed to one in the example above) after application of BDD simplification techniques.

[5] Names for propositional variables are constructed by replacing slashes by underscores and appending the variable's value with an underscore to the finite domain's variable name. The translation of Boolean finite domain variables (i.e. with domains{True, False}) is simplified, as should be obvious.

6. Industrial experiences summary

The performance and impact of formal methods in industrial settings has been the subject of many articles, see Refs. [4,6,11,26], for example. Hall [11], and Bowen and Hinchey [4], have reported success stories and have used these to argue that widely held concerns and reservations about formal methods are indeed myths that need to be dispelled. Fenton and Pfleeger have taken a more cautious view (Ref. [7], embedded in Ref. [26]). Some of their concern is that most work to date has been on specification, that we must better understand how to choose among the many competing formal methods, and that there is still no hard evidence to show that formal methods are cost effective, or that sufficient numbers of developers and users can be trained in them.

Combining experiences from this project and our previous BIS[6] project for proving consistency assertions for automotive product data [20,31], we now report on some of our own findings, and we try to relate them to the well-known 7 + 7 'myths' [4,11]. Note that both projects are concerned with finding and removing defects in mostly finished products (BIS is concerned with data consistency only). Since both originated in an order from industry, we would at once argue that they help dispel myths 6 and 7, because our formal methods were acceptable to users and were used on real software, resp. real data. In both cases, the methods were also necessary, dispelling myth 12, because we found residual bugs; however, it can be argued that the bugs were unlikely to surface.

Verification, validation, and debugging. First, let us turn to Hall's myth 1: 'Formal methods can guarantee that software is perfect'. Formal verification is attractive because, as Dijkstra has observed, it can prove the absence of errors while testing can only prove their presence. The notion of formal verification carries the connotation of complete correctness: First, a formal model $\mathscr{M}_P$ and a specification $\mathscr{S}_P$ are produced, and then it is shown that $\mathscr{M}_P \models \mathscr{S}_P$, i.e. $\mathscr{S}_P$ holds of $\mathscr{M}_P$. In contrast, validation must needs remain incomplete, because the user's expectations are informal. However, providing complete formal specifications and formal semantics is very hard and time consuming for humans, and mastering the ensuing proofs is just as hard for theorem provers, whether human or mechanical. In rule-based systems, at least the semantics part is manageable, due to their proximity to logic formalisms. Complete formal specifications, however, just do not exist in practice, or they are feasible only for a small subset of the entire system. It is only possible to capture a few of the requirements formally, as a set $\mathscr{T}_P$ of theorems which are necessary for the correctness of P. Hence the distinction between validation and verification begins to blur because all we can hope for is the formal verification of $\mathscr{T}_P$, which amounts to formal validation. Therefore it is already interesting to apply formal verification techniques to selected types of theorems, which, if they hold, will greatly increase our confidence in the system.

Both our projects are about finding bugs in existing systems. In practice, even the complete verification of a program P is less important than the discovery of program bugs, or errors. This is because the successful verification will only happen once, at the end of the development of P, whereas errors must be found during the entire development process leading up to the verification. The final verification, proving that P is free of errors of this type, is very nice to have, but in practice, due to the many additional informal requirements and parameters, it never means that P is totally correct [11]. So the real issue is debugging rather than verification in the pure sense.

Business objectives. Total correctness is practically never achievable, so the business objective is usually not total correctness. It is to make money by delivering a product which has the best quality that can be afforded. Since quality is important, elaborate quality assurance methods will already be in place (which dispels myth 10 that formal methods replace traditional ones). Formal methods are expensive to apply because they need highly trained personnel (confirming myth 4). If other methods are able to deliver the necessary quality, they will most likely be cheaper and win.

Business processes. Quality assurance is only one step in the business process of producing a product. Established successful business processes are extremely valuable and expensive to change because of many interdependent issues. At the same time they may only be laid down informally. It is very difficult to come up with a formal specification for even a part of such a business process, and it is almost impossible to change the process. New methods, such as formal methods, must be seamlessly integrated into the process and function with the established work force. As Craigen, Gerhart, and Ralston have observed [6]: "Industry will not abandon its current practices, but it is willing to augment and enhance its practices."

Time and efficiency. The time of all personnel integrated into a business process is exceedingly expensive. There is little spare time for experimentation. Formal methods must run very efficiently. While it may be of little concern how much time it takes to prove a new mathematical theorem (once), it is of great concern whether a human operator is delayed for minutes every time a formal method is activated in a development cycle. The industrial propositional formulae we faced were extremely large for BIS (hundreds of thousands of terms), but surprisingly 'harmless' for SAT checking. However, the formulae may be generated automatically from real data, so a great many of them must be handled efficiently. At one time, conversion to conjunctive normal form was a real problem for BIS until a new prover [16] was developed. We also parallelized our

[6] Baubarkeits-Informations-System (constructibility information system).

algorithms [30], but we did not yet apply parallelism in industry.

Meaningful explanation. Since debugging is the real issue rather than verification, even failed validations can be extremely useful, provided that they reveal costly errors that established processes fail to expose. Therefore we cannot take 'no' for an only answer. A failed proof is useful only if it can be explained in an intelligible way. It has been observed in this context that explanation is a sadly neglected area of automated deduction [6]. Furthermore, it must be possible to locate the corresponding defects in the source program readily from the explanation. Formulas representing real errors must be succinct and intelligible. For SA, we normalized them using BDDs; in BIS, they are way beyond what BDDs can handle, so a special explanation component was developed by A. Kaiser to find those (very few) constraints and variables which cause the huge constraint sets of BIS to become unsatisfiable [17].

Due to incomplete formalizations of the business process, there may be failed proofs that do not correspond to real (application) errors (false positives). Nobody has time to sift through reams of false positives. For both BIS and SA we had to go back and add extra axioms to our models to exclude false positives. False negatives (a failure to capture problems) can seriously undermine the credibility of formal methods, so only well debugged verification systems should be deployed; there is no time for experimentation and only a finite amount of good will out there.

7. Related work

There are two classes of related work, namely in Formal Methods in general, and in verifying expert systems in particular. Examples of the former have appeared in *IEEE Computer*, *IEEE Software*, and *IEEE Transactions on Software Engineering*, such as Refs. [4,6,11,26]. Craigen, Gerhart, and Ralston [6] point out that formal methods can be used to assure that code conforms; that a simple description of the semantics of the language is of relevance; that checkers of decidable fragments of theories are important; that feedback on failed proofs is sorely needed; and that formal methods need to augment industry's practices. These points are all confirmed by our experience, hence our emphasis on a simple logic with a highly efficient checker augmented by explanation components. We also find it highly interesting that we have found two industrial applications where propositional logic is genuinely used so that we could add SAT-based verification components with tolerable effort.

On the other hand, many different procedures for verifying rule based expert systems are proposed, covering a broad range of methods [8,10,23,24,27–29,41]. Unfortunately, the underlying semantics of the rules is not uniform: some systems interpret rules purely declarative as logical implications [25,28], whereas in other systems rules have an operational semantics, in which facts are added or removed from a pool of working knowledge (state space model) [8, 14,27]. Grumberg et al. consider an even more general (programming) system of (iterated) guarded commands [10].

Moreover, the inference mechanism is not standardized. Here, we want to point out just two differentiating aspects: forward-chaining vs. backward reasoning, and sequential vs. parallel rule execution. Whereas the former usually does not have a direct impact on rule semantics, the latter can induce considerable differences. In the parallel execution scheme the IF-statements of all rules are evaluated, and all matching actions are then executed in parallel. In the sequential setting, one of the matching actions is chosen non-deterministically. Induced differences on possible inconsistencies will be pointed out below.

Obviously—reflecting the diversity in expert systems' semantics—there is a huge variety of different error detection and verification methods. The types of errors under consideration, however, are relatively uniform, and it is gradually becoming common practice to classify them into four groups (see Refs. [22,28]): redundancy, conflict, incompleteness, and circularity errors.

Redundant rules (or parts of rules) can be removed from the expert system without affecting its deductive power. Redundant rules can arise because of duplication, subsumption, unnecessary parts in the IF statements, or more complicated forms such as chained redundancy [28].

Conflicts arise when contradictory facts can be derived from the knowledge base. In systems interpreting rules as logical implications conflicts directly correspond to logical inconsistencies. In the hypergraph approach of Ramaswamy *et al.* [28], where no negations can occur, additional consistency conditions (constraints) are used for expressing forbidden variable combinations. Systems employing the state space model reveal a different perspective of conflicts. In models with sequential execution, where negations correspond to the absence of facts in the working memory, no conflicts can occur, as it is impossible for a fact to be simultaneously present and not present. In this setting, inconsistencies are all due to additional integrity constraints on the variables—as is the case in the hypergraph approach. Parallel execution, on the other hand, can activate complementary actions, one erasing, and the other setting the same fact. Thus, conflicts between an action and its negation can directly occur in this setting.

Incompleteness deals with facts that the expert system cannot derive, but is supposed to do so. Obviously, this is more a validation than a verification issue. Often incompleteness is subdivided into three error categories [22,29]: missing rules, dead-ends (rules that produce no facts that are further needed) and unreachable goals (when the antecedent of a rule can never be satisfied).

Circularity is mainly understood in the same sense as in our paper, but in the purely declarative interpretation, this kind of circularity cannot occur. Therefore, in the rules-as-implications picture circularity usually refers to premises of a rule that can be derived by the rule system [28].

Among the rare industrial verifications of expert systems we want to mention the following: Spreeuwenberg *et al.* present a tool to verify knowledge bases built with Computer Associate’s Aion system [9]; they also treat real-life applications, for example for the Postbank Nederland BV’s assessment knowledge base [35]. Representative of many other similar projects, Hörl and Aichernig [15] formalized and verified a set of test cases for an air traffic voice communication system.

8. Conclusion

In this paper, we reported on an additional quality assurance effort involving formal methods on an industrial system close to shipment. By formalizing the IBM SA Automation Manager’s rule-based expert system we could prove a restricted non-looping property for a part of the rule system, and we could locate program bugs from failed proof attempts. After modelling the rule actions and consistency properties in ΔPDL, we converted them to a set of propositional SAT properties that current SAT-checking techniques can easily handle.

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.

In this work, we have found a case in industry where formal methods could be brought in very late to help debug an almost finished product. This was made possible because the program consists of a relatively abstract rule system given in terms of (almost) propositional logic. There was still a substantial, but manageable, formal modelling effort, described in Section 3 above. Since we mainly stayed within propositional (Boolean) logic, we had powerful industrial strength decision procedures at our disposal: advanced SAT-checking algorithms and normal form representations using BDDs. There is even some leeway here, because we did not have to resort to our parallel SAT-checker [30] yet. In those cases where our validation lemmas did not hold, we represented the error conditions as concise BDDs and could then trace the problem back from the model to error states of the original rule system. 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 (false positives).

We conclude that Formal Methods need not all be about specification, and that they can even be applied very late in industrial projects to debug, respectively validate, important aspects, provided the project already contains abstract interfaces from which the validation can proceed.

Appendix A. Proofs of lemmas and theorem

Lemma 1. *Let* $s_0 \overset{v=d}{\rightarrow} s_1$. *Then the following holds*:

(a) $s_0, s_1 \models \mathrm{RES_V}$
(b) $s_0 \models F_{v,d} \wedge \neg P_{v,d}$
(c) $s_1 \models P_{v,d}$
(d) $s_0 \models P_{w,e} \Leftrightarrow s_1 \models P_{w,e}$ *for all* $w \neq v$, $e \in D_w$.

Proof. (a) is obvious, as PRE contains $\mathrm{RES_V}$ as a conjunct, (b) follows from definition of $\tau(R_{v,d})$, (c) from property (P2) of atomic program $\alpha_{v,d}$. (d) is a consequence of (P3) and (P4) in conjunction with $\mathrm{RES_V}$. □

Lemma 2. *Let* $s_0 \overset{v=d}{\rightarrow} s_1$. *Then* $s_1 \models F$ *iff* $s_0 \models F|_{v=d}$.

Proof. We prove the lemma by induction on the structure of F. Assume F is atomic, i.e. $F = P_{w,e}$. We distinguish three cases: first, if $w \neq v$, then $P_{w,e}|_{v=d} = P_{w,e}$, and by Lemma 1(d) the claim holds. Second, if $w = v$ and $d = e$, then $P_{w,e}|_{v=d} = \top$, and by Lemma 1(c) we also have $s_1 \models P_{v,d}$. Third, if $w = v$, $d \neq e$, then $P_{w,e}|_{v=d} = \perp$, i.e. $s_o \not\models F|_{v,d}$. Also, by Lemma 1(a) and (c), we have $s_1 \not\models F$. The other cases are proved using the property that the restriction is a homomorphic extension of the atomic case. □

Lemma 3. *Let* $\vec{a} = (a_0, \ldots, a_{n-1})$ *be a loop candidate sequence. Then*

$$\mathrm{LOOP}(\vec{a}) := \mathrm{RES_V} \wedge \bigwedge_{\alpha_{v,d} \in L(\vec{a})} P_{v,d} \wedge \bigwedge_{0 \le i < n, a_i = \alpha_{v_i,d_i}} F_{v_i,d_i}|_{v_{i-1}=d_{i-1},\ldots,v_0=d_0} \tag{9}$$

is satisfiable iff there is a looping transition sequence in $T(\vec{a})$. *Moreover*, *the models of* $\mathrm{LOOP}(\vec{a})$ *are* (*starting*) *states of looping transition sequences.*

Proof. Let $\vec{a} = (a_0, \ldots, a_{n-1})$ with $a_i = \alpha_{v_i,d_i}$.
‘ ⇒ ‘: Assume $\mathrm{LOOP}(\vec{a})$ is satisfiable. Then there is a state s_0 with $s_0 \models \mathrm{LOOP}(\vec{a})$. We now construct a looping transition sequence $\vec{s} = (s_0, \ldots, s_{n-1})$ with $\vec{s} \in T(\vec{a})$, starting with s_0. Define

$$s_{i+1}(P_{w,e}) := \begin{cases} 1 & \text{if } w = v_i,\ e = d_i, \\ 0 & \text{if } w = v_i,\ e \neq d_i, \\ s_i(P_{w,e}) & \text{if } w \neq v_i. \end{cases}$$

We now show that Eqs. (8a)–(8f) hold for this $\vec{s}$.

(8a) By induction on i, using the recursive definition of s_i, and the fact that $s_0 \vDash \mathrm{RES_V}$.

(8b) As $\alpha_{v_{n-1},d_{n-1}}$ is obviously in $L(\alpha)$, we have $s_0 \vDash P_{v_{n-1},d_{n-1}}$.

(8c) $s_i(P_{v_{i-1},d_{i-1}}) = 1$ holds for $0 < i < n$ by definition of s_i.

(8e) Holds by definition of s_i, as $s_{i+1}(P_{w,e}) = s_i(P_{w,e})$ for $w \neq v_i$.

(8f) Consider any variable $w \neq v_{n-1}$. First, assume there are no a_l and d' with $a_l = \alpha_{w,d'}$. Then $s_0 \vDash P_{w,e} \Leftrightarrow s_1 \vDash P_{w,e} \Leftrightarrow \cdots \Leftrightarrow s_{n-1} \vDash P_{w,e}$ for all $e \in D_w$, which proves the claim. Otherwise, there is an $a_l \in L(\vec{a})$ with $a_l = \alpha_{w,d'}$ for some d'. The precondition $w \neq v_{n-1}$ excludes the case $l = n-1$. Therefore, we have $s_{l+1} \vDash P_{w,d'}$ by Eq. (8c) and $l < n-1$. By repeatedly using Eq. (8e), we get $s_k \vDash P_{w,d'}$ for all $l+1 \leq k \leq n-1$. $s_0 \vDash P_{w,d'}$ holds by definition of LOOP($\vec{a}$), so that $s_0 \vDash P_{w,e} \Leftrightarrow s_{n-1} \vDash P_{w,e}$ for all $e \in D_w$ by Eq. (8a).

(8d) We first prove that $s_i \vDash F_{v_i,d_i}$. By Lemma 2 we have $s_{i-1} \vDash F|_{v_{i-1}=d_{i-1}} \Leftrightarrow s_i \vDash F$ for all F and $0 < i < n$. So, by induction, $s_0 \vDash F|_{v_{i-1}=d_{i-1},\ldots,v_0=d_0} \Leftrightarrow s_i \vDash F$ for $0 < i < n$. $s_0 \vDash F_{v_0,d_0}$ obviously holds. Let us now turn to the proof of the second part, i.e. $s_i \vDash\neg P_{v_i,d_i}$. Let $J_1 = \{j < i | a_i = \alpha_{v,d'}$ for some $d'\}$ and $J_2 = \{j > i | a_i = \alpha_{v,d'}$ for some $d'\}$. First, assume that $J_1 \neq \emptyset$. Then let $j_m = \max\{J_1\}$. We now have $a_{j_m} = \alpha_{v,d'}$ with $d' \neq d$ because of Definition 2(2). So $s_{j_m+1} \vDash P_{v,d'}$, by Eq. (8e) $s_i \vDash P_{v,d'}$, and by Eq. (8a) $s_i \vDash\neg P_{v,d}$. Now, for the second case, assume $J_1 = \emptyset$. Then, as Definition 2(1) yields $J_1 \cup J_2 \neq \emptyset$, we get $J_2 \neq \emptyset$. Let $j_m = \max\{J_2\}$. Then $a_{j_m} = \alpha_{v,d'}$ with $d' \neq d$ because of Definition 2(2) and $J_1 = \emptyset$. Again, we have $s_{j_m+1} \vDash P_{v,d'}$, and by repeatedly using Eq. (8e), and once Eq. (8f), we get $s_i \vDash P_{v,d'}$. Thus, by Eq. (8a), $s_i \vDash\neg P_{v,d}$.

'$\Leftarrow$': Assume there is a looping transition sequence in $T(\vec{a})$, say $\vec{s} = (s_0, \ldots, s_{n-1})$. Then Eqs. (8a)–(8f) holds for $\vec{s}$ by Lemma 1. Now, because of Eq. (8a), $s_0 \vDash \mathrm{RES_V}$. Repeated application of Lemma 2, as above, in conjunction with Eq. (8d) yields $s_0 \vDash F_{v_i,d_i}|_{v_{i-1}=d_{i-1},\ldots,v_0=d_0}$ for $0 \leq i < n$. By repeated application of Eq. (8e) and finally Eq. (8f) we get $s_0 \vDash P_{v,d}$ for all $\alpha_{v,d} \in L(\vec{a})$. This proves $s_0 \vDash$ LOOP($\vec{a}$). □

Lemma 4. *Let $\vec{a}$ be an action sequence. If $T(\vec{a}) \neq \emptyset$, then $\vec{a}$ is a loop candidate sequence.*

Proof. As $T(\vec{a}) \neq \emptyset$, there is a sequence of states $(s_0, \ldots, s_{n-1} - 1) \in T(\vec{a})$ with $s_i \overset{v_i=d_i}{\longrightarrow} s_{i+1}$ for $0 \leq i \leq n-1$ and $s_{n-1} \overset{v_{n-1}=d_{n-1}}{\longrightarrow} s_0$. Assume now that $\vec{a}$ is not a loop candidate sequence. Then we can distinguish two cases:

(1) There is a variable v occurring only once in $\vec{a}$, say $a_i = \alpha_{v,d}$. By Eq. (8b) or (8c) we either have $s_{i+1} \vDash P_{v,d}$ if $i < n-1$ or $s_0 \vDash P_{v,d}$ if $i = n-1$. In both cases $s_j \vDash P_{v,d}$ for $n > j > i$ by Eq. (8e), and, moreover, by Eqs. (8f) and (8e) $s_k \vDash P_{v,d}$ for $k \leq i$. This is a contradiction to $s_i \vDash\neg P_{v,d}$, which follows from Eq. (8d).

(2) There are two actions $a_i = a_j = \alpha_{v,d}, i < j$, but there is (a) no index k with $i < k < j$ and $a_k = \alpha_{v,d'}$ for some d', or there is (b) no index l with either $l < i$ or $l > j$ such that $a_l = \alpha_{v,d''}$ for some d''. Let us first assume that (a) holds. Then, by Eq. (8c), we have $s_{i+1} \vDash P_{v,d}$. As there is no k with $i < k < j$ and $a_k = \alpha_{v,d'}$ for some d', we get by Eq. (8e) that $s_m \vDash P_{v,d}$ for $i < m \leq j$. But $s_j \vDash\neg P_{v,d}$ by Eq. (8d), a contradiction. Case (b) is handled similarly.

As a contradiction occurs in both cases, we can conclude that our assumption that $\vec{a}$ is not a loop candidate sequence is wrong. □

Theorem 1. LOOP($\vec{a}$) *is unsatisfiable for all loop candidate sequences $\vec{a}$ of length n iff the expert system contains no n-loops.*

Proof. '$\Rightarrow$': Let LOOP($\vec{a}$) be unsatisfiable for all loop candidate sequences $\vec{a}$ of length n. Now, assume that the expert system contains an n-loop. Then there is a looping transition sequence $\vec{s} = (s_0, \ldots, s_{n-1})$. Let $\vec{b} = (b_0, \ldots, b_{n-1})$ be the associated action sequence. By Lemma 4, as $\vec{s} \in T(\vec{b})$, $\vec{b}$ is a loop candidate sequence, and by Lemma 3 LOOP($\vec{b}$) is satisfiable, a contradiction.

'$\leftarrow$': Let the expert system contain no n-loops, i.e. there is no looping transition sequence of length n. Now assume, that LOOP($\vec{a}$) is satisfiable for some loop candidate sequence $\vec{a} = (a_0, \ldots, a_{n-1})$. Then, by Lemma 3, there is a looping transition sequence in $T(\vec{a})$ of length n, a contradiction. □

References

[1] R. Agarwal, M. Tanniru, A Petri-net based approach for verifying the integrity of production systems, Int. J. Man–Machine Studies 36 (1992) 447–468.

[2] E. Andert, Automated knowledge-base validation, Proceedings of AAAI Workshop on Verification and Validation of Expert Systems, July, 1992, pp. 122–127.

[3] A. Biere, A. Cimatti, E. Clarke, Y. Zhu, Symbolic model checking without BDDs, Tools and Algorithms for the Analysis and Construction of Systems (TACAS'99), Lecture Notes in Computer Science, vol. 1579, Springer, Berlin, 1999.

[4] J.P. Bowen, M.G. Hinchey, Seven more myths of formal methods, IEEE Soft. 12 (4) (1995) 34–41.

[5] R.E. Bryant, Symbolic boolean manipulation with ordered binary-decision diagrams, ACM Comput. Surv. 24 (3) (1992) 293–318.

[6] D. Craigen, S. Gerhart, T. Ralston, Formal methods reality check: Industrial usage, IEEE Trans. Soft. Engng 21 (2) (1995) 90–98.

[7] N. Fenton, S.L. Pfleeger, Can formal methods always deliver?, IEEE Comput. 30 (2) (1997) 34.

[8] R.F. Gamble, G.-C. Ball, H.C. Cunningham, Applying formal

verification methods to rule-based programs, Int. J. Expert Syst. 7 (3) (1994) 203–239.

[9] S. Garone, N. Buck, Capturing, reusing, and applying knowledge for competitive advantage: computer associate's aion. International Data Corporation, 2000. IDC White Paper.

[10] O. Grumberg, N. Francez, J.A. Makowsky, A proof rule for fair termination of guarded commands, Inform. Control 66 (1/2) (1985) 83–102.

[11] A. Hall, Seven myths of formal methods, IEEE Soft. 7 (5) (1990) 11–19.

[12] D. Harel, Dynamic logic, in: D. Gabbay, F. Guenthner (Eds.), Handbook of Philosophical Logic, Extensions of Classical Logic, vol. II, Kluwer Academic Publishers, Dordrecht, 1984, pp. 507–544.

[13] D. Harel, R. Sherman, Looping vs. repeating in dynamic logic, Inform. Control 55 (1–3) (1982) 175–192.

[14] F. Hayes-Roth, Rule based systems, Commun. ACM 28 (9) (1985) 921–932.

[15] J. Hörl, 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^{++}, FM'99—Formal Methods, II, Lecture Notes in Computer Science, vol. 1249, 1999, pp. 1868–1868.

[16] A. Kaiser, A SAT-based propositional prover for consistency checking of automotive product data. Technical report, Wilhelm-Schickard-Instimt für Informatik, Eberhard-Karls-Universität Tübingen, Sand 13, 72076 Tübingen, Germany, 2001. Technical Report WSI-2001-16.

[17] A. Kaiser, W. Küchlin, Explaining inconsistencies in combinatorial automotive product data, Proceedings of Second International Conference on Intelligent Technologies (InTech 2001) Bangkok, Thailand, November, Assumption University, 2001, pp. 198–204.

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

[19] H. Kautz, B. Selman, Planning as satisfiability, Proceedings of 10th European Conference on Artificial Intelligence (ECAI'92), Wiley, 1992, pp. 359–363.

[20] W. Küchlin, C. Sinz, Proving consistency assertions for automotive product data management, J. Autom. Reason., (Special Issue: Satisfiability in the Year 2000) 24 (1 and 2) (2000) 145–163.

[21] M.W. Moskewicz, C.F. Madigan, Y. Zhao, L. Zhang, S. Malik, Chaff: Engineering an efficient SAT solver, Proceedings of the 38th Design Automation Conference (DAC 2001), ACM, 2001, pp. 530–535.

[22] D.L. Nazareth, Issues in the verification of knowledge in rule-based systems, Int. J. Man–Mach. Stud. 30 (1989) 255–271.

[23] D.L. Nazareth, Investigating the applicability of Petri nets for rule-based system verification, IEEE Trans. Knowledge Data Engng 4 (3) (1993) 402–415.

[24] D.L. Nazareth, M.H. Kennedy, Verification of rule-based knowledge using directed graphs, Knowledge Acquisition 3 (1991) 339–360.

[25] T.A. Nguyen, W.A. Perkins, T.J. Laffey, D. Pecora, Checking a knowledge-based system for consistency and completeness/knowledge base verification, AI Magazine 8 (2) (1987) 69–75.

[26] S.L. Pfleeger, L. Hatton, Investigating the influence of formal methods, IEEE Computer 30 (2) (1997) 33–43.

[27] A.D. Preece, C. Grossner, T. Radhakrishnan, Validating dynamic properties of rule-based systems, Int. J. Human–Comp. Stud. 44 (2) (1996) 145–169.

[28] M. Ramaswamy, S. Sarkar, Y.-S. Chen, Using directed hypergraphs to verify rule-based expert systems, IEEE Trans. Knowledge Data Engng 9 (2) (1997).

[29] T.M. Shaft, R.F. Gamble, A theoretical basis for the assessment of rule-based system reliability, Found. Inform. Syst. July (2000) http://www.cba.uh.edu/~parks/fis/fis.htm.

[30] C. Sinz, W. Blochinger, W. Küchlin, PaSAT—parallel SAT-checking with lemma exchange: implementation and applications, in: H. Kautz, B. Selman (Eds.), LICS'2001 Workshop on Theory and Applications of Satisfiability Testing (SAT'2001), Electronic Notes in Discrete Mathematics, Boston, MA, vol. 9, Elsevier Science, 2001.

[31] C. Sinz, A. Kaiser, W. Küchlin, Detection of inconsistencies in complex product model data using extended propositional SAT-checking, in: I. Russell, J. Kolen (Eds.), Proceedings of 14th International FLAIRS Conference, Key West, FL, AAAI Press, 2001, pp. 645–649.

[32] C. Sinz, T. Lumpp, W. Küchlin, Towards a verification of the rule-based expert system of the IBM SA for OS/390 automation manager, Proceedings of the 2nd Asia–Pacific Conference on Quality Software (APAQS 2001), Hong Kong, IEEE Computer Society, 2001, pp. 367–374.

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

[34] I. Sommerville, Software Engineering, International Computer Science Series, fifth ed., Addison-Wesley, Harlow, England, 1997.

[35] S. Spreeuwenberg, R. Gerrits, M. Boekenoogen, VALENS: A Knowledge Based Tool to Validate and Verify an Aion Knowledge Base, ECAI 2000, 14th European Conference on Artificial Intelligence, vol. 10, IOS Press, 2000, pp. 731–735.

[36] The Standish Group International, Inc. Five "T's" of Data Base Availability, 1999. http://www.pm2go.com/sample_research/FiveTs.pdf.

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

[38] T.E. Uribe, M.E. Stickel, Ordered binary decision diagrams and the Davis–Putnam procedure, Proceedings of First International Conference on Constraints in Computational Logics, Lecture Notes in Computer Science, vol. 845, Springer, 1994.

[39] J. van Benthem, Modal Correspondence Theory, PhD Thesis, Institut voor Logica en Grondslagenonderzoek van Exacte Wetenschappen, University of Amsterdam, 1976.

[40] J. van Benthem, Correspondence theory, in: D. Gabbay, F. Guenthner (Eds.), Handbook of Philosophical Logic, Extensions of Classical Logic, vol. II, Kluwer Academic Publishers, 1984, pp. 167–247.

[41] R.J. Waldinger, M.E. Stickel, Proving properties of rule based systems, Int. J. Soft. Engng Knowledge Engng 2 (1) (1992) 121–144.

[42] H. Zhang, An efficient propositional prover, Proceedings of 14th International Conference on Automated Deduction (CADE-97), Lecture Notes in Computer Science, vol. 1249, Springer, 1997, pp. 272–275.