Verifying the On-Line Help System of SIEMENS Magnetic Resonance Tomographs

Carsten Sinz and Wolfgang Küchlin

Symbolic Computation Group, WSI for Computer Science, University of Tübingen and
Steinbeis Technology Transfer Center OIT, 72076 Tübingen, Germany
{sinz,kuechlin}@informatik.uni-tuebingen.de
http://www-sr.informatik.uni-tuebingen.de

Abstract. Large-scale medical systems—like magnetic resonance tomographs—are manufactured with a steadily growing number of product options. Different model lines can be equipped with large numbers of supplementary equipment options like (gradient) coils, amplifiers, magnets or imaging devices. The diversity in service and maintenance procedures, which may be different for each of the many product instances, grows accordingly. Therefore, instead of having one common on-line service handbook for all medical devices, SIEMENS parcels out the on-line documentation into small (help) packages, out of which a suitable subset is selected for each individual product instance. Selection of packages is controlled by XML terms. To check whether the existing set of help packages is sufficient for all possible devices and service cases, we developed the *HelpChecker* tool. *HelpChecker* translates the XML input into Boolean logic formulae and employs both SAT- and BDD-based methods to check the consistency and completeness of the on-line documentation. To explain its reasoning and to facilitate error correction, it generates small (counter-)examples for cases where verification conditions are violated. We expect that a wide range of cross-checks between XML documents can be handled in a similar manner using our techniques.

1 Introduction

There is a persistent trend towards products that are individually adaptable to each customer's needs (*mass customization* [Dav87]). This trend, while offering considerable advantages for the customer, at the same time demands special efforts by the manufacturer, as he now must make arrangements to cope with myriads of different product instances. Questions arising in this respect include: How can such a large set of product variants be represented and maintained concisely and uniquely? How can the parts be determined that are required to manufacture a given product instance? Is a certain requested product variant manufacturable at all? And, last but not least, how can the documentation—both for internal purposes and for the customer—be prepared adequately?

Triggered, among other reasons, by an increased product complexity, SIEMENS Medical Solutions recently introduced a formal description for their magnetic resonance tomographs (MR) based on XML. Thus, not only individual product instances, but also the set of all possible (*valid, correct*) product configurations can now be described by an

XML term which implicitly encodes the logical configuration constraints. This formal *product documentation* allows for an automated checking of incoming customer orders for compliance with the product specification. Besides checking an individual customer order for correctness, further tests become possible, including those for completeness and consistency of the on-line help system which are the topic of this paper. Similarly, cross-checks between the set of valid product instances and the parts list (in order to find superfluous parts) or other product attributes are within the reach of our method [SKK03].

In order to apply formal verification methods to an industrial process, the following steps are commonly necessary. First, a formal model of the process must be constructed. Second, correctness assertions must be derived in a formal language which is compatible with the model. Third, it must be proved mechanically whether the assertions hold in the model. Finally, those cases where the assertion fail must be explained to the user to make debugging possible. Throughout the formal process, speed is usually an issue, because in practice verification is often applied repeatedly as a formal debugging step embedded in an industrial development cycle.

In this paper we develop a formal semantics of the SIEMENS XML representation of their MR systems using propositional logic. This is accomplished by making the implicit assumptions and constraints of the tree-like XML representation explicit. We then translate different consistency properties of the on-line help system (help package overlaps, missing help packages) into propositional logic formulae, and thus we are able to apply automatic theorem proving methods (like SAT-checkers) in order to find defects in the package assignment of the on-line help system. Situations in which such a defect occurs are computed and simplified using Binary Decision Diagrams (BDDs). This exceeds the possibilities of other previously suggested checking techniques, as e.g. those of the XLinkIt system [NCEF02].

2 Product Documentation using XML

Product Structure. Many different formalisms have been proposed in the literature to model the structure of complex products [MF89,SW98,STMS98,MW98,KS00]. The method used by SIEMENS for the configuration of their MR systems was developed in collaboration with the first author of this paper and resembles the approach presented by Soininen *et al.* [STMS98]. Structural information is explicitly represented as a tree. This tree serves two purposes: first, it reflects the hierarchical assembly of the device, i.e. it shows the constituent components of larger (sub-)assemblies; and, second, it collects all available, functionally equivalent configuration options for a certain functionality. These two distinct purposes are reflected by two different kinds of nodes in the tree, as can be seen from the example in Fig. 1.

Type Nodes are employed to reflect the hierarchical structure, whereas *Item Nodes* represent possible configuration options with common functionality. From the example tree shown in Fig. 1 we may, e.g., conclude that there are two different possibilities for choosing a *System*: *Harmony* and *Concerto*. A *Harmony* system possesses three configurable (direct) subcomponents, of type *MPCU*, *Receiver*, and *Rx4*, respectively. The receiver, in turn, may be selected from the two alternatives *R-2* and *R-4*. Choos-

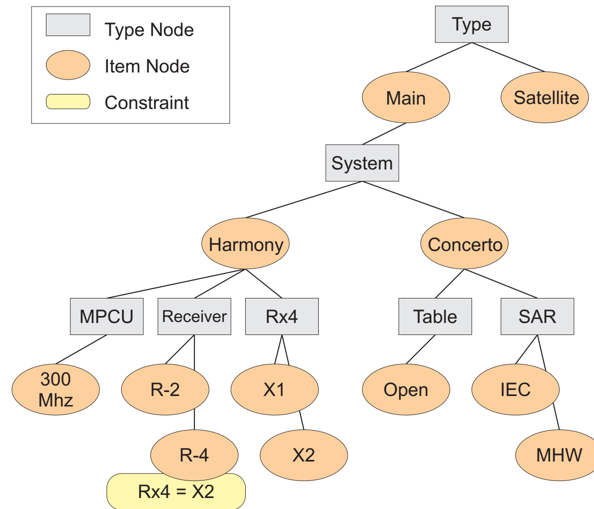

Fig. 1. Product Structure of Magnetic Resonance Tomographs (Simplified Example).

```
<Config auto-ns1:noNamespaceSchemaLocation="Config.xsd"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
   <Structure>
      <Type IDREF="INT_ConsoleType" MinOccurs="1" MaxOccurs="1">
         <Item IDREF="INI_ConsoleType_Sat"/>
         <Item IDREF="INI_ConsoleType_Main">
            <SubType IDREF="INT_System" MinOccurs="1" MaxOccurs="1">
               <!-- Harmony -->
               <Item IDREF="INI_System024">
                  <SubType IDREF="INT_Comp_MPCU" Default="INI_Comp_MPCU300"
                           ReadOnly="true" MinOccurs="1" MaxOccurs="1">
                     <Item IDREF="INI_Comp_MPCU300"/>
                  </SubType>
                  <SubType IDREF="INT_Comp_RXNumOf" Default="INI_Comp_RXNumOf1"
                           MinOccurs="1" MaxOccurs="1">
                     <Item IDREF="INI_Comp_RXNumOf1"/>
                     <Item IDREF="INI_Comp_RXNumOf2"/>
                  </SubType>
                  <SubType IDREF="INT_Comp_ReceiverNumOf" MinOccurs="1"
                           MaxOccurs="1">
                     <Item IDREF="INI_Comp_ReceiverNumOf2"/>
                     <Item IDREF="INI_Comp_ReceiverNumOf4">
                        <Conditions>
                           <Condition Type="INT_Comp_RXNumOf" Op="eq"
                                      Value="INI_Comp_RXNumOf2"/>
                        </Conditions>
                     </Item>
                  </SubType>
               </Item>
               <!-- Concerto -->  ...
            </SubType>
         </Item>
      </Type>
   </Structure>
</Config>
```

Fig. 2. Product Structure of Fig.1 in XML Representation (Excerpts).

ing the latter option puts an additional restriction on the configurable component *Rx4*: this has to be selected in its form *X2*. Each type node possesses additional attributes *MinOccurs* and *MaxOccurs* to bound the number of subitems of that type to admissible values. Assuming that for each type exactly one item has to be selected (i.e. *MinOccurs* = *MaxOccurs* = 1 for all type nodes), the configuration tree shown in Fig. 1 permits the following valid configuration (set of assignments):

$$\begin{array}{ccc} \text{Type} = \text{Main} & \text{MPCU} = \text{300MHz} & \text{Rx4} = \text{X2} \\ \text{System} = \text{Harmony} & \text{Receiver} = \text{R-4} & \end{array}$$

Within the SIEMENS system, the tree describing all product configurations is represented as an XML term. The term corresponding to the tree of Fig. 1 is shown in Fig.2. All XML terms are checked for well-formedness using XML Schemas [XML01].

We will use the simplified configuration example of this section throughout the rest of the paper for illustration purposes. The experiments of Section 4, however, were conducted on more complex and more realistic data sets.

Structure of On-Line Help. The on-line help pages that are presented to the user of an MR system may depend on the configuration of the system. For example, help pages should only be offered for those components that are in fact present in the system configuration. Moreover, for certain service procedures (e.g., tune up, quality assurance), the pages depend not only on the system configuration at hand, but also on the (workflow) steps that the service personnel already has executed. Thus, the help system is both configuration and workflow state dependent.

To avoid writing the complete on-line help from scratch for each possible system configuration and all possible workflow states, the whole help system is broken down into small *Help Packages* (see Fig. 3). A help package contains documents (texts, pictures, demonstration videos) on a specialized topic. The authors of the help packages decide autonomously about how they break down the whole help into smaller packages. So it is their own decision whether to write a whole bunch of smaller packages, one for each system configuration, or to integrate similar packages into one.

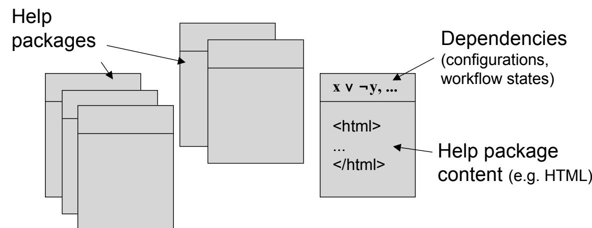

Fig. 3. Illustration of Help Packages: For each system configuration and workflow state a suitable help package has to be selected (controlled by dependencies).

Now, in order to specify the assignment of help packages to system configurations, a list of *dependencies* is attached to each help package, in which the author lists the system configurations and workflow states for which his package is suitable (see Fig. 4 for an example): all of a dependency's `RefType`/`RefItem` assignments must match in order to activate the package and to include it into the set of on-line help pages for that system. Multiple matching situations may be specified by associating further `Dependency`-elements to the package.

```
<Package ID="HLP_HP-1-181203-01-001" Name="HP-1-181203-01-001">
   <Content> ... </Content>
   <Dependencies>
      <Dependency>
         <RefType IDREF="INT_Workflow">
            <RefItem IDREF="INI_Workflow_TUNEUP"/>
         </RefType>
         <RefType IDREF="INT_System">
            <RefItem IDREF="INI_System_003"/>
         </RefType>
      </Dependency>
   </Dependencies>
</Package>

<Context>
   <RefType IDREF="INT_System">
      <RefItem IDREF="INI_System_003"/>
   </RefType>
   <RefType IDREF="INT_Workflow">
      <RefItem IDREF="INI_Workflow_TUNEUP"/>
   </RefType>
   <RefType IDREF="INT_WorkflowMode">
      <RefItem IDREF="INI_WorkflowMode_General"/>
   </RefType>
   <RefType IDREF="INT_WorkflowSfp">
      <RefItem IDREF="INI_WorkflowSfp_SfpTuncalOpen"/>
   </RefType>
</Context>
```

Fig. 4. Example of a Help Package (with Dependencies) and a Help Context.

The situations for which help packages must be available are specified by the engineering department using so-called *Help Contexts*. A help context determines system parameters and workflow steps for which a help package must be present. Examples for both a help package and a context (in XML representation) can be found in Fig. 4. The help package of this example fits any state of workflow *tune up* and all configurations of *System_003*. The example's context specifies that for step *TuncalOpen* in the *tune up* procedure of *System_003* a help package is required.

Currently, almost a thousand help contexts are defined for eleven MR systems, each with millions of different configuration possibilities. So, in spite of in-depth product knowledge, it is a difficult and time consuming task for the authors of help packages to find gaps (missing packages) or overlaps (ambiguities in package assignment) in the help system. To assist the authors, we therefore developed the *HelpChecker* tool, which is able to perform cross-checks between the set of valid system configurations, the situations for which help may be requested (determined by the contexts) and the situations for which help packages are available (determined by the packages' dependencies).

3 Logical Translation of Product Structure and Help System

To check the completeness and consistency of the on-line help system we need a translation into a logical formalism. We have chosen propositional logic for this purpose because of its relative simplicity and the presence of fast and elaborate decision procedures (SAT, BDD). We now lay down precisely what constitutes a consistent help system. Informally speaking, for each situation in which help may be requested for an existing system (and therefore a valid system configuration) there should be a matching help package. This means, help should be *complete*. Furthermore, to avoid possible ambiguities or even contradictions, there should be exactly one unique help package. This means, help should be *consistent*.

Therefore, we first have to find out which situations and product configurations can actually occur. We therefore develop a formalization of the product structure by building a configuration validity formula (ValidConf) describing the set of all valid configurations. The validity formula can automatically be derived from the XML data of the product structure and consists of consistency criteria for each of the structure's tree nodes. For a type node the following three validity conditions have to hold:

T1. The number of sub-items of the node must match the number restrictions given by the MinOccurs and MaxOccurs attributes.
T2. All selected sub-items must fulfill the validity conditions for item nodes.
T3. No sub-items may be selected that were not explicitly listed as admissible for this type.

For an item node the following three validity conditions have to hold:

I1. All sub-type nodes must fulfill the validity conditions for type nodes.
I2. The item's constraint, if present, has to be fulfilled.
I3. Unreferenced types and their items must not be used in the configuration. Types are considered unreferenced, if they do not appear as a subnode of the item.

We now informally define completeness and consistency of the on-line help system.

Definition 1. *The on-line help system is complete if, for each context, a matching help package exists. Only valid system configurations have to be considered.*

Remember that contexts specify situations (system configuration plus workflow state) for which help may be requested by the user. Thus the system has to make sure that for each such situation a help package is available.

To define consistency, we first need the notion of overlapping help packages:

Definition 2. *There is an overlap between two help packages ("ambiguity"), if there is a context and a valid system configuration for which both help packages' dependencies match (i.e., evaluate to true).*

Definition 3. *An on-line help system is consistent if there are no overlaps between help packages.*

In the next section we will give propositional criteria for these two properties. To build the link between XML terms and propositional logic, we will have to select subelements and attributes from an XML term. For this purpose we will use XPath [XPa02] expressions as shown in Table 1. The result of an XPath selection is always a set of XML nodes or a set of attributes. In Table 1, a stands for an arbitrary XML attribute and p for an arbitrary path, i.e. a list of XML elements separated by slashes (/).

Table 1. XPath Examples.

expression	denotation	example(s)
$/p$	absolute path	/Config/Structure
$p/..$	parent element	Type/Item/.. (= Type)
$p@a$	attribute selection	Item@MaxOccurs, SubType@IDREF

3.1 Formalization of the Product Structure

In this subsection we derive a propositional logic formula describing all valid system configurations. The variables of this formula stem from the XML specification's unique identifiers (ID and IDREF attributes) for types (InvType) and items (InvItem). A propositional variable is true for a given configuration if and only if the respective type or item is actually present in the configuration, i.e. is selected for the present product instance. Thus, the item-variables uniquely describe the system configuration, and a type variable is true if and only if at least one of its items occurs in the configuration.

Validity of a configuration:

$$\text{ValidConf} = \text{TypeDefs} \ \wedge \bigvee_{\substack{t \in \text{/Config/} \\ \text{Structure/Type}}} \text{ValConfT}(t) \qquad (\ddagger)$$

$$\text{TypeDefs} = \bigwedge_{\substack{t \in \text{/Inventory/} \\ \text{InvTypes/InvType}}} \Big(t@\text{ID} \ \Longleftarrow \bigvee_{i \in t/\text{InvItem}} i@\text{ID} \Big)$$

Formula ValidConf describes the set of all valid system configurations. A configuration is valid, if and only if it respects the type definitions (TypeDefs) and if it matches at least one configuration structure of the XML document, which is assured by the formula part around ValConfT(t). Whether a configuration matches a structure depends on the items of the configuration, on the tree-like assembly of the structure nodes as well as the constraints given in the item nodes. The type definitions are collected in a subformula (TypeDefs) that is responsible for a type variable to be set, as soon as at least one of its items is selected. As the MR system structure is recursively defined over tree nodes, the validity formulae are also recursive. In the same way, the distinction between type and item nodes is carried over to a distinction between validity formulae for type and item nodes. The overall validity of a configuration is determined by its top node in the tree, as can be seen from Formula (‡).

Validity of a type node:

$$\begin{aligned}
\text{ValConfT}(t) &= \text{CardinalityOK}(t) \wedge \text{SubItemsValid}(t) \\
&\quad \wedge \text{ForbidUnrefItems}(t) \\
\text{CardinalityOK}(t) &= \text{S}^{t@\text{MaxOccurs}}_{t@\text{MinOccurs}}(\{i@\text{IDREF} \mid i \in t/\text{Item}\}) \\
\text{SubItemsValid}(t) &= \bigwedge_{i \in t/\text{Item}} \Big(i@\text{IDREF} \Rightarrow \text{ValConfI}(i)\Big) \\
\text{ForbidUnrefItems}(t) &= \bigwedge_{i \in \text{unrefItems}(t)} \neg i@\text{IDREF}
\end{aligned}$$

A type node t is valid if and only if the three conditions (corresponding to T1-T3) of $\text{ValConfT}(t)$ hold. First, the number of selected items must match the MinOccurs and MaxOccurs attributes ($\text{CardinalityOK}(t)$). To express number restrictions, we use the selection operator S introduced by Kaiser [Kai01,SKK03]. $S^a_b(M)$ is true if and only if between a and b formulae in M are true. Second, the validity of all selected subitems of type t, i.e. those items i, for which $i@\text{IDREF}$ is true, must be guaranteed ($\text{SubItemsValid}(t)$). And, third, items that are not explicitly specified as sub-items of type node t are not allowed ($\text{ForbidUnrefItems}(t)$).

Validity of an item node:

$$\begin{aligned}
\text{ValConfI}(i) &= \text{SubTypesValid}(i) \wedge \text{ConditionValid}(i) \\
&\quad \wedge \text{ForbidUnrefTypes}(i) \\
\text{SubTypesValid}(i) &= \bigwedge_{t \in i/\text{SubType}} \text{ValConfT}(t) \\
\text{ConditionValid}(i) &= \begin{cases} \top & \text{if } i/\text{Conditions} = \emptyset, \\ \bigvee_{c \in i/\text{Conditions}} \\ \quad \bigwedge_{d \in c/\text{Condition}} \\ \qquad \text{DecodeOp}(d) & \text{otherwise} \end{cases} \\
\text{ForbidUnrefTypes}(i) &= \bigwedge_{t \in \text{unrefTypes}(i)} \neg t@\text{IDREF}
\end{aligned}$$

The validity of item nodes is defined in an analogous way. Again, three conditions (according to I1-I3) have to be fulfilled for an item node i to be valid. First, all sub-type nodes of item i have to be valid. Second, the item node's *Condition* XML-elements, if present, have to be fulfilled ($\text{ConditionValid}(i)$), where each *Condition* is a disjunction of conjunctions (DNF) of atomic equality or disequality expressions, as delivered by DecodeOp. And, third, unreferenced types, i.e. types that are not used beyond item node i, may not be used ($\text{ForbidUnrefTypes}(i)$).

Definitions of auxiliary expressions used in these formulae can be found in Appendix A.

3.2 Formalization of Help Package Assignment

To formalize the help package assignment we first need three basic formula definitions. Here, c and p are XML help context and help package elements, respectively.

Assignment of help packages:

$$\begin{aligned}
\text{HelpReq}(c) &= \bigwedge_{t \in c/\text{RefType}} \text{HelpTypeCond}(t) \\
\text{HelpProv}(p) &= \bigvee_{d \in p/\text{Dependencies}} \bigwedge_{t \in d/\text{Dependency}} \text{HelpTypeCond}(t) \\
\text{HelpTypeCond}(t) &= \begin{cases} \bigvee_{i \in t/\text{RefItem}} i@\text{IDREF} & \text{if } t/\text{RefItem} \neq \emptyset, \\ \bigvee_{\substack{i \in /\text{Inventory}/ \\ \text{InvTypes}/t/\text{InvItem}}} i@\text{ID} & \text{otherwise} \end{cases}
\end{aligned}$$

$\text{HelpReq}(c)$ defines for which situations, i.e. configurations and workflows, context c requires a help package, whereas $\text{HelpProv}(p)$ determines the situations for which help package p provides help. Now, completeness of the help system is equivalent to the validity of

$$\bigwedge_{c \in /\text{Help}/\text{Contexts}} \left(\text{HelpReq}(c) \wedge \text{ValidConf} \Rightarrow \bigvee_{p \in /\text{Help}/\text{Packages}} \text{HelpProv}(p) \right) . \qquad (*)$$

All cases which satisfy the negation of this formula are error conditions. Moreover, there is an overlap between help packages p_1 and p_2 if and only if

$$\bigvee_{c \in /\text{Help}/\text{Contexts}} \left(\text{HelpReq}(c) \wedge \text{ValidConf} \wedge \text{HelpProv}(p_1) \wedge \text{HelpProv}(p_2) \right) \qquad (**)$$

is satisfiable. Thus, the help system is consistent, if the latter formula is unsatisfiable for all help packages p_1 and p_2 with $p_1 \neq p_2$. All cases which satisfy this formula are thus error conditions.

4 Technical Realization and Experimental Results

Our implementation *HelpChecker* is a C++ program that builds on Apache's Xerces XML parser to read the SIEMENS product and help system descriptions. From these data, it generates Formulae (*) and (**). After having generated these formulae, it checks their satisfiability (in case of (*), it checks satisfiability of the negation). The intermediate conversion to conjunctive normal form (CNF) required by the SAT-checker is done using the well-known technique due to Tseitin [Tse70]. In case of an error condition, a formula is generated describing the set of situations in which this error occurs. This formula, call it F, is simplified by existential abstraction over irrelevant variables using standard BDD techniques (i.e. by replacing F by $\exists x F$ or, equivalently, by $F|_{x=0} \vee F|_{x=1}$ for an irrelevant propositional variable x).

HelpChecker is embedded into a larger interactive authoring system for the authors of help packages at SIEMENS. The system is not yet in everyday use, but still in the evaluation phase. Production use of the system is planned for the end of 2004. Thus, we do not have user feedback yet, and can only report on manually created (and thus possibly artificial) tests. All test cases were generated by SIEMENS documentation experts, though.

First experiments and timing measurements with the *HelpChecker* were conducted on a data set containing eleven lines of basic MR systems, 964 help contexts and twelve (dummy) help packages. To check the completeness and consistency of this data set, 35 SAT instances were generated (we use a fast approximative pre-test for package overlaps that filters out trivial cases). These SAT instances contained 1425 different propositional variables and between 11008 and 11018 clauses. Ten of them were satisfiable, corresponding to error cases (as can be seen from Equations (*) and (**) above), 25 unsatisfiable. One satisfiable instance corresponded to a missing help package, the other nine were due to package overlaps.

Initially, to check satisfiability we used a sequential version of our parallel SAT-checker PaSAT [SBK01]. Unsatisfiability could always be determined by unit propagation alone, the maximal search time for a satisfiable instance amounted to 15.90 ms on a 1.2 GHz Athlon running under Windows XP (80 branches in the Davis-Putnam algorithm, search heuristics MAX OCC). The current version of *HelpChecker* makes use of a BDD-based satisfiability checker developed at our department, and refrains from using a Davis-Putnam-style SAT-checker. The resulting BDDs contained up to 9715 nodes and 458 variables (with intermediate BDD sizes of over 100,000 nodes). A complete check of our experimental XML help—including computation of error situations—with eleven model lines, 964 help contexts and twelve help packages took 6.96 seconds. These input data already contain the complete configuration structure for all of SIEMENS' current MR model lines. However, the number of help packages is much lower than what we expect during production use.

In our context, speed is an important issue, because the authoring system will not allow a new help package to be checked in unless the *HelpChecker* reports an absence of all error conditions.

5 Related Work and Conclusion

Related Work. On the syntactic level, consistency checking of XML documents can be accomplished for the most part by XML Schema [XML01]. The appropriateness (i.e. consistency) of the semantic content, however, can not—or at most partially—be checked using these techniques. Among the different approaches to check the semantical consistency of XML documents are work on Java-based XML document evaluation by Bühler and Küchlin [BK01], the XLinkIt system by Nentwich *et al.* [NCEF02] and work on consistency checking of CIM models by Sinz *et al.* [SKKM03]. All these approaches differ considerably in the extent of expressible formulae and practically checkable conditions. From a logical point of view, none of these techniques exceeds an evaluation of first-order formulae in a fixed structure, which is not sufficient for our

application. In this respect, our method opens up new application areas for the discipline of XML checking.

For work on consistency checking of Boolean configuration constraints in the automotive industry see [KS00] and [SKK03].

Conclusion. In this paper we presented an encoding of the configuration and on-line help system of SIEMENS MR devices in propositional logic. Consistency properties of the on-line help system are expressed as Boolean logic formulae, checked by a SAT-solver, and simplified using BDD techniques. By using a Boolean encoding we are able to employ advanced SAT-solvers as they are also used, e.g., in hardware verification to efficiently check formulae with hundreds of thousands of variables.

Although we demonstrated the feasibility of our method only for the MR systems of SIEMENS Medical Solutions, we suppose that the presented techniques are also usable for other complex products. More generally, we expect that a wide range of cross-checks between XML documents can be computed efficiently using SAT-solvers.

References

[BK01] D. Bühler and W. Küchlin. A flexible similarity assessment framework for XML documents based on XQL and Java Reflection. In *Proc. 14th Intl. Conf. on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems (IEA/AIE 2001)*, LNAI, Budapest, Hungary, June 2001. Springer-Verlag.

[Dav87] S. M. Davis. *Future Perfect*. Addison-Wesley, 1987.

[Kai01] A. Kaiser. A SAT-based propositional prover for consistency checking of automotive product data. Technical report, WSI-2001-16, University of Tübingen, 2001.

[KS00] W. Küchlin and C. Sinz. Proving consistency assertions for automotive product data management. *J. Automated Reasoning*, 24(1–2):145–163, February 2000.

[MF89] S. Mittal and F. Frayman. Towards a generic model of configuration tasks. In *Proc. of the 11th Intl. Joint Conf. on Artificial Intelligence*, pages 1395–1401, Detroit, MI, August 1989.

[MW98] D.L. McGuiness and J.R. Wright. Conceptual modelling for configuration: A description logic-based approach. *AI EDAM*, 12(4):333–344, 1998.

[NCEF02] C. Nentwich, L. Capra, W. Emmerich, and A. Finkelstein. XLinkIt: A consistency checking and smart link generation service. *ACM Transactions on Internet Technology*, 2(2):151–185, May 2002.

[SBK01] C. Sinz, W. Blochinger, and W. Küchlin. PaSAT - parallel SAT-checking with lemma exchange: Implementation and applications. In H. Kautz and B. Selman, editors, *LICS'2001 Workshop on Theory and Applications of Satisfiability Testing (SAT'2001)*, volume 9 of *Electronic Notes in Discrete Mathematics*, Boston, MA, June 2001. Elsevier Science Publishers.

[SKK03] C. Sinz, A. Kaiser, and W. Küchlin. Formal methods for the validation of automotive product configuration data. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, 17(1):75–97, January 2003. Special issue on configuration.

[SKKM03] C. Sinz, A. Khosravizadeh, W. Küchlin, and V. Mihajlovski. Verifying CIM models of Apache web server configurations. In *Proc. of the 3rd International Conference on Quality Software (QSIC 2003)*, pages 290–297, Dallas, TX, November 2003. IEEE Computer Society.

[STMS98] T. Soininen, J. Tiihonen, T. Männistö, and R. Sulonen. Towards a general ontology of configuration. *AI EDAM*, 12(4):357–372, 1998.

[SW98] D. Sabin and R. Weigel. Product configuration frameworks – a survey. *IEEE Intelligent Systems*, 13(4):42–49, July/August 1998.

[Tse70] G. S. Tseitin. On the complexity of derivation in propositional calculus. In A. O. Silenko, editor, *Studies in Constructive Mathematics and Mathematical Logic*, pages 115–125, 1970.

[XML01] *XML Schema Parts 0–2: Primer, Structures, Datatypes. W3C Recommendation*, May 2001.

[XPa02] *XML Path Language 2.0. W3C Working Draft*, April 2002.

A Appendix: Auxiliary Definitions

$$\text{DecodeOp}(d) = \begin{cases} d\text{@Value} & \text{if } d\text{@Op} = \text{“eq”}, \\ \neg d\text{@Value} & \text{if } d\text{@Op} = \text{“ne”} \end{cases}$$

$$\text{unrefItems}(t) = \text{allItems}(t) \setminus t/\text{Item}$$

$$\text{allItems}(t) = \bigcup_{t'\text{@ID}=t\text{@IDREF}} /\text{Inventory/InvTypes/}t'/\text{InvItem}$$

$$\text{unrefTypes}(i) = (\text{refTypesT}(i/..) \setminus \{i/..\}) \setminus \text{refTypesI}(i)$$

$$\text{refTypesT}(t) = \{t\} \cup \bigcup_{i \in t/\text{Item}} \text{refTypesI}(i)$$

$$\text{refTypesI}(i) = \bigcup_{t \in i/\text{RefType}} \text{refTypesT}(t)$$

$$\text{S}_a^b(M) = \begin{cases} S^b(M) & \text{if } a = 0, \\ S^b(M) \wedge \neg S^{a-1}(M) & \text{otherwise} \end{cases}$$

$$\text{S}^b(M) = \bigwedge_{\substack{K \subseteq M \\ |K|=b+1}} \bigvee_{f \in K} \neg f$$