J Autom Reasoning (2006) 37:45–66
DOI 10.1007/s10817-006-9037-y

Checking Consistency and Completeness of On-Line Product Manuals

Carsten Sinz · Wolfgang Küchlin · Dieter Feichtinger · Georg Görtler

Abstract As products are growing more complex, so is their documentation. With an increasing number of product options, the diversity in service and maintenance procedures grows accordingly. This trend also holds for large-scale medical devices such as magnetic resonance (MR) tomographs. Siemens Medical Solutions has thus decided against one common on-line service handbook for all its MR tomographs. Instead, they fragment the on-line documentation into small packages, out of which a suitable subset is selected for each individual product instance. Selection of (so-called) *help packages* is controlled by XML terms encoding Boolean choice conditions. To assure that the set of available help packages is sufficient for all valid product instances, we developed a tool called *HelpChecker* that provides a transformation of XML terms to propositional logic formulas and then employs BDD-based methods to ascertain completeness of the on-line documentation and to support authors in locating any gaps. Experiments with SAT-Solvers were also made.

Key words real-world applications · problem encoding · BDD-techniques · SAT.

C. Sinz (✉)
Institute for Formal Models and Verification, Johannes Kepler University Linz,
Altenbergerstr. 69, 4040 Linz, Austria
e-mail: carsten.sinz@jku.at

W. Küchlin
Steinbeis Technology Transfer Center OIT, Eberhard Karls University Tübingen, Sand 13,
72076 Tübingen, Germany
e-mail: kuechlin@informatik.uni-tuebingen.de

D. Feichtinger · G. Görtler
Siemens Medical Solutions, Hartmannstr. 16, 91052 Erlangen, Germany

D. Feichtinger
e-mail: dieter.feichtinger@siemens.com

G. Görtler
e-mail: georg.goertler@siemens.com

1. Introduction

There is a persistent trend toward products that are individually adaptable to each customer's needs (*mass customization* [7]). This trend, while offering considerable advantages for the customer, at the same time demands special efforts by the manufacturer, who now must make arrangements to cope with myriad different product instances. Questions arising in this context 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 – the question we specifically address in this paper – how can the accompanying documentation such as service and user manuals be customized consistently with the product configuration?

Triggered – among other reasons – by an increased product complexity, Siemens Medical Solutions recently introduced a semi-formal description for their magnetic resonance (MR) tomographs 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 that 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 this method [28].

In order to apply automated reasoning to an industrial process, the following steps are commonly necessary [31]. First, a formal model of the process must be constructed. Second, correctness assertions must be derived in a formal language that is compatible with the model. Third, it must be proved mechanically whether the assertions hold in the model. Fourth, 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 [30].

In this paper we develop a formal semantics for the XML representation of the Siemens MR systems using propositional logic. This is accomplished by making the implicit assumptions and constraints of the semi-formal XML representation explicit. We then translate different consistency properties of the on-line help system (help package overlaps, missing help packages) into propositional logic formulas, and thus we are able to apply automatic theorem-proving methods 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 by using binary decision diagrams (BDDs). This approach exceeds the possibilities of previously suggested XML checking techniques, for example, those of the XLinkIt system [23].

2. Product Configuration with XML

2.1. Product Structure

Many different formalisms have been proposed in the literature to model the structure of complex products [14, 19, 21, 25, 34]. 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. [34]. Structural information is explicitly represented as an AND–OR tree. This tree serves two purposes. First, it reflects the hierarchical assembly of the device; that is, it shows the constituent components of larger (sub)assemblies (indicated by solid lines in Figure 1). Second, it collects all available, functionally equivalent (or similar) configuration options for a particular functionality (indicated by dashed lines). The latter can also be regarded as an *is–a* relationship, whereas the former expresses a *has–a* relationship. These two distinct purposes are also reflected by two different kinds of nodes in the tree, as can be seen from the example in Figure 1.

Type nodes (OR-nodes) reflect a common type that all their direct child nodes have in common. Typically exactly (or sometimes at least) one of the child nodes has to be selected in a valid configuration (thus, OR-node). So, for example, the *Receiver* node gathers all available nodes of type receiver (*R-2* and *R-4*) and indicates that exactly one of them has to be selected. *Item nodes* represent concrete configuration items (e.g., parts or assemblies). The child nodes of an item node are the subassemblies that are required for the item to be complete. All of them have to be present in a valid configuration (thus, AND node). So, for example, system *Harmony* requires three subassemblies: one of type *MPCU*, one of type *Receiver*, and one of type *Rx4*.

From the example tree shown in Figure 1 we can therefore, for example, conclude that there are two different possibilities for choosing a *System*: *Harmony* and *Concerto*. A *Harmony* system possesses three configurable (direct) subcomponents: type *MPCU*, *Receiver*, and *Rx4*. The receiver, in turn, may be selected from the two options: *R-2* and *R-4*. Choosing the latter option puts an additional restriction on the configurable component *Rx4*: this has to be selected in its form *X2* if *R-4* is selected. Each type node possesses two additional attributes, *MinOccurs* and *MaxOccurs*, to bound the number of admissible subitems of that type. If we assume that for each type exactly one item has to be selected (i.e., MinOccurs = MaxOccurs = 1 for all

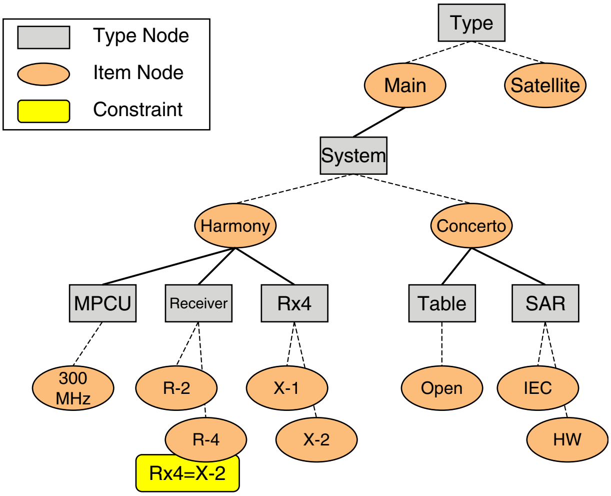


Figure 1 Product structure of magnetic resonance tomographs (simplified example).

type nodes), the configuration tree shown in Figure 1 permits the following valid configuration (regarded as a set of items assigned to types):

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

A particular system configuration is completely determined by a complete set of assignments, that is, a set where all cardinality constraints (*MinOccurs* and *MaxOccurs*) are satisfied. As the item names are unique and each item can be selected at most once, a configuration is already determined by its set of items. This alleviated translation to propositional logic considerably, as thus each item can be considered as a propositional variable and a system configuration corresponds to an assignment to these variables.

Within the Siemens system, the tree describing all product configurations is represented as an XML term. The term corresponding to the tree of Figure 1 is shown in Figure 2. The XML terms reflect the tree structure almost one-to-one. There is a Type element for each type node, and an Item element for each item node of the tree. In an *Inventory* (not shown in Figure 2) all possible node names are stored via ID attributes. These can then be referenced in the configuration structure via IDREF attributes. So, for example, the Receiver node name is indicated by

```
<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_System_024">
                  <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>
```

Figure 2 Excerpts of the XML representation corresponding to the product structure shown in Figure 1.

the IDREF attribute INT_Comp_ReceiverNumOf. Moreover, cardinality constraints on the number of admissible subitems of a type node can be specified using the two attributes MinOccurs and MaxOccurs. Conditions are expressed in the form | < `Type` >< `Op` >< `Value` > |, where |Op| must be one of ‘eq’ or ‘ne’, indicating equality or inequality, respectively. A condition T eq V requires that the item with name V is selected for the type with name T in a valid configuration, whereas T ne V requires that item V is not selected. All XML terms are checked for well-formedness by using XML Schema [37].

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, realistic data.

2.2. Structure of On-Line Help

The on-line help pages that are presented to the user of an MR system depend on the configuration of the system. For example, help pages are offered only for those components that are in fact present in the system configuration. Moreover, for certain service procedures (e.g., tuneup, quality assurance), the accessible pages depend not only on the system configuration at hand but also on the (workflow) steps that the service personnel already has executed and on the level of knowledge of the user. Workflows are specified as finite transition systems, where states are labeled with properties denoting, for example, the current action the user has to perform or his knowledge level. Consequently, the help system depends not only on the system configuration but also on further parameters like the workflow state.

In order 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 Figure 3). A help package contains documents (texts, pictures, demonstration videos) on a specialized topic. The authors of the help packages decide autonomously how they break down the whole help (i.e., the material for all manuals) into smaller packages. Hence, it is their own decision whether to write a collection of smaller packages, one for each system configuration, or to integrate similar packages into one.

In order to specify the assignment of help packages to system configurations and workflow states, a list of *dependencies* is attached to each help package, in which the

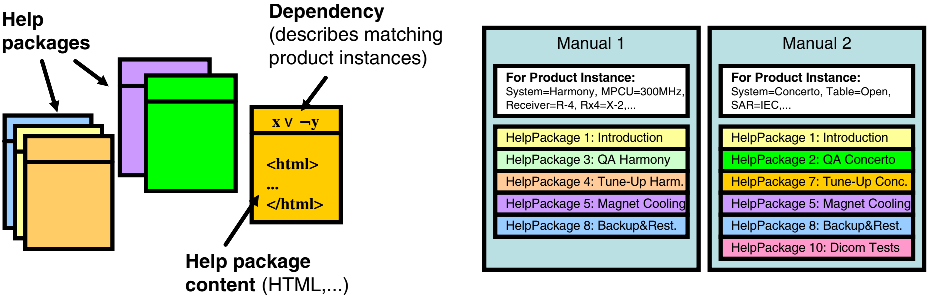


Figure 3 Illustration of help packages: for each system configuration a suitable help package has to be selected (controlled by dependencies; workflows not shown).

author lists the situations for which his package is suitable (see Figure 4, top part, for an example): all of a dependency's `RefType/RefItem` assignments must match in order to activate the package and to include it in the set of on-line help pages for that system. So, for example, the package of Figure 4 is selected for all situations in which INT_System = INI_System_003 and INT_Workflow = INI_Workflow_TUNEUP. Multiple matching situations (e.g., for either a special coil or a special receiver) may be specified by associating further `Dependency` elements with a package.

The situations for which help packages must be available are specified by the engineering department by using *help contexts*. A help context determines system parameters and workflow steps for which a help package must be present. An example of a help context (in XML representation) can be found in Figure 4 on the bottom. The help package of this example fits any state of workflow *tuneup* (in which system parameters are optimized by the maintenance personnel) and all configurations of *System_003*. The example's context specifies that for step *TuncalOpen* in the *tuneup*

```
<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>
```

Figure 4 Example of a help package (with dependencies) and a help context.

procedure of *System_003* a help package is required if the *workflow mode* is set to *general*.

Currently, more than a thousand help contexts are defined for 11 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 documents. To assist the authors, we therefore developed an add-on tool, called *HelpChecker*, 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 Encoding of Product Structure and Help System

To check 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 simplicity and the presence of fast and elaborate decision procedures (SAT, BDD). Encoding in a description logic [1] would also have been possible, but because of lack of experience in using description logics for large-scale projects, we decided against this approach.

We now present 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. That is, help should be *complete*. Furthermore, in order to avoid possible ambiguities or even contradictions, there should be exactly one unique help package. That is, 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 be derived automatically from the XML data of the product structure and consists of consistency criteria for each of the product structure's tree nodes.

For a *type node* the following three validity conditions have to be met:

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

In our example, condition T3 would thus ensure that choosing a *Receiver* for a *Concerto* system resulted in an invalid configuration.

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

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

In our example, I3 would ensure that below the *Satellite* node no further types may be used.

We now give (still informal) definitions for completeness and consistency of the on-line help system that we will use later.

DEFINITION 3.1. 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 3.2. There is an overlap between two help packages (‘ambiguity’) if there exist a context and a valid system configuration for which both help packages' dependencies match (i.e., evaluate to true).

DEFINITION 3.3. 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 XML terms. For this purpose we will use XPath [38] expressions (in *abbreviated syntax*) as shown in Table I. The result of an XPath selection is always a set of XML nodes (in the case of a path selection) or an attribute (in the case of an attribute selection). We assume that attributes are always defined (which is ascertained by XML Schemas). So, for example, the expression /Config/Structure/Type selects all XML nodes that are reached when following each path Config→Structure→Type from the root node of the XML document. In Table I, *a* stands for an arbitrary XML attribute and *p* for an arbitrary path, that is, a list of XML elements separated by slashes (/).

3.1. Formalization of the Product Structure

We now derive a propositional logic formula describing all valid system configurations, which means that the models of this formula are exactly the valid system configurations. The variables occurring in this formula stem from the XML specification's unique identifiers (ID and IDREF attributes) for types (XML element InvType) and items (InvItem). Each identifier is a character string in the XML representation and is bijectively mapped to a propositional variable in our encoding. The interpretation

Table I Path examples as used in the propositional logic translation.

Expression	Denotation	Example(s)
/*p*	Absolute path	/Config/Structure
p/..	Parent element	Type/Item/.. (= Type)
p@*a*	Attribute selection	Item@MaxOccurs, SubType@IDREF

of propositional variables is as follows: A variable is true for a given configuration if and only if the respective type or item is present in the configuration, that is, if and only if it is selected for the present product instance. Thus, item-variables uniquely describe the system configuration (as mentioned in Section 2.1), and a type-variable is true if and only if at least one of its items occurs in the configuration. We now gradually derive this configuration validity formula.

Validity of a configuration:

$$\begin{aligned}
\text{ValidConf} &= \text{TypeDefs} \wedge \text{TypeAliases} \\
&\quad \wedge \text{ConfigStructure} \wedge \text{ForbidGlobalUnrefTypes} \\
\text{TypeDefs} &= \bigwedge_{\substack{t \in /\text{Inventory}/\\ \text{InvTypes/InvType}}} \Bigg(\Bigg(\bigvee_{i \in t/\text{InvItem}} i@\text{ID} \Bigg) \Rightarrow t@\text{ID} \Bigg) \\
\text{TypeAliases} &= \bigwedge_{\substack{t \in /\text{Inventory}/\\ \text{InvTypes/InvTypeAlias}}} \Big(t@\text{ID} \Leftrightarrow t@\text{Base} \Big) \\
\text{ConfigStructure} &= \bigvee_{\substack{t \in /\text{Config}/\\ \text{Structure/Type}}} \text{ValConfT}(t) \\
\text{ForbidGlobalUnrefTypes} &= \bigwedge_{t \in \text{globalUnrefTypes}} \neg t@\text{IDREF}
\end{aligned}$$

Formula ValidConf describes the set of all valid system configurations. A configuration is valid if and only if it respects the type definitions (TypeDefs), type aliases are set up correctly (TypeAliases), no defined but unreferenced types are used (ForbidGlobalUnrefTypes), and it matches at least one configuration structure of the XML document (ConfigStructure). The last condition is assured by the big disjunction over ValConfT(t), which means that for each valid configuration the top node t of at least one configuration structure must satisfy ValConfT(t).[1] As the MR system structure is defined recursively over tree nodes (cf. Figure 1), the validity formulas (ValConfT and ValConfI) are also recursive. The distinction between type and item nodes in the XML model is also carried over to a distinction between validity formulas for type and item nodes.

The type definitions specified in subformula TypeDefs ensure that a type variable is set as soon as at least one of its items is selected. (The Inventory contains a list of all possible types together with a list of possible items for each of them.) This approach simplifies the definition of unreferenced types.

Type aliases are used to define alternative names (stored under attribute ID) for existing types (stored under attribute Base) within the XML product structure. The correct mapping of alias types to their base types is assured by formula TypeAliases.

Turning back to our example, and assuming an inventory specifying three items INI_System_024, INI_System_005 and INI_System_007 (for systems *Harmony*,

---

[1] Variables i and t are assumed to range over XML nodes here. Attributes like t@ID are identified with propositional variables.

Avanto, and *Concerto*, respectively) of type INT_System, we obtain the following TypeDefs formula for this particular type:

$$\begin{aligned}(\text{INI\_System\_024} \vee \text{INI\_System\_005} \vee \text{INI\_System\_007})\\ \Rightarrow \text{INT\_System}\ ,\end{aligned}$$

by which the type-variable INT_System is set as soon as any items of this type are set. The formula ForbidGlobalUnrefTypes excludes all types occurring in the inventory but not in the configuration tree structure. So if we had two types INT_Coil and INT_Country in the inventory, which do not show up in the configuration tree structure, we obtain the formula

$$\neg\text{INT\_Coil} \wedge \neg\text{INT\_Country}$$

for ForbidGlobalUnrefTypes, which forbids the use of these types (and by formula TypeDefs also their items) in any configuration.

Validity of a type node: [2]

$$\begin{aligned}
\text{ValConfT}(t) &= \text{CardinalityOK}(t) \wedge \text{SubItemsValid}(t)\\
&\quad \wedge\text{ForbidUnrefItems}(t)\\
\text{CardinalityOK}(t) &= \begin{cases} \mathrm{S}_1^1(\{i@\text{IDREF} \mid i \in t/\text{Item}\}) & \\ \qquad\qquad \text{if } t@\text{CheckMode} = \text{ExactlyOne} & \\ \mathrm{S}_{t@\text{MinOccurs}}^{t@\text{MaxOccurs}}(\{i@\text{IDREF} \mid i \in t/\text{Item}\}) & \\ \qquad\qquad\qquad\qquad \text{otherwise} & \end{cases}\\
\text{SubItemsValid}(t) &= \bigwedge_{i\in t/\text{Item}} \Big(i@\text{IDREF} \Rightarrow \text{ValConfI}(i)\Big)\\
\text{ForbidUnrefItems}(t) &= \bigwedge_{i\in \text{unrefItems}(t)} \neg i@\text{IDREF}
\end{aligned}$$

A type node t is valid if and only if the three conditions (corresponding to T1-T3) of ValConfT(t) hold. First, the number of selected items must match the MinOccurs and MaxOccurs attributes (CardinalityOK(t)) of the type node. There is one exception – when the type node's CheckMode attribute is set to ExactlyOne – in which case an explicit number restriction of exactly one is assumed. The reason for this exceptional handling is explained in Section 5. To express number restrictions, we use the selection operator S introduced by Kaiser [13, 28]. $S_b^a(M)$ is true if and only if between a and b formulas in M are true. Second, the validity of all selected subitems of type t, that is, those items i, for which i@IDREF is true, must be guaranteed (SubItemsValid(t)). Third, items that are not explicitly specified as subitems of type node t are not allowed (ForbidUnrefItems(t)).

Expanding these definitions for the *Receiver* type node of our example (which can be found under t_R = /Config/Structure/.../INT_SubType[@IDREF = 'INT_*Comp_ReceiverNumOf*']) and using R_2 and R_4 as abbreviations for the IDREFs

---

[2] Definitions for auxiliary expressions used in these formulas but not defined here can be found in the Appendix.

of the subitems of t_R, that is, INI_Comp_ReceiverNumOf2 and INI_Comp_Receiver NumOf4, we obtain the following:

$$\begin{aligned}
\text{CardinalityOK}(t_R) &= S_1^1(\{R_2, R_4\}) \\
&= S^1(\{R_2, R_4\}) \wedge \neg S^0(\{R_2, R_4\}) \\
&= (R_2 \vee R_4) \wedge \neg(R_2 \wedge R_4) \\
\text{SubItemsValid}(t_R) &= (R_2 \Rightarrow \text{ValConfI}(i_1)) \wedge (R_4 \Rightarrow \text{ValConfI}(i_2)) \\
\text{ForbidUnrefItems}(t_R) &= \top
\end{aligned}$$

Here, i_1 and i_2 are the paths to the two subitems of the *Receiver* type node. The first formula ascertains that the cardinality constraint is satisfied, the second formula ascertains that the subitems are valid if they are selected, and the third formula forbids items of type *Receiver* that occur in the inventory but are not subitems of node t_R. As there are no such subitems, the conjunction is over the empty set and thus equivalent to true.

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 subtype nodes of item i have to be valid. Second, the item node's *Condition* XML-elements, have to be satisfied (ConditionValid(i)) if present, where each *Condition* is a disjunction of conjunctions (DNF) of atomic equality ($=$) or disequality ($\neq$) expressions, as delivered by DecodeOp. Third, unreferenced types, that is, types that are not used beyond item node i, may not be used (ForbidUnrefTypes(i)).

Considering item *R-4* (named INI_Comp_ReceiverNumOf4 in the XML file) of our example and denoting the path to this node by i_{R4}, we obtain the following formulas:

$$\begin{aligned}
\text{SubTypesValid}(\mathrm{i_{R4}}) &= \top \\
\text{ConditionValid}(\mathrm{i_{R4}}) &= \text{INI\_Comp\_RXNumOf2} \\
\text{ForbidUnrefTypes}(\mathrm{i_{R4}}) &= \top
\end{aligned}$$

The first formula checks subtypes for validity, which is trivial, since node i_{R4} possesses no subtypes. The second formula ascertains that the node's condition is valid, which simply enforces INI_Comp_RXNumOf2 to be set to true. The third formula forbids unreferenced types, which is also trivially true.

3.2. Formalization of Help Package Assignment

To formalize the help package assignment, we first define three basic properties. Within these definitions, c and p are XML help context and help package elements, respectively.

Assignment of help packages:

$$\mathrm{HelpReq}(c) = \bigwedge_{t \in c/\mathrm{RefType}} \mathrm{HelpTypeCond}(t)$$

$$\mathrm{HelpProv}(p) = \bigvee_{d \in p/\mathrm{Dependencies}} \bigwedge_{t \in d/\mathrm{Dependency}} \mathrm{HelpTypeCond}(t)$$

$$\mathrm{HelpTypeCond}(t) = \begin{cases} \neg\mathrm{HelpTypeSubCond}(t) & \text{if } t\text{@Negate} = \text{true} \\ \mathrm{HelpTypeSubCond}(t) & \text{otherwise} \end{cases}$$

$$\mathrm{HelpTypeSubCond}(t) = \begin{cases} \bigvee_{i \in t/\mathrm{RefItem}} i\text{@IDREF} & \text{if } t/\mathrm{RefItem} \neq \emptyset, \\ \bigvee_{i \in \mathrm{allItems}(t)} \neg i\text{@IDREF} & \text{otherwise} \end{cases}$$

HelpReq(c) defines for which situations (i.e., combinations of configurations and workflows) context c requires a help package, whereas HelpProv(p) determines the situations for which help package p provides help. Situations are implicitly specified (in the XML representation) as formulas in a generalized conjunctive normal form (CNF) in the case of help contexts and as disjunctions of generalized CNFs in the case of help package dependencies. The latter leaves even more freedom to write down constraints. In a generalized CNF, each clause may also be negated (indicated by the Negate attribute). If a RefType has no subitems in a context or dependency specification, this is interpreted as a situation in which none of the items of this type are present.

Considering our example, we obtain the following formulas (denoting the help package and context paths by p_1 and c_1, respectively):

$$\begin{aligned} \mathrm{HelpReq}(c_1) &= s_{003} \wedge w_{\mathrm{TUNEUP}} \wedge wm_{\mathrm{General}} \wedge ws_{\mathrm{SfpTuncalOpen}} \\ \mathrm{HelpProv}(p_1) &= w_{\mathrm{TUNEUP}} \wedge s_{003} \end{aligned}$$

Here we have used abbreviations for the Boolean variables, for example, s_{003} for INI_System_003.

3.3. Consistency Criteria

With these definitions, we are now in a position to give propositional logic formulas corresponding to completeness and consistency of the help system.

Completeness of the help system is equivalent to validity of formula COMP defined as

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

Thus, for completeness to hold, there must be a matching help package for each situation that belongs to a help context and describes a valid configuration. Situations for which the formula does not hold are error conditions that can be reported by the *HelpChecker*.

Let us now turn to consistency with respect to package overlaps: There is an overlap between help packages p_1 and p_2 if and only if formula $\mathrm{Overlap}(p_1, p_2)$ defined as

$$\bigvee_{c \in \text{/Help/Contexts}} \Big(\mathrm{HelpReq}(c) \wedge \mathrm{ValidConf} \wedge \mathrm{HelpProv}(p_1) \wedge \mathrm{HelpProv}(p_2)\Big)$$

is satisfiable. Thus, there is an overlap between packages p_1 and p_2 if there is a situation that at the same time describes a valid configuration, belongs to a help context, and selects both packages p_1 and p_2 simultaneously. If no such situation exists, that is, if formula CONS defined as

$$\bigwedge_{\substack{p_1, p_2 \in \text{/Help/Packages} \\ p_1 \neq p_2}} \neg\mathrm{Overlap}(p_1, p_2)$$

is valid, then the help system is *consistent*. Again, all cases in which this condition is violated are error situations that can be reported by the *HelpChecker*.

4. Technical Realization and Experimental Results

Our implementation, called *HelpChecker*, is a C++ program that builds on Apache's Xerces XML parser to read the Siemens product configuration and help system description (package dependencies and contexts). From this data, it generates formulas COMP and CONS. These formulas are then negated and transformed into BDDs [3].[3] More exactly, one BDD is generated for formula COMP and one BDD for each pair of packages for formula CONS (i.e., we generate the Overlap formulas explicitly; however, we avoid trivial cases in this step). By using the negation, the models of the BDD correspond one-to-one to error situations. This BDD, call it E, is simplified by existential abstraction over irrelevant variables using standard BDD techniques (i.e., by replacing E by $\exists x E$ or, equivalently, by $E|_{x=0} \vee E|_{x=1}$ for an irrelevant propositional variable x). Irrelevant variables are those variables that do not occur in any help context or help package dependency, but only in the ValidConf part of the test formula (these are internally used variables on details of the MR tomographs that are of no relevance to the help package authors). Then, the whole set of error situations (i.e., all models of the simplified BDD) is dumped into a result file in XML format. Generating all error situations at once is important because the authors of help packages are not supposed to make incremental runs removing one error situation after the other, but prefer to have – at a glance – a complete indication of where residual errors remain.

As an optimization in order to further speed error detection, we partition both the set of help packages and the set of help contexts based on the (typically unique) workflow items these are associated with. This is especially useful for the test on package overlaps, where the quadratic number of pairs of help packages can be reduced considerably.

---

[3] Whenever we talk of BDDs we mean reduced ordered BDDs. We use a BDD package developed by the first author of this paper. One of the main design goals of the package was to reduce memory consumption (compared to other BDD implementations). We have not yet made a comparison with other BDD packages such as CUDD [35].

We conducted a series of experiments and timing measurements with the *HelpChecker* on different test data sets provided by the Siemens MR department. These data sets contained between three and 11 model lines of basic MR systems, between 77 and 3871 help contexts and between zero and 928 help packages (see Table II). The 11 systems in test case E-T6-4 have been only partially specified, whereas the systems of the other test cases already contain the complete system description. (The names of the test cases are abbreviations of internally used filenames, partly containing submitter names, system versions and encoded dates.) Context specifications are complete only for the test cases E-U-805, E-U-140-1, and E-F-405. For all but the last test case (E-H-306), real help packages are not yet available. We therefore used dummy help packages that were provided by the Siemens documentation department. The size of the larger data sets (e.g., E-F-405) is, however, already comparable to the sizes finally to be expected.

Test runs of the *HelpChecker* were performed on a Windows XP PC with one Pentium 4 CPU running at 3.6 GHz and 1 GB of main memory. All run times in Table II are given in seconds and include time for parsing the XML file, conversion to propositional logic, building the BDDs, computing the error cases by existential abstraction, and writing the results to an XML file. The penultimate column of the table gives the number of errors found by the *HelpChecker* in the form x / y, where x is the number of missing packages and y the number of help package overlaps. The error cases were (partially) checked by the Siemens documentation department, and all reported errors were confirmed. The sizes of the XML files containing system descriptions and help contexts were up to 2.14 MB (with E-U-140-1 being the largest). Memory consumption of the *HelpChecker* was up to 80.4 MB of main memory for the E-F-405 test case. The final sizes of BDDs generated during all tests (i.e., the simplified BDDs describing sets of error situations) were never larger than 3,000 nodes. Intermediate BDDs, however, were much bigger. The BDDs for the ValidConf part alone consisted of 3,479 nodes and 365 propositional variables for test case E-T6-4 and 892 nodes and 235 variables for test case E-F-405.

The run time of a complete test run depend, of course, on the number of help contexts, the number of help packages, and the size of the configuration structure. Checking package overlaps, however, is independent of the number of help packages. The sizes of the BDDs mainly depend on the number of help contexts and the size of the configuration structure, and as these were already completely specified during our test runs, we do not expect any scalability problems on the final data sets. Moreover, we have implemented a partitioning technique that splits the large sets of

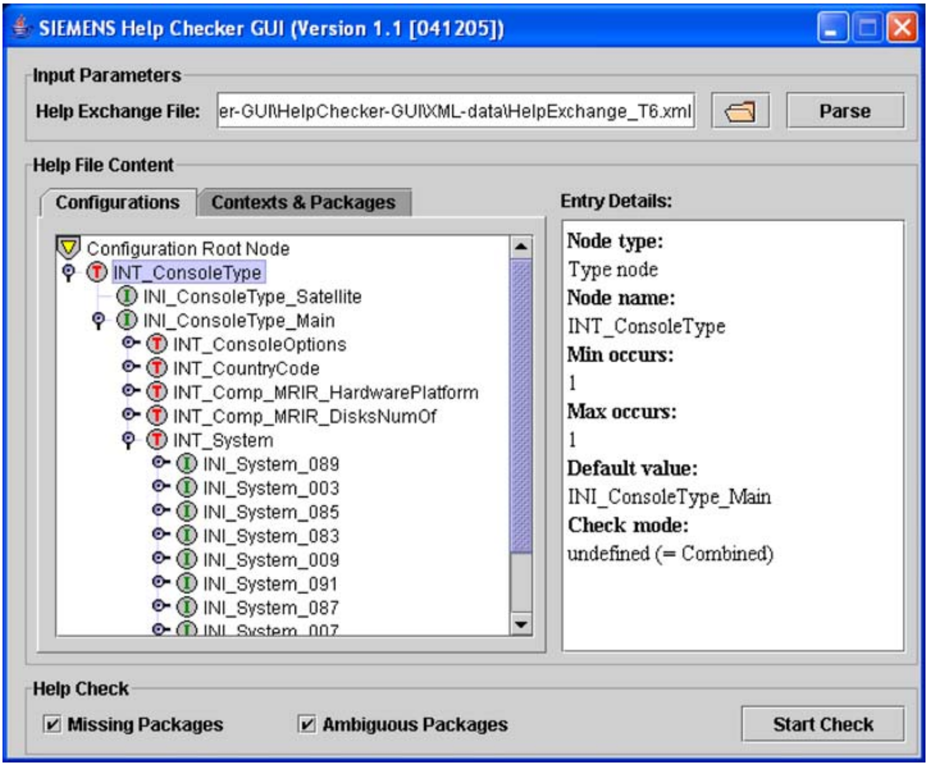
Table II Statistics on test runs performed by the *HelpChecker*.

Test case	#Systems	#Contexts	#Packages	#Errors	Run-time
E-T6-4	11 (partial)	964	12	905/9	1.94
E-U-805	4	3, 871	0	3871/0	10.95
E-F-505	4	1, 031	1	1,030/0	8.70
E-U-140-1	4	3, 871	3	3,869/1	11.13
E-F-405	4	3, 871	928	2,916/20	42.34
E-VF10A	3	77	48	34/1	0.94
E-H-306	4	1, 862	544	261/299	29.63

help packages and contexts into smaller portions that can be checked independently (the partitioning is based on workflow items).

To facilitate testing of the *HelpChecker*, we have developed a simple Java client (see Figure 5). This client allows loading of XML files containing system descriptions (configurations) as well as help package dependencies and contexts (so-called HelpExchange files). Both system structure and help packages can be displayed. The user can also initiate consistency checks and view the results (see Figure 6). This Java client is used only for testing purposes, however, and is not part of the final product.

In an early stage of the project we also made experiments with a SAT solver [30]. During these experiments, 35 SAT instances were generated for test case E-T6-4 (we used a fast approximative pretest for package overlaps that filters out trivial cases). Conversion to CNF, which is necessary for most SAT solvers, was done using the well-known technique due to Tseitin [36]. The generated SAT instances contained 1,425 different propositional variables and between 11,008 and 11,018 clauses (we employed a slightly different problem encoding then; see [30]). To check satisfiability, we used a sequential version of our parallel SAT solver PaSAT [27], which implements a variant of the DPLL algorithm [5, 6] with conflict clause generation and clause learning [17], as it is found in most state-of-the-art SAT solvers (e.g., zChaff [22] or MiniSAT [8]). Our solver indicated that ten of the instances



Help Exchange File: er-GUI\HelpChecker-GUI\XML-data\HelpExchange_T6.xml

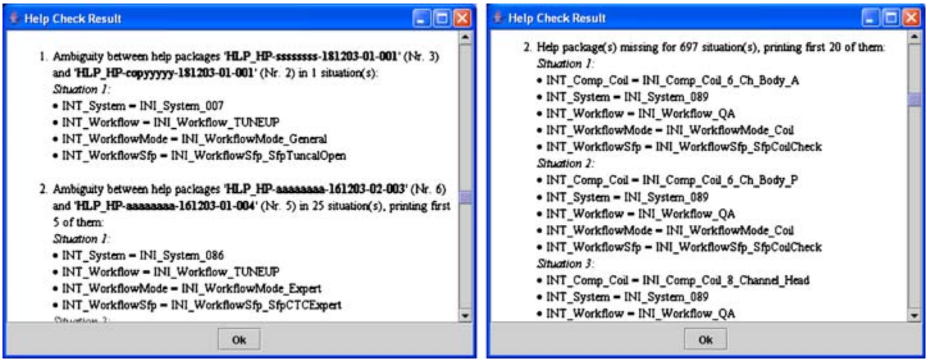
Configuration Root Node
- INT_ConsoleType
  - INI_ConsoleType_Satellite
  - INI_ConsoleType_Main
    - INT_ConsoleOptions
    - INT_CountryCode
    - INT_Comp_MRIR_HardwarePlatform
    - INT_Comp_MRIR_DisksNumOf
    - INT_System
      - INI_System_089
      - INI_System_003
      - INI_System_085
      - INI_System_083
      - INI_System_009
      - INI_System_091
      - INI_System_087
      - INI_System_007

Entry Details:

Node type:
Type node
Node name:
INT_ConsoleType
Min occurs:
1
Max occurs:
1
Default value:
INI_ConsoleType_Main
Check mode:
undefined (= Combined)

☑ Missing Packages ☑ Ambiguous Packages

Figure 5 Experimental Java client serving as a graphical user interface to the *HelpChecker*. Part of the product structure of the loaded test instance (E-T6-4) is displayed on the *left*.



```
1. Ambiguity between help packages 'HLP_HP-sssssss-181203-01-001' (Nr. 3)
   and 'HLP_HP-copyyyyy-181203-01-001' (Nr. 2) in 1 situation(s):
   Situation 1:
   • INT_System = INI_System_007
   • INT_Workflow = INI_Workflow_TUNEUP
   • INT_WorkflowMode = INI_WorkflowMode_General
   • INT_WorkflowSfp = INI_WorkflowSfp_SfpTuncalOpen

2. Ambiguity between help packages 'HLP_HP-aaaaaaaa-161203-02-003' (Nr. 6)
   and 'HLP_HP-aaaaaaaa-161203-01-004' (Nr. 5) in 25 situation(s), printing first
   5 of them:
   Situation 1:
   • INT_System = INI_System_086
   • INT_Workflow = INI_Workflow_TUNEUP
   • INT_WorkflowMode = INI_WorkflowMode_Expert
   • INT_WorkflowSfp = INI_WorkflowSfp_SfpCTCExpert
```

```
2. Help package(s) missing for 697 situation(s), printing first 20 of them:
   Situation 1:
   • INT_Comp_Coil = INI_Comp_Coil_6_Ch_Body_A
   • INT_System = INI_System_089
   • INT_Workflow = INI_Workflow_QA
   • INT_WorkflowMode = INI_WorkflowMode_Coil
   • INT_WorkflowSfp = INI_WorkflowSfp_SfpCoilCheck
   Situation 2:
   • INT_Comp_Coil = INI_Comp_Coil_6_Ch_Body_P
   • INT_System = INI_System_089
   • INT_Workflow = INI_Workflow_QA
   • INT_WorkflowMode = INI_WorkflowMode_Coil
   • INT_WorkflowSfp = INI_WorkflowSfp_SfpCoilCheck
   Situation 3:
   • INT_Comp_Coil = INI_Comp_Coil_8_Channel_Head
   • INT_System = INI_System_089
   • INT_Workflow = INI_Workflow_QA
```

Figure 6 Reports generated by *HelpChecker*. On the *left*, two instances of an overlap error are displayed, showing involved help packages and configurations for which the overlap occurs. On the *right*, situations for which help packages are missing are reported.

were satisfiable (correlating with error cases) and 25 were unsatisfiable. One of the satisfiable instances corresponded to a missing help package; the other nine were due to package overlaps. Unsatisfiability could always be determined by unit propagation alone; the maximal search time for a satisfiable instance amounted to 15.9 ms (on a 1.2 GHz Athlon running under Windows XP then). These surprisingly good results when applying SAT solvers to the configuration domain coincide with earlier observations made by the authors in the field of automotive product configuration [14]. We assume that the good results of SAT solvers on instances stemming from product configuration are due to the fact that inconsistencies (unsatisfiable instances) typically involve only a small fraction of the clauses of the whole instance. Therefore, small proofs exist for these formulas. Current SAT solvers, which are typically tuned for model-checking problems arising in hardware verification, seem to be especially well suited for such instances.

Although results with SAT techniques were very convincing, we do not employ them in the current version of the *HelpChecker*. Whereas performance is not an issue (in fact it is even better than with BDDs), SAT solvers possess the drawback that they do not allow for a concise presentation of all models (resp. error cases) of a formula. Of course, it is possible without too much effort to modify a SAT solver in such a way that it successively generates all models. This would still be insufficient, however, as subsequent operations on the set of all models, for example, to eliminate irrelevant variables, are still hard to realize.

5. Practical Aspects

HelpChecker is embedded in a larger, interactive authoring system for the writers of help packages at Siemens Medical Solutions. The authoring system was developed by Tanner AG, Germany, a company specializing in industrial documentation systems. A screenshot of this authoring system is shown in Figure 7. The authoring system uses an XML data base as its core component and allows calling the HelpChecker by pressing a 'Check Consistency' button.

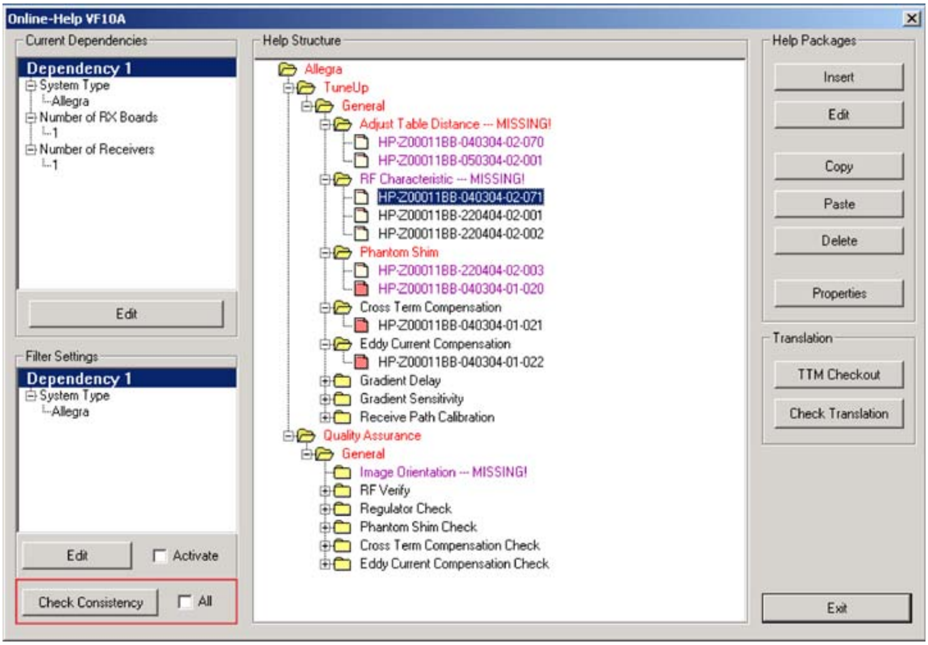


Figure 7 Siemens authoring tool: help packages can be added to the system, modified, deleted, and so forth. The main view shows a tree-structured representation of the help contexts. Missing packages are reported with a tagged error message ('– MISSING!') and a visual emphasis (*red*/*magenta* color). Package overlaps are also reported with a visual marker (*magenta*).

When this button is pressed, an XML file is generated that specifies the tests the *HelpChecker* has to perform (e.g., check both package overlaps and missing packages, but only for system *Allegra*). This – together with a link to the XML data containing the configuration structure and help data – is sent to the *HelpChecker*, which then builds BDDs and computes results. These are then sent back to the authoring system, where they are displayed in a tree-shaped structure showing the until-then-existing help packages. Errors are displayed with a color code highlighting erroneous packages. No further, more detailed data is given to the user to track the cause of the error, as this has not yet been considered necessary.

The authoring system has been in production use since October 2005 (still in a pilot-phase though), and both the conversion of old help pages from predecessor systems and the writing of new help pages is under way. First MR tomographs containing on-line manuals checked by the *HelpChecker* are supposed to ship in the second half of 2006.

The authoring system allows checking not only of the complete on-line help but also of fractions of it. For example, checks can be restricted to only one model line (in Figure 7 tests are restricted to the model line *Allegra*) or to only that part of the help document that the author is currently working on (by selecting nodes in the Help Structure; see Figure 7).

During development of the *HelpChecker* we observed that – as usual in leading-edge software development – the system specifications and thus their formalization are not fixed but change frequently over time. Thus, the encoding also had to be fine-tuned frequently. Specifically, the Check Mode attribute (see the definition of CardinalityOK(t) on page 11) was not present initially but was introduced later to handle a special, but frequently occurring situation with magnetic coils (multiple items of the same type are allowed in the configuration, but only those cases have to be checked where exactly one of them is present). Moreover, the Negate attribute for help contexts and dependencies (to negate clauses) and the interpretation of contexts (see the definition of HelpTypeCond(t) on page 14) was changed during the project. However, we conjecture that having precise mathematical underpinnings of the software (as given by the translation to propositional logic) facilitates the adoption of new requirements.

6. Related Work

A lot of different schemes for product configuration have been suggested in the literature [9, 16, 20, 21, 24], starting with McDermott's work on R1 [18] and Digital's XCON [2], both of which deal with the configuration of computer systems. Among the different formalisms that have been proposed are constraint satisfaction [16, 24], rule-based (expert) systems [18], SAT solving [14], feature logic [32], description logics [19, 20], and various graphical formalisms (see, e.g., [9]). Also, techniques from the area of logic programming (such as negation as failure or stable model semantics [10]) have been used for configuration [33].

Most of this work, however, is focused on configuration formalisms and on checking individual customer's orders for correctness. Less work has been done on consistency checking of configuration data as a whole [14] or on cross-checking product data with other related data such as handbooks. For work on consistency checking of Boolean configuration constraints in the automotive industry, see [14] and [28].

For SAT solving and constraint satisfaction, high-performance solvers are available today and are used in many industrial projects. SAT solving is the currently dominating technique used in hardware design verification (there are special tracks on SAT in design automation conferences), and commercial products for sales configuration based on constraint satisfaction (e.g., the ILOG Configurator) are available. For other logical formalisms such as description logic, solvers have also been implemented (like RACER [11] or FaCT [12]), but their practicability for large-scale industrial projects remains to be shown. The proven success of propositional reasoning techniques was one of our motivations for choosing propositional logic in the commercial project presented in this article.

Concerning consistency checking of XML documents, different approaches can be found in the literature. In order to check consistency of XML documents on the syntactic level, the W3C standards Document Type Definitions (DTDs) and XML Schema [37] have been developed and are in widespread use today. Alternatives to XML Schema are also available, for example, the Schematron rule language [15] or the XLinkIt system by Nentwich et al. [23]. Also related is work on consistency checking of CIM models by Sinz et al. [29] and on Java-based XML document evaluation by Bühler and Küchlin [4].

All these approaches differ considerably in the extent of expressible formulas and practically checkable conditions. The correctness of the semantic content, however, can be checked only to a certain extent by using these techniques. From a logical point of view, none of these techniques exceeds an evaluation of first-order formulas in a fixed structure, which is not sufficient for our application, which requires construction of different (propositional) models and thus real combinatorial search. In this respect, our method opens up new application areas for the discipline of XML checking.

7. 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 formulas and checked by using BDD techniques. Error conditions are output after symbolic simplification. By using a Boolean encoding we can also make use of advanced SAT-solvers as they are used, for example, in hardware verification to efficiently check formulas 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 useful for other complex products as well. More generally, we expect that a wide range of cross-checks between XML documents can be computed efficiently by using automated theorem-proving techniques based on SAT-solvers and BDDs.

Appendix

In this appendix we give definitions and explanations for the formulas and expressions skipped over in Section 3. We start with auxiliary definitions for sets of XML nodes.

Auxiliary set definitions:

$$\begin{aligned}
\mathrm{globalUnrefTypes} &= \mathrm{/Inventory/InvTypes/InvType} \setminus \bigcup_{\substack{t \in \mathrm{/Config/} \\ \mathrm{Structure/Type}}} \mathrm{refTypesT}(t) \\
\mathrm{unrefItems}(t) &= \mathrm{allItems}(t) \setminus t\mathrm{/Item} \\
\mathrm{allItems}(t) &= \bigcup_{t'\mathrm{@ID}=\mathrm{baseTypeID}(t)} \mathrm{/Inventory/InvTypes/}t'\mathrm{/InvItem} \\
\mathrm{baseTypeID}(t) &= \begin{cases} t'\mathrm{@Base} & \text{if } \exists t' \in \mathrm{/Inventory/InvTypes/InvTypeAlias} \\ & \qquad \text{with } t\mathrm{@IDREF} = t'\mathrm{@ID} \\ t\mathrm{@IDREF} & \text{otherwise} \end{cases}
\end{aligned}$$

The set globalUnrefTypes contains all type nodes that are defined but do not occur in any configuration structure. The set unrefItems(t) contains all items having the same type as node t but are not (direct) child nodes of t. These items are considered invalid for node t, as they are not explicitly given. They are computed as the set difference between the set of all items of this type (allItems(t)) and the explicitly specified child nodes (t/Item). The set allItems(t) contains all items of type t that are declared in the inventory; type aliases are reduced to their base types (specified under attribute Base).

$$\begin{aligned}\mathrm{unrefTypes}(i) &= (\mathrm{refTypesT}(i/..) \setminus \{i/..\}) \setminus \mathrm{refTypesI}(i)\\ \mathrm{refTypesT}(t) &= \{t\} \cup \bigcup_{i \in t/\mathrm{Item}} \mathrm{refTypesI}(i)\\ \mathrm{refTypesI}(i) &= \bigcup_{t \in i/\mathrm{RefType}} \mathrm{refTypesT}(t)\end{aligned}$$

The set unrefTypes(i) contains type nodes that are not allowed as child nodes of item node i. Such nodes do not occur below (i.e., as direct or indirect child node of) node i but are used on other branches of the configuration tree. It is sufficient to include only those nodes that are not already excluded in a parent item node i', as these are already contained in unrefTypes(i'). The auxiliary functions refTypesT(t) (resp. refTypesI(i)) compute the set of all type nodes that occur below type node t (including t) (resp. below item node i) (referenced nodes). With these functions, unrefTypes(i) is computed as the set of all referenced types of the parent node (refTypesT($i/..$)) that are not referenced by i itself (refTypes(i)).

Auxiliary formula definitions:

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

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

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

DecodeOp(d) is used within conditions of item nodes to enforce (‘eq’) (resp. exclude (‘ne’)) certain values on other item nodes. The resulting formula forces the corresponding propositional variables to be either constantly *true* or constantly *false*.

The selection operator S_a^b is used to formulate cardinality constraints. For two natural numbers a and b with $a \leq b$, formula $S_a^b(M)$ is true if and only if between a and b formulas out of the set M are true. Operator $S^b(M)$ is true if and only if at most b formulas out of set M are true. The selection operators may produce formulas having an exponential size in the numbers a and b. This can be avoided by using a more sophisticated encoding (see, e.g., [26]).

References

1. Baader, F., McGuinness, D., Nardi, P., Patel-Schneider, P. (eds.) The Description Logic Handbook. Cambridge University Press, Cambridge, UK (2003)
2. Barker, V.E., O’Connor, D.E.: Expert systems for configuration at Digital: XCON and beyond. Commun. ACM **32**(3), 298–318 (1989)
3. Bryant, R.E.: Graph-based algorithms for Boolean function manipulation. IEEE Trans. Comput. **35**(8), 677–691 (1986)
4. Bühler, D., Küchlin, W.: 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 2070, Budapest, Hungary. Springer, Berlin Heidelberg New York (2001)

5. Davis, M., Logemann, G., Loveland, D.: A machine program for theorem-proving. Commun. ACM **5**(7), 394–397 (1962)
6. Davis, M., Putnam, H.: A computing procedure for quantification theory. J. ACM **7**(3), 201–215 (1960)
7. Davis, S.M.: Future Perfect. Addison-Wesley, New York (1987)
8. Eén, N., Sörensson, N.: An extensible SAT-solver. In: Proc. 6th Intl. Conf. on Theory and Applications of Satisfiability Testing (SAT 2003), pp. 502–518. Springer, Berlin Heidelberg New York (2003)
9. Felfernig, A., Friedrich, G.E., Jannach, D.: UML as domain specific language for the construction of knowledge-based configuration systems. Int. J. Softw. Eng. Knowl. Eng. **10**(4), 449–469 (2000)
10. Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. In: Proc. 5th Intl. Conf. on Logic Programming, pp. 1070–1080. The MIT Press, Cambridge, MA (1988)
11. Haarslev, V., Möller, R.: Description of the RACER system and its applications. In: Working Notes of the 2001 International Description Logics Workshop (DL-2001), Stanford, CA (2001)
12. Horrocks, I.: FaCT. In: Proceedings of the 1998 International Workshop on Description Logics (DL'98), IRST, Povo-Trento, Italy (1998)
13. Kaiser, A.: A SAT-based propositional prover for consistency checking of automotive product data. Technical report, WSI-2001-16, University of Tübingen (2001)
14. Küchlin, W., Sinz, C.: Proving consistency assertions for automotive product data management. J. Autom. Reason. **24**(1–2), 145–163 (2000)
15. Lee, D., Chu, W.W.: Comparative analysis of six XML schema languages. SIGMOD Record (ACM Special Interest Group on Management of Data) **29**(3), 76–87 (2000)
16. Mailharro, D.: A classification and constraint-based framework for configuration. Artif. Intell. Eng. Des. Anal. Manuf. (AI EDAM) **12**(4), 383–397 (1998)
17. Marques-Silva, J.P., Sakallah, K.A.: Conflict analysis in search algorithms for propositional satisfiability. In: Proc. IEEE International Conference on Tools with Artificial Intelligence (1996)
18. McDermott, J.: R1: A rule-based configurer of computer systems. Artif. Intell. **19**(1), 39–88 (1982)
19. McGuiness, D.L., Wright, J.R.: Conceptual modelling for configuration: a description logic-based approach. AI EDAM **12**(4), 333–344 (1998)
20. McGuinness, D.L.: Configuration. In Baader et al. [1], pp. 397–414
21. Mittal, S., Frayman, F.: Towards a generic model of configuration tasks. In: Proc. 11th Intl. Joint Conf. on Artificial Intelligence, pp. 1395–1401, Detroit, MI (1989)
22. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an efficient SAT solver. In: Proc. 38th Design Automation Conference (DAC'01) (2001)
23. Nentwich, C., Capra, L., Emmerich, W., Finkelstein, A.: XLinkIt: A consistency checking and smart link generation service. ACM Transactions on Internet Technology **2**(2), 151–185 (2002)
24. Sabin, D., Freuder, E.C.: Configuration as composite constraint satisfaction. In: Luger, G.F. (ed.) Proc. Artificial Intelligence and Manufacturing Research Planning Workshop, pp. 153–161. AAAI, Albuquerque, NM (1996)
25. Sabin, D., Weigel, R.: Product configuration frameworks – a survey. IEEE Intell. Syst. **13**(4), 42–49 (1998)
26. Sinz, C.: Towards an optimal CNF encoding of Boolean cardinality constraints. In: Proc. 11th Intl. Conf. on Principles and Practice of Constraint Programming (CP 2005), pp. 827–831 (2005)
27. Sinz, C., Blochinger, W., Küchlin, W.: PaSAT - parallel SAT-checking with lemma exchange: Implementation and applications. In: Kautz, H., Selman, B. (eds.) LICS'2001 Workshop on Theory and Applications of Satisfiability Testing (SAT'2001), vol. 9 of Electronic Notes in Discrete Mathematics. Elsevier Science, Boston, MA (2001)
28. Sinz, C., Kaiser, A., Küchlin, W.: Formal methods for the validation of automotive product configuration data. Artif. Intell. Eng. Des. Anal. Manuf. **17**(1), 75–97 (2003). Special issue on configuration
29. Sinz, C., Khosravizadeh, A., Küchlin, W., Mihajlovski, V.: Verifying CIM models of Apache web server configurations. In: Proc. 3rd International Conference on Quality Software (QSIC 2003), pp. 290–297. IEEE Computer Society, Dallas, TX (2003)
30. Sinz, C., Küchlin, W.: Verifying the on-line help system of SIEMENS magnetic resonance tomographs. In: Proc. 6th Intl. Conf. on Formal Engineering Methods (ICFEM'2004), pp. 391–402. Springer, Seattle, WA (2004)
31. Sinz, C., Lumpp, T., Schneider, J., Küchlin, W.: Detection of dynamic execution errors in IBM System Automation's rule-based expert system. Inf. Softw. Technol. **44**(14), 857–873 (2002)

32. Smolka, G.: A feature logic with subsorts. LILOG Report 33, IWBS, IBM Deutschland, Stuttgart, Germany (1988)
33. Soininen, T., Niemelä, I.: Developing a declarative rule language for applications in product configuration. In: Proc. 1st Intl. Workshop on Practical Aspects of Declarative Languages (PADL'99), pp. 305–319, San Antonio, TX (1999)
34. Soininen, T., Tiihonen, J., Männistö, T., Sulonen, R.: Towards a general ontology of configuration. AI EDAM **12**(4), 357–372 (1998)
35. Somenzi, F.: Cudd: Cu decision diagram package release (1998)
36. Tseitin, G.S.: On the complexity of derivation in propositional calculus. In: Slisenko, A.O. (ed) Studies in Constructive Mathematics and Mathematical Logic, pp. 115–125 (1970)
37. XML Schema Parts 0–2: Primer, Structures, Datatypes. W3C Recommendation (2001)
38. XML Path Language 2.0. W3C Working Draft, April 2002