Configuration Dynamics Verification Using UPPAAL

David Fabian
Dept. of Mathematics
Faculty of Nuclear Sciences and Physical Engineering
Czech Technical University in Prague
(fabiadav@fjfi.cvut.cz).

Radek Mařík
Dept. of Cybernetics
Faculty of Electrical Engineering
Czech Technical University in Prague.

Abstract

Modern software applications can have very complicated internal dynamics. Most of the software tools are written in an imperative programming language which can quickly become impractical for describing complex dynamics. Also, it is very hard to verify that the code actually covers fully all aspects of the tool's dynamics. Propagation rules are suitable as a means for specification and verification of such dynamic systems. We have selected a software tool from the domain of configuration for our study. Configuration wizards and tools are examples of software applications where even a small change made by the user can lead to a very complex outcome. In this paper, a configuration hierarchical model and a syntax of propagation rules are introduced. These constructs can be used to describe declaratively the dynamics that is typical for software configuration tools. The hierarchical model is then used for describing the internal dynamics of the configuration tool Freeconf. This specific model instance is then implemented in UPPAAL and verified by the UPPAAL model-checker.

1 Introduction

Software applications become more and more complicated, nowadays. The complexity of the internal dynamics of a modern software application can be hard to maintain. Software configuration is one of the areas where the dynamics can become very complicated.

Software configuration can be divided into two distinct groups. In literature, configuration is usually understood as finding such a combination of software/hardware modules that the resulting product satisfies some prescribed requirements [Vlaeminck *et al.*, 2009]. On the other hand, from the point-of-view of the end-user, configuration process means changing some options (configuration keys) of a finished product, so that it will adapt to the user's needs (changing the background of the desktop, choosing the size of the subtitles in the media-player, setting up permissions for the web server) [Liaskos *et al.*, 2005; Fabian, 2012]. Nowadays, there exist software tools designed to aid the user with these application adjustments. They often offer a GUI (Graphical User Interface) in a form of one or more configuration windows and are usually hard-wired to the application itself. There also exist general-purpose configuration tools such as KConfigXT [TechBase, 2012], and Freeconf [Fabian, 2012].

Since there can be many possible configuration keys in a configuration, it is natural to organize the keys into hierarchical categories. Each key also has an inner state formed by some properties that describe it. When the user interacts with a configuration tool, some configuration keys change their state in reaction to the user's input. Every change can be propagated further across the hierarchy and induce more changes in other keys depending on the semantics of the properties. In a configuration tool with many internal key properties, the amount of property interactions can lead to a complex dynamical behavior which is difficult to implement in an imperative style programming language. However, it is straightforward to describe the dynamics in a declarative form as propagation rules. The elegance of this approach consists in condensing the description of the dynamics (which can be very complex) to a single list of rules. That list can be then (semi)automatically verified for its soundness and completeness.

This topic is related to problems from the domain of production rules and knowledge bases [Arman, 2013; Preece and Shinghal, 1994; Preece and Shinghal, 1992]. Some initial work has been done to address the problem of automatic detection of rules redundancy and inconsistency in [Lukichev, 2011]. In the paper, the author uses description logic [Nardi and Brachman, 2003] to describe, in an abstract and general way, some typical patterns which can lead to an inconsistent or non-minimal set of production rules. The long-term goal of our work, however, is to develop a usable software application which would (semi)automatically verify the minimality and consistency of hierarchical models that are used in software configuration tools.

In this paper, an attempt to model and verify configuration software dynamics is introduced. A general model of a configuration hierarchy is described together with declarative rules that are used to model the dynamics on top of the hierarchy. Further, it is shown on a specific instance of the model (which is used in Freeconf) how such a set of rules can look like. Finally, the soundness of this instance is studied and verified by using the model-checking utility UPPAAL.

The rest of this paper is divided as follows. In Section 2, the hierarchical model and propagation rules are presented. Section 3 describes the properties used in Freeconf and the specific set of rules that describe the dynamics of properties propagation in it. Section 4 introduces UPPAAL, a model-checking software tool. In Section 5, it is presented how the Freeconf propagation rules can be modeled and their soundness verified in UPPAAL. Finally, Section 6 summarizes the results and Section 7 concludes.

2 Hierarchical Model and Rules

In this section, a configuration hierarchical model and a syntax of propagation rules will be introduced.

2.1 Hierarchical Model

The hierarchical configuration model can be thought as a rooted acyclic graph where each node has an internal state. The state of a node can be changed directly by the environment (i.e., the user) or indirectly as a result of propagation of some direct change. The internal state will be limited to only Boolean and bounded integer properties in this paper.

Definition The *internal state* of a node in the hierarchical configuration model is a tuple of sets (B, I, D), where $B_i \in B$ represents a Boolean property and $I_j \in I$ represents an integer property from a single bounded integer domain D. At least one of the sets B and I must be non-empty.

Definition A *node* of a configuration hierarchical model is a tuple (i, p, C, X), where i is a unique positive integer index, p is the node's parent index, C is a (possibly empty) set of indices of the node's successors in the graph, and X is the internal state of the node. There exists a special parent index $\emptyset$ for the top-level node of the hierarchy denoting the absence of a parent.

Because some properties in different nodes can have the same name, a term X_j^i will be used henceforth to denote a property X_j of the node with index i.

Definition A *configuration hierarchical model* M is a non-empty set of nodes. A top-level node, i.e. the one with the parent index $\emptyset$ must always be present.

2.2 Propagation Rules

In general, the user can initiate a propagation by changing any property of an arbitrary node in the hierarchy (even more properties at once). From that, based on the semantics of the propagation, the change can propagate within that node, further up to the node's parent, down to its successors, or does not have to propagate at all.

The dynamics can be formally described by propagation rules.

Definition A *propagation rule* has a form $\mathcal{A} \to \mathcal{B}$, where $\mathcal{A}$ is the head (condition) of the rule and $\mathcal{B}$ is the body (action). The head is always bound to a specific node and consists of a non-empty conjunction of the node's Boolean properties or their negations and terms $I_j \circledast v_j$, where I_J is the node's integer property and v_j is an integer constant. $\circledast$ is a substitute for comparison operators $<, >, \leq, \geq, ==, \neq$. The body is formed by a non-empty conjunction of assignments to, in general, Boolean and integer properties of the node itself, to properties of the node's parent and to properties of its children. If the head of a propagation rule is satisfied (i.e. all Boolean properties are true and all comparison operations hold) the rule fires and the body is executed leading to a change of values of other properties.

The terms "head" and "body" are used in this particular order to match Constraint Handling Rules (CHR) terminology since there are plans to use CHR to develop propagation rules solver (see Section 7).

A syntactical shortcut will be used to express $I_j = I_j + 1$ and $I_j = I_j - 1$, i.e., incrementing and decrementing an integer property by one, as I_j++ and I_j--, respectively. According to the definition, a general propagation rule that operates only within a single node with index a will have the following form

$$\left(\bigwedge_i B_i^a \wedge \bigwedge_m (I_m^a \circledast v_m)\right) \to \bigwedge_j (B_j^a = b_j) \wedge \bigwedge_k (I_k^a = c_k) \ .$$

A propagation rule that represents a communication between a node and its parent will be

$$\left(\bigwedge_i B_i^a \wedge \bigwedge_m (I_m^a \circledast v_m)\right) \to \bigwedge_j (B_j^p = b_j) \wedge \bigwedge_k (I_k^p = c_k) \ .$$

Finally, a propagation rule that describes a communication between a node and some of its children can be described as

$$\left(\bigwedge_i B_i^a \wedge \bigwedge_m (I_m^a \circledast v_m)\right) \to \bigwedge_{j,l} \left(B_j^{C_l} = b_j\right) \wedge \bigwedge_{k,l} \left(I_k^{C_l} = c_k\right) \ .$$

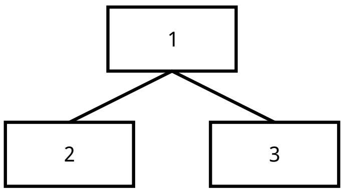

Figure 1: An example of configuration hierarchical model with three nodes.

To better illustrate how different types of propagation can look like, let us consider a configuration hierarchical model M with three nodes as given in Figure 1. Each node will have an identical structure of the internal state $X = (bool_1, bool_2, int_1, \{0, 1, 2\})$. The model M will be a set of three tuples

$$\begin{aligned} M = \{ \ & \left(1, \emptyset, \{2,3\}, \left(bool_1^1, bool_2^1, int_1^1, \{0,1,2\}\right)\right), \\ & \left(2, 1, \emptyset, \left(bool_1^2, bool_2^2, int_1^2, \{0,1,2\}\right)\right), \\ & \left(3, 1, \emptyset, \left(bool_1^3, bool_2^3, int_1^3, \{0,1,2\}\right)\right) \ \} \ . \end{aligned}$$

Let us further assume that the semantics of the properties declares that:

- whenever $bool_1$ is $false$ for node two, $bool_2$ must also be $false$ for that particular node
- whenever $bool_2$ is $true$ and int_1 is greater than one in node three, the value of the parent's int_1 must be two
- whenever int_1 is zero for node one, $bool_2$ for node two must be $true$ and int_1 for node three must be one

The respective propagation rules are given below.

$$\begin{aligned} &\neg bool_1^2 \to bool_2^2 = false \\ &bool_2^3 \wedge int_1^3 > 1 \to int_1^1 = 2 \\ &int_1^1 == 0 \to bool_2^2 = true \wedge int_1^3 = 1 \end{aligned}$$

Of course, the body of a propagation rule can affect not only the node itself, the parent, and the successors separately, but also any combination of the respective internal states. In general case, poorly designed rules can form a loop and thus lead to a non-terminating computation.

3 Freeconf properties

In this section, Freeconf is briefly introduced and its internal dynamics modeled as a configuration hierarchical model is presented.

3.1 Freeconf Tool

Freeconf is a general-purpose cross-platform configuration utility developed at Czech Technical University [Fabian, 2011; Fabian, 2012]. The tool is supposed to create an intermediate layer between the user and an application without any configuration GUI. An example of such applications can be various application servers, web servers, some movie players, and basically any program that stores its configuration in configuration text files. When requested by the user, Freeconf automatically generates a configuration dialog with application specific configuration options (configuration keys), the user then can change the configuration according to her liking, and during saving Freeconf transforms the output into the respective native configuration files from where the application can read the changes. The details about this process can be found in [Fabian, 2012].

3.2 Key and Section Properties

property	meaning
static mandatory (*sman*)	If it is false, the key is never shown to the user unless *show all* property is set. If it is true the visibility is controlled by the dynamic equivalent of this property.
static active (*sact*)	If it is false, the key is as being commented out from the list of possible keys. No further action is applicable to the key and the key is not visible to the user.
dynamic mandatory (*dman*)	This property can only be set as a result of propagation of the user's action. If it is true, the key is mandatory and must be shown. This property has no meaning when the static mandatory property is set to false.
dynamic active (*dact*)	This property can only be set as a result of propagation of the user's action. If it is false, the key does not have sense in the current settings. This property has no meaning when the static active property is set to false.
value set (*valset*)	This property is true iff the key has a value assigned. In some configuration, there can be an initial state where the key does not have any value.
default value set (*defvalset*)	This property is true iff the key has a default value assigned. There can be some keys in the configuration which do not have a default values assigned.
undefined (*undef*)	This is an aggregation property which is true iff the value and default values are both unset.
inconsistent	The key is inconsistent with the configuration iff it is undefined and is dynamically mandatory and dynamically active.
show all (*showall*)	This is a special property which overrides whether the keys are shown to the user or not. Every key will always have the same value of this property because the user will change it at once for all the keys (broadcast change). If this property is true all dynamically active keys are shown to the user, even those that are not mandatory.

Table 1: Freeconf key properties.

Freeconf can handle hundreds or even thousands of configuration options. To avoid overfilled and confusing configuration dialogs, it is necessary to divide the options into specific categories. It is done by assigning a set of Boolean properties to every option such that truthfulness of a specific set of properties means the option belongs to the respective category. At the moment, Freeconf uses, apart from the basic set of properties that are static and do not participate in property propagation, a set of properties for every configuration key. *Static* properties are meant to be fixed throughout the run of Freeconf and cannot be changed by the user. *Dynamic* properties, on the other hand, can have their values changed in reaction to the user's action quite often. *Mandatory* property reflects the fact that the key is vital for the configuration and must be shown to the user at any case. *Activity* of the key determines its current state of presence or absence in the configuration. The key properties are given in Table 1.

Semantically related configuration keys are often grouped together to so called *configuration sections*. These are basically containers which can hold both other configuration sections or configuration keys. The sections have themselves some properties that help them to keep track of the state of their direct successors and react, for example, to the situation where all successors of a given section should be hidden. In that case, the section should hide itself too. The current set of section properties is given in Table 2.

Freeconf has a semantics which describes the evolution of properties values in reaction to the users actions. It has been formulated as a set of propagation rules in [Fabian *et al.*, 2012].

In the context of Section 2, it is easy to encode Freeconf properties propagation into a configuration hierarchical model. There will be two types of nodes in the model, one for configuration keys and one for configuration sections. Following Definition 2.1, the internal state of a configuration key will be a tuple with nine Boolean properties and no integer properties:

$$\begin{aligned} X = (\ &\{defvalset, valset, sman,\\ &dman, sact, dact,\\ &undef, inconsistent, showall\},\\ &\emptyset,\\ &\emptyset\ )\ . \end{aligned}$$

The internal state of a configuration section will be a tuple of two Boolean properties and four integer properties:

$$\begin{aligned} Y = (\ &\{empty, inconsistent, showall\},\\ &\{mancounter, actcounter,\\ &\ inccounter, sectionshowncounter\},\\ &[0 \ldots N]\ )\ , \end{aligned}$$

where N is the number of successors of the section. The configuration hierarchy will be formed with configuration keys as leaf nodes and configuration sections as non-leaf nodes.

3.3 Propagation Rules

The list of all propagation rules that describe the dynamics in Freeconf is given in [Fabian *et al.*, 2012] where the rules are represented using a rule-based constraint programming syntax [Brand, 2004]. A transformation to the propagation rules syntax is straightforward. For example, whenever *dynamic mandatory* property of a node changes its value, the parent must be informed and its *mandatory counter* must be adjusted accordingly. Propagation rules describing this change are shown below.

$$\begin{aligned} dman_i &\rightarrow mancounter_i^p\text{++}\\ \neg dman_i &\rightarrow mancounter_i^p\text{--} \end{aligned}$$

In Freeconf, rules rewrite only nodes that are higher up in the hierarchy, i.e., node to section and section to section communication. This flow of information suffices for the needs of Freeconf and prevents non-termination.

property	meaning
mandatory counter (*mancounter*)	This counter reflects the number of key successors that are dynamic mandatory.
active counter (*actcounter*)	This counter reflects the number of key successors that are dynamic active.
inconsistent counter (*inccounter*)	This counter reflects the number of successors (even sections) that are inconsistent.
section shown counter (*sectionshowncounter*)	This counter holds the number of successor sections that are not empty.
empty	This property is true iff there are no mandatory and active key successors and no non-empty successor sections.
inconsistent	This property is true if there is at least one inconsistent successor. In another words, the inconsistent counter is not zero.
show all (*showall*)	The same property as in the case of the keys. The user will change the value for every section and every key at once overriding the emptiness of the section.

Table 2: Freeconf section properties.

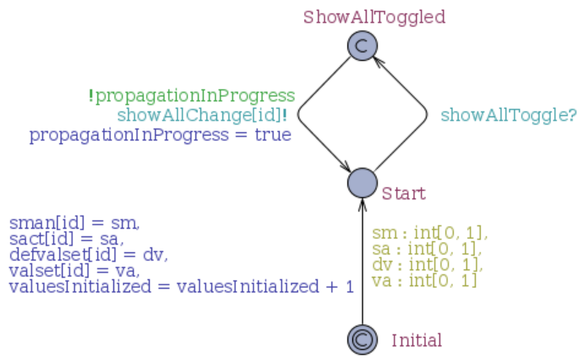

Figure 2: A state machine automaton template designed in UPPAAL.

modality	meaning
E <> φ	There exists a run in which φ eventually holds.
A <> φ	In every run, φ eventually holds.
E[ ]φ	There exists a run in which φ always holds.
A[ ]φ	In every run, φ always holds.

Table 3: Modalities used in UPPAAL.

4 UPPAAL

UPPAAL is a model-checking verification software of real-time dynamic systems developed by Upsalla University and Aalborg University [Behrmann *et al.*, 2004; David *et al.*, 2009]. A modeled system is represented as a network of timed automata and one can verify the soundness of the model by querying the built-in model checker.

4.1 Modeling in UPPAAL

UPPAAL offers a GUI written in Java to design each automaton of the network graphically. One can also program parts of the automata using a C-like syntax language. A very useful feature is templating which simplifies designing of very similar automata by using constant template parameters. UPPAAL will automatically unfold a template by creating an automaton according to the template for every possible value of the template parameter.

UPPAAL further supports adding guards to transitions, time invariants to nodes, choosing non-deterministically a value of a variable during a transition, and a synchronization of two or more concurrent transitions via signals. An example of a modeled automaton template can be seen in Fig.2.

UPPAAL offers declaration of constants, typed variables and single-dimensional and multi-dimensional arrays. A variable can be either Boolean, or a bounded integer (in fact, Boolean is a special case of `int[0,1]`). The scope of a variable is either local to the automaton, or is global, so all parts of the model can access the variable. In Fig. 2, the variable `sm` is Boolean and its value is chosen randomly upon transition from the `Initial` state to the `Start` state. During that same transition, the value of `sm` is assigned to the cell of array `sman` with index `id`. The variable `id` is in fact a template parameter and its value is different yet constant in every instance of this template.

4.2 Query Language

The main purpose of a model-checker is to verify the model w.r.t. a requirement specification [Behrmann *et al.*, 2004]. The requirements must be formulated in a well-defined language. UPPAAL uses a simplified version of timed computation tree logic (TCTL) [Alur *et al.*, 1993].

$$\mathrm{E} <> forall(i : id\_s) manCounter[i] < 0 \qquad (1)$$

An example of a query is given in Equation 1. The query usually starts with a path quantifier and a modality determining the validity of a formula along a specific run of the system. All possible modalities are presented in the following table.

Apart from the modalities stated above, UPPAAL also supports the *until* modality φ --> ψ which can be read as "In every run, if φ holds then ψ eventually holds". The same formula can be obtained by using only the modalities from Table 3 $\mathrm{A}[\,](\varphi => (A <> \psi))$. UPPAAL however does not support multiple modalities in a single query, so --> is the only possibility.

The model-checker in UPPAAL is written in C and it is possible to choose different sub-algorithms it uses during the verification via the program's menu. If the verification of a query fails, UPPAAL can be set to produce a counter-example in the form of a system trace. The trace demonstrates how to get from the initial state to a state where the query does not hold.

5 Modeling of Rules Using UPPAAL

Since the hierarchical configuration model can become quite large and there can exist a lot of propagation rules, it would be profitable to have a means of verification that the rules express the intended semantics correctly, are consistent, and not redundant. The hierarchical model can be thought of as a Kripke structure where each instance of the model forms a possible world and propagation rules describe possible transitions. Since UPPAAL in its core uses Kripke structures, it is natural to model the configuration hierarchical model in UPPAAL.

As the first attempt, the Freeconf model was verified in UPPAAL. The entire Freeconf specific configuration hierarchical model was encoded into UPPAAL and then the model-checker was used to perform the verification. In the following sub-sections, each of the steps will be briefly described. The entire model and all

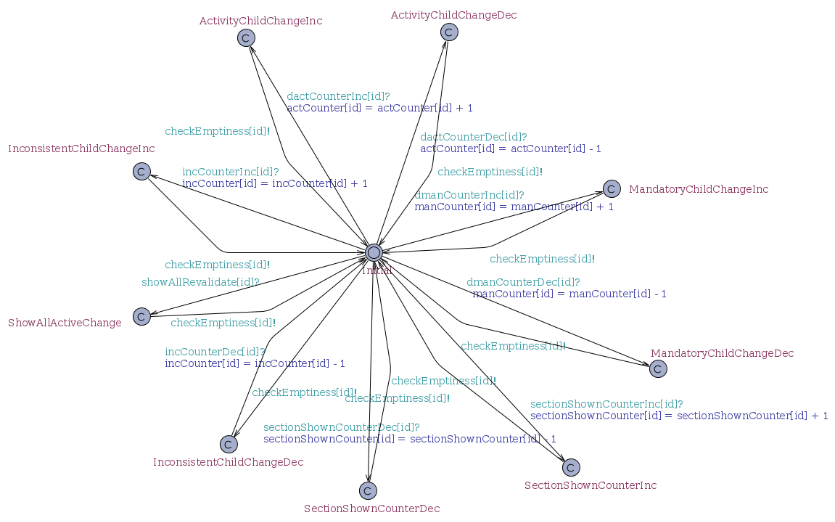

Figure 3: Freeconf section node modeled in UPPAAL.

queries can be found at `http://kmlinux.fjfi.cvut.cz/~fabiadav/phd/uppaal/freeconf_model.zip`.

5.1 Modeling Phase

There are multiple problems to be solved while modeling the hierarchy. Firstly, Freeconf key and section properties forming the internal states of Freeconf nodes have to be inserted. As shown in Listing 1, they are declared as global Boolean and integer arrays in a straightforward way, where each node of the hierarchical model is assigned a non-negative index and accessing the internal state of the node is simply done by reading elements from all the arrays with the same index. Even though in Freeconf, integer counters generally do not have the same domains, it is acceptable to approximate the general situation by setting the upper limit of each counter to the number of nodes in the hierarchy (to the number of sections in the hierarchy for *sectionShownCounter*). Property *show all* is implemented as a single shared Boolean variable for the user template to be able to change the property at once.

```
const int N = 3; // number of keys
const int M = 2; // number of sections

bool defvalset[N], valset[N], sman[N],
     dman[N], sact[N], dact[N], undef[N],
     inconsistent[N];

int[0, N] incCounter[M], manCounter[M],
          actCounter[M];
int[0, M] sectionShownCounter[M];
bool empty[M];
bool showAll;
```

Listing 1: Declaration of Freeconf properties

The hierarchical nodes are encoded as state machine automaton templates parametrized by node indices. For each key, there exists an automaton `Node` which was presented earlier in Figure 2. The automaton has two main tasks to perform — setting the initial state of all its properties to random values (so that the model-checker can later test all possible combinations of properties values) and updating the visibility in the case when the user has modified the show all property. For each section, there is an automaton `Section` given in Figure 3.

The section template merely listens to signals from its successors and updates its respective counters and the emptiness status.

The most complicated part deals with the design of the hierarchical model. Since channel synchronization and global variables are the only possibilities to exchange information between the automata in UPPAAL, properties propagation is implemented by using these two. Two separate tree connections have to be considered because they behave differently. For section-key connections (that is to model the propagation between the keys and their parent sections), a special automaton template `NodeSectionDispatcher` is created. For section-section connections (i.e., for modeling the propagation between the sections and their parent sections), another automaton template `SectionDispatcher` is created. `NodeSectionDispatcher` template is parametrized by an index which spawns from zero to the number of section-key pairs. `SectionDispatcher` template is similarly parametrized by an index going from zero to the number of section-section pairs (including the top-level pair whose one end is connected the top- level section and the other to `TopLevelTerminator` automaton). Two global two-dimensional arrays are declared which, in fact, model the hierarchical tree as a parent-child relation for node indices.

```
const int NODECON = 3;
const int SECTIONCON = 2;
const int[0, N] disIdx[NODECON][2] =
                {{0, 0}, {1, 0}, {2, 1}};
const int[0, M] disSecIdx[SECTIONCON][2] =
                {{0, 1}, {1, 2}};
```

Listing 2: Hierarchical tree modeled as two two-dimensional arrays

Array `disIdx` has key indices in its first dimension and their parent section indices in its second dimension. Thus, in the example above key zero has section zero as its parent, key one has section zero, etc. Array `disSecIdx` is the equivalent of `disIdx` array but this time it describes section relationship. Note that in the initialization of `disSecIdx`, constant two is used even though in this case, only two sections are present in the example model, and hence

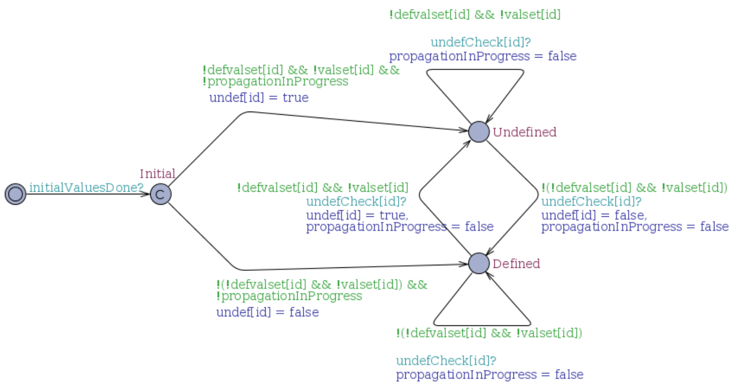
!defvalset[id] && !valset[id]
undefCheck[id]?
propagationInProgress = false
!defvalset[id] && !valset[id] &&
!propagationInProgress
undef[id] = true
Undefined
Initial
initialValuesDone?
C
!defvalset[id] && !valset[id]
undefCheck[id]?
undef[id] = true,
propagationInProgress = false
!(!defvalset[id] && !valset[id])
undefCheck[id]?
undef[id] = false,
propagationInProgress = false
Defined
!(!defvalset[id] && !valset[id]) &&
!propagationInProgress
undef[id] = false
!(!defvalset[id] && !valset[id])
undefCheck[id]?
propagationInProgress = false

Figure 4: Propagation rules modifying the property *undefined* modeled as a state machine.

the index should be invalid. The first index that is not a valid section index is, however, used to denote the top-level connection and is thus valid in this context.

Constants NODECON and SECTIONCON represent the number of section-key connections and section-section connections, respectively. These arrays are used by the dispatcher automata to automatically dispatch propagation signals across the tree. This settings is flexible in the way that changing the shape of the hierarchy is simply a matter of adjusting four constants and two arrays.

While the tree structure can be abstracted and generalized to allow easy modifications, the actual rules have to be hard-wired into the automata templates because the semantics behind those rules must be expressed in a visual form. For example, the automaton presented in Figure 4 models the behavior of propagation rules that modify *undefined* property.

Enforcing causality of the propagation turns out to be another complication. For example, the initialization phase, where the keys are assigned some initial values and the first step of the propagation fires, must come before the user is activated. Enforcing causality does not require to use clocks that are the integral part of UPPAAL, only auxiliary variables and synchronization channels suffice. On the other hand, since automata execution is by default parallel in UPPAAL, the auxiliary code that controls causality renders the model more complicated and less clear.

Finally, the user is modeled as a single state machine automaton. The automaton non-deterministically chooses a key in the hierarchy and one of its properties that is changeable at the current state. In order to avoid race-conditions, every user's action must lock the hierarchy until the propagation has been finished. A global variable propagationInProgress is used for this purpose.

5.2 Verification Phase

Of course, since Freeconf model can become arbitrarily large by adding more sections and keys to the model, only a small amount of actual instances of the general Freeconf model is verified by the exemplar UPPAAL model. The instances are given in Figure 5. In reality, Freeconf can easily have four or five levels of sections in the hierarchy since there exist auxiliary non-visible sections and window tabs that act as sections in the model. On the other hand, Freeconf model is somehow homogeneous which means that if node to section and section to section properties propagation is valid than Freeconf model with arbitrarily large tree should be also valid.

Model-checking queries are divided into several groups. The first group serves a purpose of testing the UPPAAL model itself, because the encoding of the problem is not very straightforward and often

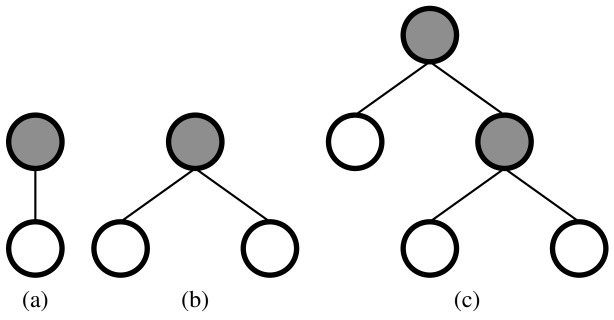

Figure 5: Freeconf hierarchical models validated in UPPAAL. Sections are depicted as gray, keys are white.

leads to cycles and non-termination. The basic liveness checking query $E <> deadlock$ appears to be very useful.

When the model is ready, it is necessary to further test whether updates to the integer properties do not set any value out of the domain. Queries similar to the one in the following listing are used for this test for every counter.

$$E <> forall(i : id_s)\ actCounter[i] > N$$
$$E <> forall(i : id_s)\ actCounter[i] < 0$$

Of course, just in this situation, the domains of the respective counters are changed to $[-1, N+1]$ or $[-1, M+1]$. Other basic type checking was also done.

In [Fabian *et al.*, 2012], one of the open problems was to determine if the following two sets of propagation rules that are given in Equation 2 and 3 are complementary and if one can replace the rule heads in the second set by just the negation of the heads from the first set. These two rules affect the state of the section *emptiness* property. They have got non-symmetric heads as a result of iterative development of Freeconf.

$$\begin{aligned}
&empty_i^i \wedge \Big(sectionshowncounter_i^i > 0\Big) \rightarrow \varphi \\
&showall_i^i \wedge \Big(activeshown_i^i > 0\Big) \rightarrow \varphi \\
&\neg showall_i^i \wedge \Big(activeshown_i^i > 0\Big) \wedge \\
&\wedge \Big(mandatoryshowncounter_i^i > 0\Big) \rightarrow \varphi \\
&\varphi := \Big(empty_i^i = false \wedge (sectionshowncounter_i^p - -)\Big)
\end{aligned} \tag{2}$$

$$\begin{aligned}
&\neg empty_i^i \wedge \Big(mandatoryshowncounter_i^i == 0\Big) \wedge \\
&\wedge \Big(sectionshowncounter_i^i == 0\Big) \rightarrow \varphi' \\
&\Big(activeshowncounter_i^i == 0\Big) \wedge \\
&\wedge \Big(sectionshowncounter_i^i == 0\Big) \rightarrow \varphi' \\
&\varphi' := \Big(empty_i^i = true \wedge (sectionshowncounter_i^p + +)\Big)
\end{aligned} \tag{3}$$

By using a query (and its derivation with negated outer conjuncts) which are given in Equation 4, it can be shown that the rule sets are not complementary and there exist situations where both rules are applicable. To solve this error in the model, one has to modify the rule head in the second set by adding $\neg showall_i$ as is shown in Equation 5. After this minor tweak, the rules behave correctly and the *emptiness* property is set correctly in all situations with the existing code in Freeconf.

$$\begin{aligned}
&E <> forall(i : id_s)(sectionShownCounter[i]|| \\
&||(showAll\&\&actCounter[i])|| \\
&||(!showAll\&\&actCounter[i]\&\&manCounter[i]))\&\& \\
&\&\&((!manCounter[i]\&\&!sectionShownCounter[i] \\
&\&\&!showAll)||(!sectionShownCounter[i]\&\& \\
&\&\&!actCounter[i]))
\end{aligned} \tag{4}$$

$$\begin{aligned}
&\neg empty_i^i \wedge \Big(mandatoryshowncounter_i^i == 0\Big) \wedge \\
&\wedge \Big(sectionshowncounter_i^i == 0\Big) \wedge \underline{\neg showall_i^i} \rightarrow \varphi' \\
&\Big(activeshowncounter_i^i == 0\Big) \wedge \\
&\wedge \Big(sectionshowncounter_i^i == 0\Big) \rightarrow \varphi'
\end{aligned} \tag{5}$$

One of the hardest part is to construct a query that would allow us to ask for the correctness of propagation of the node and section *inconsistency* property. The final query which is shown below in Equation 6 must use UPPAAL node names and the *until* operator to be able to express the property update dynamics. Since UPPAAL does not allow universal quantification with the *until* operator, it is not possible to create a general query for every node.

$$\begin{aligned}
&(empty[0]\&\&empty[1]\&\&dman[0]\&\&dact[0]\&\& \\
&Inconsistent(0).SetInconsistentTrue) \\
&\text{-->}(!empty[0]\&\&!empty[1]\&\& \\
&TopLevelTerminator.TerminationDone)
\end{aligned} \tag{6}$$

5.3 UPPAAL Model-checker Performance

Because UPPAAL model is parametrized so that it can be very easily modified to represent a different instance of Freeconf model, it is interesting to measure UPPAAL's model-checker performance with respect to the size of Freeconf model. A fixed query 7 which tests the correctness of the node *inconsistency* property modifications is used for the needs of this measurement. The query uses "for all" path quantifier A and "always" temporal modality [ ], so UPPAAL would have to traverse a great amount of states to draw a conclusion.

$$\begin{aligned}
&A[\,]forall(i : id_k)(undef[i]\&\&dman[i]\&\&dact[i]\&\& \\
&\&\&!propagationInProgress\&\&!userAction\&\& \\
&\&\&initFinished)\ imply\ (inconsistent[i])
\end{aligned} \tag{7}$$

In Table 4, the time and consumed memory it took to finish the validation process of the query for the three Freeconf models which were introduced in Figure 5 is given together with the upper estimate on the number of states UPPAAL has to traverse. The test was performed on a desktop PC with Intel Core 2 Quad Q9550 CPU at 2.83 GHz, 4 GiB RAM, running 64 bit Linux 3.1.10 and a development snapshot of UPPAAL 4.1.14.

Model	Time (s)	Memory (KiB)	# of states
a	0.07	6889	16384
b	1.67	24572	21233664
c	189.1	2147932	17592186044416

Table 4: Performance statistics of UPPAAL model-checker with respect to the size of Freeconf model.

6 Results

The entire Freeconf model has been encoded in UPPAAL. Some of the parts of the hierarchical configuration model, like the tree structure, are easy to implement in UPPAAL, others, like propagation rules, are more problematic and sometimes require a substantial amount of auxiliary coding/modeling to be able to express certain features of the hierarchical model, e.g. causality. On the other hand, UPPAAL offers extra constructs like global and local clocks and time invariants that are not needed for modeling of a hierarchical configuration model.

As can be seen in Table 4, there is an exponential explosion in the number of states in the Freeconf model. The UPPAAL model-checker can handle a large number of states but even for a small Freeconf model like the one presented in Figure 5c, it already consumes over two gigabytes of memory to finish verification of a single query. Using UPPAAL to verify other (possibly) heterogeneous configuration hierarchical models would be problematic. UPPAAL also does not provide a simple way of storing a result of verification (the only report on the overall state of verification is a green or red light next to each query). When one has a functioning and verified model and adds or modifies some aspect of it, one would like to re-verify the model and get all differences in the verification results. One would also like to automatize verification by having a verified reference instance of the model for comparison. None of this is supported in UPPAAL as of yet.

7 Conclusion & Future Works

In this paper, the first step towards a (semi)automatic mechanism of dynamic rules verification has been made. A configuration hierarchical model has been defined and a syntax of propagation rules has been described. The dynamics of evolution of various properties of configuration keys in the configuration tool Freeconf has been briefly introduced as an instance of the general hierarchical model. This Freeconf model instance has been further encoded as a set of state machine automata templates in UPPAAL. The UPPAAL model-checker has been used to verify certain shapes of the Freeconf model.

In the future, a custom domain specific model-checker should be written which would utilize the knowledge of the hierarchical configuration model and the propagation rules and would allow to express a specific problem with minimum overhead. All propagation rules describing the dynamics of the problem should be held at one place for maximum readability. The model-checker should also be able to re-verify the model in the case of an update of the rules. The intention at the moment is to use CHR (Constraint Handling Rules) [Frühwirth, 2009; Frühwirth and Raiser, 2011] as a base programming language for the model-checker implementation.

8 Acknowledgments

This work was partially supported by the project of the Student Grant Agency of the Czech Technical University in Prague No. SGS11/161/OHK4/3T/14, 2011-13.

References

[Alur *et al.*, 1993] Rajeev Alur, Costas Courcoubetis, and David Dill. Model-checking in dense real-time. *Information and Computation*, 104:2–34, 1993.

[Arman, 2013] Nabil Arman. Improving rule base quality to enhance production systems performance. *International Journal of Intelligence Science*, 3(1):1–4, 2013.

[Behrmann *et al.*, 2004] Gerd Behrmann, Re David, and Kim G. Larsen. A Tutorial on Uppaal. pages 200–236. Springer, 2004.

[Brand, 2004] Sebastian Brand. *Rule-Based Constraint Propagation Theory and Applications*. PhD thesis, Universiteit van Amsterdam, 2004.

[David *et al.*, 2009] Alexandre David, Tobias Amnellough, and Martin Stiggeore. *Uppaal 4.0: Small Tutorial*, 2009.

[Fabian *et al.*, 2012] David Fabian, Radek Mařík, and Tomáš Oberhuber. Towards a formalism of configuration properties propagation. In *ConfWS'12*, pages 15–20. CEUR Workshop Proceedings, 2012.

[Fabian, 2011] David Fabian. *System for Simplified Generating of Configurations*. Master thesis, Faculty of Nuclear Sciences and Physical Engineering, Prague, 2011. in Czech.

[Fabian, 2012] David Fabian. Freeconf: A general-purpose multiplatform configuration utility. In *Doktorandské dny 2012*, pages 21–30. ČVUT v Praze, 2012.

[Frühwirth and Raiser, 2011] Thom Frühwirth and Frank Raiser. *Constraint Handling Rules: Compilation, Execution, and Analysis*. Books on Demand, 2011.

[Frühwirth, 2009] Thom Frühwirth. *Constraint Handling Rules*. Cambridge University Press, 2009.

[Liaskos *et al.*, 2005] Sotirios Liaskos, Alexei Lapouchnian, Yiqiao Wang, Yijun Yu, and Steve Easterbrook. Configuring common personal software: a requirements-driven approach. In *Proceedings of the 13th IEEE International Conference on Requirements Engineering*, RE '05, pages 9–18, Washington, DC, USA, 2005. IEEE Computer Society.

[Lukichev, 2011] Sergey Lukichev. Improving the quality of rule-based applications using the declarative verification approach. *International Journal of Knowledge Engineering and Data Mining*, 1(3):254–272, December 2011.

[Nardi and Brachman, 2003] Daniele Nardi and Ronald J. Brachman. The description logic handbook. chapter An introduction to description logics, pages 1–40. Cambridge University Press, New York, NY, USA, 2003.

[Preece and Shinghal, 1992] Alun D. Preece and Rajjan Shinghal. Verifying expert systems: A logical framework and a practical tool. *Expert Systems with Applications*, 5:421–436, 1992.

[Preece and Shinghal, 1994] Alun D. Preece and Rajjan Shinghal. Foundation and application of knowledge base verification. *Int J Intell Syst 1994;9(8):683–702. Duftschmid, S. Miksch*, 22:23–41, 1994.

[TechBase, 2012] KDE TechBase. Using KConfig XT. `http://techbase.kde.org/Development/Tutorials/Using_KConfig_XT`, 2012.

[Vlaeminck *et al.*, 2009] Hanne Vlaeminck, Joost Vennekens, and Marc Denecker. A logical framework for configuration software. In *Proceedings of the 11th ACM SIGPLAN conference on Principles and practice of declarative programming*, PPDP '09, pages 141–148, New York, NY, USA, 2009. ACM.