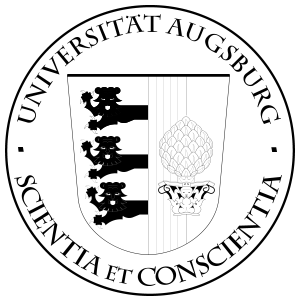
Universität Augsburg



Model Checking of Asbru

Andriy Dunets, Simon Bäumler, Arjen Hommersom, Wolfgang Reif

Report 2009-14 July 2009

Institut für Informatik
D-86135 Augsburg

Abstract

Model checking is known as an automatic verification technique which can be applied very efficiently. Formalisation of medical guidelines in Asbru enables us to apply formal verification on medical guidelines. In this deliverable we show, how model checking can be applied to Asbru plans. After a short survey, which model checking techniques are suitable for verification of Asbru, we show how a finite state model can be constructed from Asbru plans and how they can automatically compiled. Further, we discuss what kind of properties can be verified with model checking.

Contents

Chapter 1

Introduction

Model checking is a verification technique, which is used to verify that a given finite state model satisfies a property specified as temporal logic formula. In contrast to interactive verification this technique is usually fully automatic, i.e. the verification process itself needs no user interaction. Since its introduction 1986 by Clarke, Emerson and Sistla[1], model checking has evolved to a successful and reliable verification technique, which is further developed by a big research community.

In the Protocure I project we were developing only the interactive verification of Asbru. After this project was accomplished and we gained much experience with the verification of Asbru, we decided to investigate the use of alternative verification techniques. Although, interactive verification is the central theme of our research within the scope of the Protocure II project, investigating other alternative verification techniques is also a topic for the Protocure II project. According to the formal semantics of Asbru, hierarchical plans that model the protocol are represented by state charts. State charts are hierarchical state transition systems and are highly suitable for model checking. Also both interactive theorem proving and model checking are using linear temporal logic for the formulation of properties. Therefore, the idea of applying model checking to the verification of Asbru models seems a logical choice. In the beginning we decided to realize a flexible and a pragmatic solution in order to evaluate these approach. Already first experiments with model checking of Asbru produced very promising results. So, we decided to go even further and apply model checking to the verification of a wide range of properties, i.e. structural and medical properties. Further, we also applied model checking for critiquing of medical guidelines. In this technical report we will deliver a detailed insight in the different aspects of model checking technique used for the verification of Asbru. Parts of this work were previously published in [2] and [3].

In Section 2 we will give an introduction to temporal logic which builds the basis for the verification of Asbru. Section 3 compares different model checking

techniques and their tools, which are potentially suitable for the verification of Asbru. The choice of an appropriate model checking tool is the most crucial decision in the whole process. It is particularly difficult to make because there are so many tools for this popular verification technique.

The key to the verification by model checking is the construction of a finite state model of the verified system. Section 4 gives a detailed description of this process and discusses the correctness of results. In many cases the construction of semantically equivalent finite state model for an Asbru model is done automatically and is not an expensive step. Nevertheless, Asbru makes it possible to construct models which are infinite in state space and can not be straightly mapped on a finite state transition system. In this case another level of abstraction is needed. Section 5 discusses the implications on the verification process done by the application of an abstraction to the original model. The well known counter example guided abstraction refinement (CEGAR) process is shortly discussed and the link to the verification of Asbru properties is established. Section 6 explains how the compiler works and Section 7 describes the main classes of properties we considered for the verification by model checking (important parts of this are also discussed in deliverable D4.2c of the Protocure II project). Section 8 summarizes the general results we achieved by implementing this approach. Section 9 explains how the whole approach of model checking of Asbru can be enhanced by considering medical background knowledge. During the verification of many medical properties we have made experiences showing how important it is to consider the additional medical knowledge during the verification. Section 10 summarizes this deliverable.

All of the model checking verification results, which are a big part of workpackage 4.3, are presented in Deliverable 4.2c [4] together with the verification results of the interactive verification.

Chapter 2

Temporal Logic

As medical management is a time-oriented process, diagnostic and treatment actions described in guidelines are performed in a temporal setting. It has been shown previously that the step-wise, possibly iterative, execution of a guideline can be described precisely by means of temporal logic [5]. This is a modal logic, where relationships between worlds in the usual possible-world semantics of modal logic is understood as time order.

Medical guidelines can be interpreted as a Kripke structure M over a set of atomic propositions AP, which formally is defined as a four tuple $M = (S, S_0, R, L)$ where

- S is a finite set of states.
- $S_0 \subseteq S$ is the set of initial states.
- $R \subseteq S \times S$ is a transition relation that must be total, i.e., for every $s \in S$ there is a state $s' \in S$ such that $R(s, s')$.
- $L : S \rightarrow 2^{AP}$ is a function that labels each state with the set of atomic propositions true in that state.

A *path* in the model M from a state s is an infinite sequence $\pi = s_0 s_1 s_2 \ldots$ such that $s_0 = s$ and $R(s_i, s_{i+1})$ holds for all $i \geq 0$. With π^i we denote the suffix of π starting at s_i.

2.1 Branching-time Logic

The logic typically used for specifying properties of the Kripke structures is branching-time logic or Computation Tree Logic (CTL) [6, 7, 8]. This logic uses atomic propositions and Boolean connectives (e.g., $\neg, \vee, \wedge$) to build up

more complicated expressions for describing properties of states. Furthermore, CTL formulas can be composed of *path quantifiers* and *temporal operators* for describing properties of *computation trees*, i.e., the tree that is formed by designating a state in the Kripke structure as the initial state and then unwinding the structure into an infinite tree according to the transition relation R with the initial state as root. The path quantifiers are **A** and **E** to specify that all of the paths or some of the paths starting at a specific state have some property. The temporal operators describe properties of a path through the tree. The five temporal operators used are **X**, **G**, **F**, **U**, and **R**. With $\mathbf{X}\varphi$ being true if φ holds in the next state, $\mathbf{G}\varphi$ if φ holds in the current state and all future states, $\mathbf{F}\varphi$ if φ holds in some state in the future (or is true in the current state), $\varphi\mathbf{U}\psi$ if φ holds until ψ holds, i.e., there is a state on the path where ψ holds and in every preceding state φ holds, and $\varphi\mathbf{R}\psi$ if ψ holds along the path up to and including the first state where φ holds, however φ is not required to hold eventually.

In CTL there are two types of formulas: *state formulas*, which are true in a specific state, and *path formulas*, which are true along a specific path. The syntax of state and path formulas is defined as follows:

- Each atomic proposition is a state formula.
- If f and g are state formulas, then $\neg f$, $f \vee g$, and $f \wedge g$ are state formulas.
- If f is a path formula, then $\mathbf{E}f$ and $\mathbf{A}f$ are state formulas.
- If f and g are state formulas, then $\mathbf{X}f$, $\mathbf{G}f$, $\mathbf{F}f$, $f\mathbf{U}g$, and $f\mathbf{R}g$ are path formulas.

The semantics of CTL is defined with respect to a Kripke structure M. Given a state formula f, the notation $M, s \models f$ denotes that f holds in state s of the Kripke structure M. Assuming that f_1 and f_2 are state formulas and g_1 and g_2 are path formulas, the relation $\models$ can be defined inductively as shown in Figure 2.1.

2.2 Fairness Constraints

Additionally to the choice of CTL or LTL, one also has to consider the issue of *fairness*. In the case of medical guidelines, we are only interested in the correctness along fair computation paths. This means that we assume that medical actions that depend on a manual interaction with a physician are not ignored indefinitely by the physician. For example, obtaining the value of a medical quantity or initiating a medical action. Such properties cannot be directly specified in CTL. In case we need to deal with fairness in CTL, we modify the semantics as given in Figure 2.1 slightly and refer to the logic by *fair semantics*. A *fairness constraint* can be an arbitrary set of states, usually represented by a logical formula. A fair path is then a path that contains an

$M, s \models p$	$\Leftrightarrow$	$p \in L(s)$
$M, s \models \neg f_1$	$\Leftrightarrow$	$M, s \not\models f_1$
$M, s \models f_1 \vee f_2$	$\Leftrightarrow$	$M, s \models f_1$ or $M, s \models f_2$
$M, s \models f_1 \wedge f_2$	$\Leftrightarrow$	$M, s \models f_1$ and $M, s \models f_2$
$M, s \models \mathbf{E}g_1$	$\Leftrightarrow$	there is a path π from s such that $M, s \models g_1$
$M, s \models \mathbf{A}g_1$	$\Leftrightarrow$	for every path π starting from s such that $M, s \models g_1$
$M, \pi \models f_1$	$\Leftrightarrow$	s is the first state of π and $M, s \models f_1$
$M, \pi \models \neg g_1$	$\Leftrightarrow$	$M, \pi \not\models g_1$
$M, \pi \models g_1 \vee g_2$	$\Leftrightarrow$	$M, \pi \models g_1$ or $M, \pi \models g_2$
$M, \pi \models g_1 \wedge g_2$	$\Leftrightarrow$	$M, \pi \models g_1$ and $M, s \models g_2$
$M, \pi \models \mathbf{X}g_1$	$\Leftrightarrow$	$M, \pi^1 \models g_1$
$M, \pi \models \mathbf{F}g_1$	$\Leftrightarrow$	there exists a $k \geq 0$ such that $M, \pi^k \models g_1$
$M, \pi \models \mathbf{G}g_1$	$\Leftrightarrow$	for all $k \geq 0, M, \pi^k \models g_1$
$M, \pi \models g_1 \mathbf{U} g_2$	$\Leftrightarrow$	there exists a $k \geq 0$ such that $M, \pi^k \models g_2$ and for all $0 \leq j < k, M, \pi^j \models g_1$
$M, \pi \models g_1 \mathbf{R} g_2$	$\Leftrightarrow$	there exists a $k \geq 0$ such that $M, \pi^k \models g_1$ and for all $0 \leq j < k, M, \pi^j \models g_2$

Figure 2.1: Semantics of CTL with f_1 and f_2 representing state formulas and g_1 and g_2 representing path formulas

element of each fairness constraint infinitely often. The path quantifiers are restricted to fair paths. We write $M, s \models_F f$ to indicate that the state formula f is true in state s in the Kripke structure M under the fair semantics. For the fairness semantics, only the first, fifth, and sixth clause in Figure 2.1 are modified into:

$M, s \models_F p$	$\Leftrightarrow$	there exists a fair path starting from s and $p \in L(s)$
$M, s \models_F \mathbf{E}g_1$	$\Leftrightarrow$	there exists a fair path π from s such that $M, s \models_F g_1$
$M, s \models_F \mathbf{A}g_1$	$\Leftrightarrow$	for all fair paths π starting from s such that $M, s \models_F g_1$

2.3 Linear Temporal Logic

Linear Temporal Logic (LTL) provides operators for describing events along a single computation path. Each formula is of the form $\mathbf{A}f$, with f being a path formula, which is either an atomic proposition or inductively defined as $\neg f$, $f \vee g$, $f \wedge g$, $\mathbf{X}f$, $\mathbf{F}f$, $\mathbf{G}f$, or $f\mathbf{R}g$ with f, g formulas.

The semantics of LTL with respect to a Kripke structure M is shown in Figure 2.2. The notation $M, s \models f$ means that f holds at state s in the Kripke structure M. Similarly, in case f is a path formula, $M, s \models f$ means that f holds along path π in the Kripke structure M. A path π in M is defined as an infinite sequence $\pi = s_0 s_1 s_2 \ldots$ such that $s_0 = s$ and $R(s_i, s_{i+1})$ holds for all $i \geq 0$.

$\pi \models p$	$\Leftrightarrow$	s is the first state of π and $p \in L(s)$
$\pi \models \neg f_1$	$\Leftrightarrow$	$\pi \not\models f_1$
$\pi \models f_1 \vee f_2$	$\Leftrightarrow$	$\pi \models f_1$ or $\pi \models f_2$
$\pi \models f_1 \wedge f_2$	$\Leftrightarrow$	$\pi \models f_1$ and $\pi \models f_2$
$\pi \models \mathbf{X} f_1$	$\Leftrightarrow$	$\pi^1 \models f_1$
$\pi \models \mathbf{F} f_1$	$\Leftrightarrow$	there exists a $k \geq 0$ such that $\pi^k \models f_1$
$\pi \models \mathbf{G} f_1$	$\Leftrightarrow$	for all $k \geq 0, \pi^k \models f_1$
$\pi \models f_1 \mathbf{U} f_2$	$\Leftrightarrow$	there exists a $k \geq 0$ such that $\pi^k \models f_2$ and for all $0 \leq j < k, \pi^j \models f_1$
$\pi \models f_1 \mathbf{R} f_2$	$\Leftrightarrow$	there exists a $k \geq 0$ such that $\pi^k \models f_1$ and for all $0 \leq j < k, \pi^j \models f_2$

Figure 2.2: Semantics of LTL

With π^i we denote the suffix of π starting at s_i. We will often shorten $M, s \models f$ to $s \models f$ as the model M is clear from the context.

Chapter 3

Comparison of Model Checking Techniques and Tools

To verify a property practically all model checking algorithms compute the whole state space of the model. The need of this computation is usually the main disadvantage of model checking compared to other verification techniques, as the state space can grow exponentially to the number of used variables or concurrent components (this problem is also called *state explosion problem*).

There are various newer model checking techniques that reduce the run-time complexity of the verification and therefore help to counter the state explosion problem. On the other hand, to our knowledge no existing technique is capable of eliminating this problem entirely. In Section 3.1 we compare complexity issues of LTL and CTL model checking. The following Sections 3.2 to 3.6 discuss some different model checking techniques and tools. The chapter concludes in Section 3.7 with a comparison of the applicability of different model checking tools with respect to Asbru. For more details about model checking in general we recommend [9].

3.1 Model Checking Complexity Issues

The complexity of model-checking with LTL and CTL formulas is well understood. Suppose we are given a program of size n and temporal specification of size m. For a CTL formula, model checkers run in $O(mn)$ [1], whereas, for LTL formulas, the complexity is $O(n2^m)$ [10]. The latter (probably) cannot be improved as model checking with LTL has been found to be PSPACE-complete. The linear-branching model checking problem has been found to be

EXPSPACE-hard [11], making it strictly more difficult than model checking LTL formulas alone. Note that this result heavily relies on the blowup of LTL formulas to automata, which suggests that it would be possible to find a better result if one restricts the form of LTL formulas.

The relative low computational complexity of CTL has made it the industrial standart for model checking for decades. However, the discussion whether to use CTL or linear-time temporal logic (LTL) for model checking is far from over, as LTL is usually more intuitive and is better suited as a specification language. For example, recently, Vardi [12] revived this discussion by summing up the advantages of linear-time frameworks in terms of expressiveness, compositionality, property-specific abstractions, uniformity and the use of bounded model checking (cf. Section 3.5).

Clearly, LTL is not the only alternative logic that can be applied for model checking. For example, alternating time temporal logic [13, 14] has been studied in the context of model checking for open systems and games in which there is an alternation between moves. However, a complete overview of alternative logics is beyond the scope of this deliverable.

3.2 Automata based Model Checking

Automata based model checking is usually used to verify LTL properties. It uses Büchi-Automatons [15], which are in principle an extension of finite state automata to use infinite words as input. The principle of automata based model checking is to compute an Büchi-Automaton for the system and another for the negated LTL property[16]. The check, whether the property is valid is then reduced to the check, whether the intersection of both automata is empty. This technique can be further enhanced by heuristics such as partial order reduction and by exploiting symmetry to reduce the state explosion.

One of the most popular automata based model checking tools is SPIN [17] which was developed by Holzmann and Peled at Bell Laboratories since 1980 and is freely available since 1991. SPIN makes use of partial order reduction and is mainly used for verification of asynchronous systems and in particular communication protocols. It features a rich C-like input language called PROMELA which explicitly supports channel and message passing. SPIN is able, beside verifying LTL properties, to detect deadlocks and unreachable code.

3.3 Symbolic Model Checking

Symbolic Model Checking was invented around 1987 and is commonly considered as a breakthrough in model checking techniques [18, 19]. The idea behind symbolic model checking is to use a boolean encoding for the state transition system and the set of states. The size of this encoding is greatly reduced by

representing sets of states and the state transition system by BDDs (Binary Decision Diagrams) [20], which are traditionally used to represent boolean functions. The algorithm for symbolic model checking uses fixpoint operators, which operate on the BDDs.

The introduction of symbolic model checking was very successful in the last decade in the research community as well as in industry application. It is especially common in the verification of hardware designs.

One of the most popular tools for symbolic model checking is the Symbolic Model Verifier, SMV [19]. The original SMV was developed at Carnegie Mellon University (and therefore often called CMU SMV). It is a CTL model checking tool that supports fairness and the use of modules. CMU SMV is currently not further developed, the latest update is November 2001. A branch of SMV is developed by the Cadence Berkeley Labs and is commonly called Cadence SMV. It supports CTL and LTL model checking, bounded model checking (see Section 3.5) and has a variety of special techniques, e.g. for refinement verification, temporal case splitting, symmetry reductions, data type reductions, and uninterpreted functions. Another branch of SMV is NuSMV [21]. It is also an extension of the original CMU SMV and supports features similar to Cadence SMV. All three variants support the same input language of the original CMU SMV which is described in [22].

Another model checking tool which supports symbolic model checking is SAL (Symbolic Analysis Laboratory) [23], which is developed by a cooperation of SRI International, Stanford University, UC Berkely and Verimag. Besides symbolic model checking, this tool supports explicit-state model checking, witness-producing model checking (not discussed here), and bounded model checking (see Section 3.5).

3.4 Timed Model Checking

Timed model checking is used for the verification of real-timed systems where time and timing issues play an important part. In comparison to other model checking techniques, where all variables can only have discreet values, timed model checking has to deal with continuous time values. This increases complexity and difficulty of the model checking algorithms for such systems.

A popular tool for timed model checking is Uppaal [24], developed at the Universities of Uppsala and Aalborg. It uses timed automata extended with data types as bounded integers, arrays, etc. Other tools for timed model checking are Kronos, Raven and Verus.

Similar, but more general, to timed model checking is model checking of hybrid systems, which can contain both, discreet and continuous values. A model checking tool for such systems is HyTech [25], developed by Henzinger, Ho and Wong-Toi.

3.5 Bounded Model Checking

As discussed in the previous Sections, symbolic model checking is often based on the use of Binary Decision Diagrams (BDD). This approach has been very successful in the last decade, which can be observed by the fact that this technique is used on a wide industrial scale. Nonetheless, BDDs may grow exponentially, which restricts the size of the model that can be verified efficiently.

An alternative technique that can be used is so-called *Bounded Model Checking* (BMC), which is based on propositional SAT solving [26]. This technique was first introduced by Biere et al. [27] in 1999. So far, it has been shown that the BMC may be used to verify systems that could not be verified with BDD techniques. However, at the same time, problems exist that can be solved more efficiently using BDD techniques.

The idea of BMC is to search for a counter-model in executions of whose length is bounded by some constant k. Typically, the model checker iterates from 0 to the pre-defined upper bound k and terminates once it has found a counter-example. As a consequence, this method is incomplete as the smallest counter-example may be beyond the upper bound provided by the user. Experiments have shown that if the counter-example can be found within a bound of around 60 or 80 cycles, depending on the SAT solver and the model, BMC outperforms BDD approaches.

Tools that support bounded model checking are for example SMV and SAL Both tools were described more closely in Section 3.3. Results of our experiments with bounded model checking are described in Section 8.2.

3.6 Modular Model Checking

In model checking literature, the approach of verifying a restricted part of the system (i.e., the protocol consistent with medical practice) is called *modular verification* (cf. [28]). In the assumed-guarantee paradigm, the specification of a module consists of a specification of guaranteed behavior assuming that the system behaves in a certain way, i.e., the assumed behavior. In our work, the assumed behavior is written down in a linear temporal logic and the guaranteed behavior in branching temporal logic. The assume-guarantee assertions are written down as $[\varphi]M\langle\psi\rangle$, meaning that the branching temporal formula ψ holds in the computation tree that consists of all computations of the program, described by M, that satisfy the linear temporal formula φ. Because we have a mixture of linear and branching formulas, this has been named the linear-branching model checking problem [11].

Modular model checking is for example supported by SMV, which is described in Section 3.3.

3.7 Suitable Model Checking Tools for Asbru

To evaluate the possibilities of model checking of Asbru-plans we needed an appropriate model checking tool. Considering the work, which is necessary to find a good representation of Asbru-plans in a model checkers input language, program a compiler (which is a necessity considering the size of most practical relevant Asbru-plans) and test the model checker with practical relevant examples, we decided to concentrate in this work package on a single model checking tool. Yet, as we wanted to be as flexible as possible, we needed a tool which offered us the possibility to use many different technologies. This is especially interesting as there existed no research about model checking of clinical guidelines at the beginning of our work and it was not predictable, which techniques would be most successful and which not. Also, another requirement on the tool we used was, that it was stable and successfully tested for practical applications.

The first decision we had to make was, whether we wanted to use timed model checking or not. Timed model checking offers us the ability to check the rich time annotations, which are possible in Asbru. But on closer inspection timed model checking of Asbru plans has some problems: A disadvantage of timed model checking is, that all tools for timed model checking rely only on a single model checking technique which are specialized on timed systems. Tools like SMV or SAL offer a much broader scope of techniques and promise much better applicability for systems or properties, which are not time dependent. Also, one of our goals was to provide a fully automatic translation of Asbru models. As the evaluation of many time annotations relies on history, a suitable abstraction for the history must be found. This is hard to do generic. As time was not relevant in all properties for the breastcancer case study, which we used for this project, we decided against using timed model checking. Nevertheless, timed model checking is an interesting technique which could be very relevant for the verification of time dependent properties of Asbru models.

We chose SMV as tool for verification with Asbru. As mentioned above, it is a popular, stable and technically mature tool, which offers much flexibility, as it supports many different model checking techniques. Of all non timed tools we considered, SMV seems to us the best choice for verification of Asbru.

Chapter 4

Construction of a finite state model

The crucial point in model checking and verification in general is the computation of an optimal abstraction of the examined system. Recently, much research has concentrated on tackling the state explosion problem. A variety of abstraction techniques have been developed, for example [29], [30], [31], [32] and [33]. An important observation is the basis for these investigations: there are different aspects in the concrete model that have no impact on the checked property and can be abstracted in such a way that the size of the model is drastically reduced, but the property is still safely verified, i.e. the satisfaction of a property over the abstract model implies its satisfaction over the concrete model.

Fortunately, the definition of the formal semantics of Asbru is done using state charts, see [34]. State charts are hierarchical state transition systems and therefore their semantics can be very intuitively represented by finite state transition systems. Although state charts only represent the definition of the semantics and the Asbru semantics itself is implemented in KIV using parallel programs of the dynamic logic ITL (Interval Temporal Logic)[35], the equivalence between both is guaranteed and we have not to bother about it.

The Asbru model of a medical protocol consists for the most part of the definitions of Asbru plans which model the control structure of the protocol. Their semantics is defined by means of state charts, as mentioned before, and do not represent a big problem to express using finite state transition systems (even no abstraction is needed). Nevertheless, one can ask about the patient data input during the execution of plans. In order to make verification results more general we do not restrict the Asbru model to some specific patient group or with other words we do not introduce the patient model. This means patient data input has nondeterministic nature. Although sometimes it makes sense to restrict the model by adding some medical knowledge about the behavior of patients etc.

Though a bit difficult to implement, the SMV model of Asbru plans is finite and no abstraction is needed in order to represent it as a finite state model. On the other hand, the patient data input, which has an impact on the execution of plans, is potentially infinite and therefore must sometimes be abstracted to make it suitable for model checking. Fortunately Asbru supports abstraction of patient data in an appropriate way. Therefore making patient data finite is not a problem. Still it can become a difficult issue if we have Asbru conditions with time annotations in the model. In this case we have to apply a generic abstraction, which is realized in the context of state charts (modeling behavior of Asbru models) using well known concept of *environment*. We will address issues concerning data input to Asbru plans in the section about environment.

In the following section we will consider the detailed representation of the semantics of a general Asbru plan using finite state transition system in its ultimate form, i.e. SMV notation.

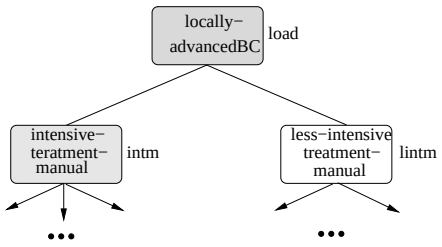


Figure 4.1: Asbru plan *locally-advanced-BC*

4.1 Finite state model of Asbru plans

For the better understanding of the finite state model of Asbru plans we will also shortly review some relevant parts of the Asbru semantics, for more details on the semantics we refer to [34]. Using concrete examples from the breast cancer case study we will explain the construction of finite state model for the plan state model, different types of plan execution controls as well as modeling some special features of Asbru.

Consider an example from chapter 3 of breast cancer guideline, see Fig. 4.1. Plan *locally-advanced-BC* is the highest plan in the whole plan hierarchy of chapter 3. It has two sub plans *intm* and *lintm*. The plan body type of *locally-advanced-BC* plans is *anyorder* with *wait-for intm*, i.e. after the plan is activated it begins to activate its children sequentially (as far as the candidates become *ready*) but in a nondeterministic order. According to the formal semantics of Asbru this behavior is modeled by the state chart represented on Fig. 4.2.

The hierarchical state automaton, represented in Fig. 4.2, starts in the state *Inactive* and switches to the state *Considered* as far as the signal *Consider* is

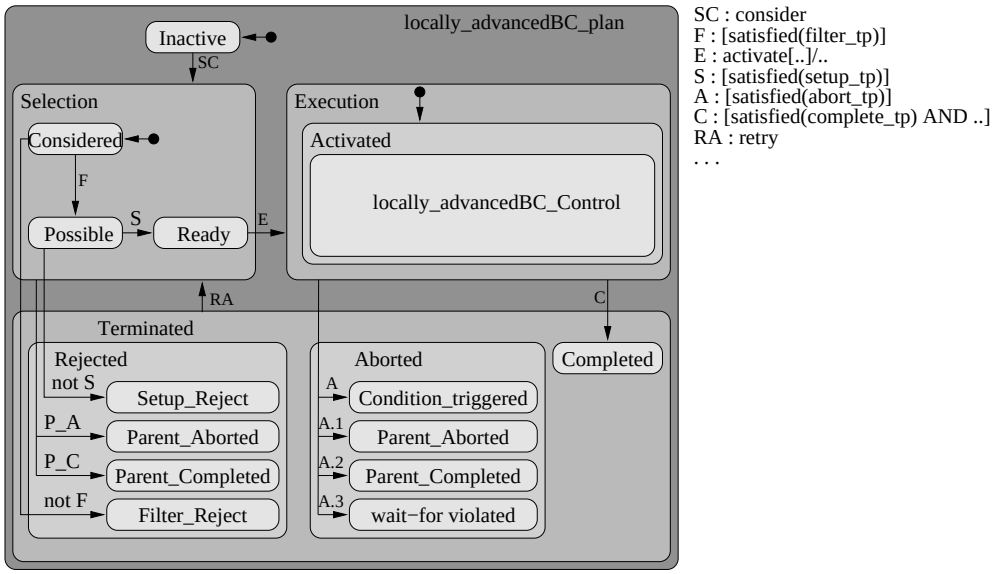


Figure 4.2: State chart modeling behavior of the plan *locally-advanced-BC*

received from the parent plan and so forth. If the state *Ready* is reached the plan waits for the signal *Activate* from its parent in order to be activated. In the state *Activated* the sub state called *locally-advanced-BC-Control* becomes active. This state is represented by a state chart which models the control of the plan *locally-advanced-BC* and is responsible for the broadcasting/unicasting *Consider-/Activate*-signals to the sub plans according to sub plans execution strategy (sequentially, in any order, unordered or in parallel). All of the condition are generated for every plan and included in the SMV model. The following SMV code represents the definitions of the conditions for the *locally-advanced-BC* plan:

```
DEFINE
loadp_is_terminated := (loadp_state = rejected) | (loadp_state = aborted) |
                                                  (loadp_state = completed);
loadp_is_aborted := (loadp_state = rejected) | (loadp_state = aborted);
loadp_consider_condition := 1;
loadp_filter_condition := diagnosis = locally-advanced-BC;
loadp_setup_condition := 1;
loadp_activate_condition := (1);
loadp_parentterm_condition := 0;
loadp_reject_condition := loadp_parentterm_condition |
                          (loadp_state = considered & !loadp_filter_condition);
loadp_abort_multi_condition := loadp_abort_condition | loadp_parentterm_condition |
                                                              loadp_child_is_aborted;
loadp_abort_condition := 0;
loadp_complete_multi_condition := (loadp_complete_condition & loadp_waitfor_condition);
loadp_complete_condition := 1;
```

```
loadp_waitfor_condition := intmp_state=completed;
loadp_is_in_selection_phase := (loadp_state = considered) | (loadp_state = possible) |
                                                              (loadp_state = ready);
loadp_retry_condition := 0;
loadp_child_is_aborted := intmp_is_aborted;
loadp_ifthen_condition := 1;
```

For example, according to the definition of the *loadp_complete_multi_condition*, the plan can complete only if its complete condition is satisfied and the wait for strategy is taken into account. The definition of the *loadp_waitfor_condition* requires the completion of the *intm* sub plan in order the parent can complete. The boolean value of those conditions triggers in specific states various transitions of plan state. Corresponding to the state chart in Figure 4.2 the transition relation for the plan state variable *loadp_state* is defined as follows (using SMV syntax representation):

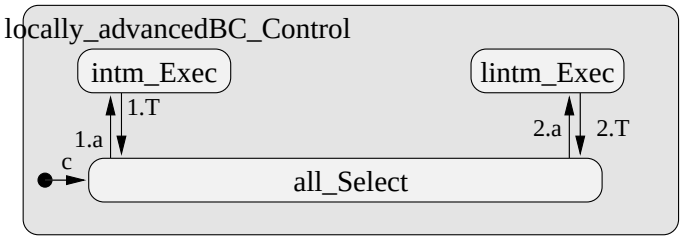
```
MODULE load_plan(loadp_state,loadp_parentterm_condition,loadp_consider_condition,
                 loadp_filter_condition,loadp_is_terminated,loadp_is_in_selection_phase,
                 loadp_setup_condition,loadp_reject_condition,loadp_activate_condition,
                 loadp_abort_multi_condition,loadp_complete_multi_condition,
                 loadp_retry_condition)
ASSIGN
 init(loadp_state) := inactive;
 next(loadp_state) := case
   loadp_parentterm_condition : inactive;
   loadp_is_terminated & loadp_retry_condition : considered;
   loadp_state = inactive & loadp_consider_condition : considered;
   loadp_is_in_selection_phase & loadp_reject_condition : rejected;
   loadp_state = considered & loadp_filter_condition : possible;
   loadp_state = possible & loadp_setup_condition : ready;
   loadp_state = ready & loadp_activate_condition : activated;
   loadp_state = activated & loadp_complete_multi_condition : completed;
   loadp_state = activated & loadp_abort_multi_condition : aborted;
   1 : loadp_state;
 esac;
```

In case of *locally-advanced-BC* plan both conditions *loadp_consider_condition* and *loadp_activate_condition* are just set to *1*, i.e. *true*, because this plan is the highest in the hierarchy and has no parent plan. But, for example, these conditions for the plan *intm* depend on *Consider/Activate* signals of the parent plan *load*:

```
intmp_consider_condition := (loadp_control_consider_signal = intm) |
                            (loadp_control_consider_signal = all);
intmp_activate_condition := ((loadp_control_activate_signal = intm) |
                             (loadp_control_activate_signal = all));
```

The condition is evaluated to *true* if and only if the signal is received by the plan or the signal is broadcasted by the parent (*loadp_control_consider_signal = all*). Depending on the type of control of the parent plan, the transition relation for control signals can be defined in a different way. For example the plan *load* has *anyorder* type of control. As far as plan *load* reaches the state *Activated* that sub

state *locally-advanced-BC-control* becomes and the corresponding state chart cares for sending of control signals, see state chart in Figure 4.3. Firstly, this state chart makes the transition to the state *all_Select*. During this transition, the action *c* is performed, which just broadcasts the *Consider* signal to all of the sub plans.



c : /intensive_treatment_manual_plan.consider;
less_intensive_treatment_manual_plan.consider;

1.a : [in(intensive_treatment_manual_plan.Selected)]
/intensive_treatment_manual_plan.activate;

1.T : [in(intensive_treatment_manual_plan.Terminated)]

2.a : . . .

Figure 4.3: State chart modeling control of the plan *locally-advanced-BC*

In the state *all_Select* each of the transitions *1.a* and *1.a* can be made and the *Activated* signal sent to the corresponding sub plan. This exactly corresponds to the nondeterministic execution order of the *anyorder* control type. The state chart in Fig. 4.3 can be translated to a finite state transition system as follows:

```
MODULE load_plan_control(loadp_state,loadp_control_consider_signal,
                         loadp_control_activate_signal,
                         intmp_state,intmp_retry_condition,
                         lintmp_state,lintmp_retry_condition)
ASSIGN
 init(loadp_control_consider_signal) := none;
 next(loadp_control_consider_signal) := case
   loadp_state = activated : all;
   !loadp_state = activated : none;
 esac;

 init(loadp_control_activate_signal) := none;
 next(loadp_control_activate_signal) := case
   loadp_state = activated :  case

     loadp_control_activate_signal = none : case
       intmp_state = ready & lintmp_state = ready : {intm,lintm};
       !intmp_state = ready & lintmp_state = ready : {lintm};
```

```
      intmp_state = ready & !lintmp_state = ready : {intm};
      !intmp_state = ready & !lintmp_state = ready : none;
      1 : loadp_control_activate_signal;
    esac;

    loadp_control_activate_signal = intm & (intmp_state = aborted |
                                        intmp_state = completed) :
    case
      lintmp_state = ready : {lintm};
      !lintmp_state = ready : none;
      1 : loadp_control_activate_signal;
    esac;

     loadp_control_activate_signal = lintm & (lintmp_state = aborted |
                                         lintmp_state = completed) :
    case
      intmp_state = ready : {intm};
      !intmp_state = ready : none;
      1 : loadp_control_activate_signal;
    esac;

    1 : loadp_control_activate_signal;
  esac;
  !loadp_state = activated : none;
esac;
```

In the first part the transition relation for the *Consider* signal is defined. The definition of *loadp_control_activate_signal* is a bit more complicated since we have to take all possibilities into account and still preserve the nondeterministic order of activation sub plans. As a result, we obtain an exponential number of cases. For example, in chapter 1 from the breast cancer guideline the definition of *anyorder* control of the plan *treat* with 8 sub plans required ca. 700 lines of code.

The definition of control signals for other control types is comparatively simple. Let us consider an abstract example of the parent plan P with three children A, B, C. In case of the *sequential* control type the definition in SMV notation corresponding to the Asbru semantics would look like:

```
init(P_control_consider_signal) := none;
next(P_control_consider_signal) := case
   P_state = activated : case
        A_state = inactive : A;
        B_state = inactive & A_terminated : B;
        C_state = inactive & A_terminated & B_terminated : C;
        1 : P_control_consider_signal;
   esac;
  !P_state = activated : none;
esac;

init(P_control_activate_signal) := none;
next(P_control_activate_signal) := case
    P_state = activated : all;
   !P_state = activated : none;
esac;
```

Similarly for the *parallel* type:

```
init(P_control_consider_signal) := none;
next(P_control_consider_signal) := case
    P_state = activated : all;
   !P_state = activated : none;
esac;

init(P_control_activate_signal) := none;
next(P_control_activate_signal) := case
    P_state = activated : case
       (A_state = ready | A_terminated) &
       (B_state = ready | B_terminated) &
       (C_state = ready | C_terminated) : all
       1 : none;
    esac;
    !P_state = activated : none;
esac;
```

And the *unordered* type:

```
init(P_control_consider_signal) := none;
next(P_control_consider_signal) := case
     P_state = activated : all;
    !P_state = activated : none;
esac;

init(P_control_activate_signal) := none;
next(P_control_activate_signal) := case
    P_state = activated : all;
   !P_state = activated : none;
esac;
```

Due to the high expressiveness of the Asbru language it is also possible to specify hierarchies of plans which are not necessarily trees, but graphs. Consider the case where the child C has two parents P_1 and P_2. This situation can have an impact on the communication between both parents and the child. Firstly, if P_i terminates for some reason then C has also to terminate only if it was activated by P_i. Secondly, consider following circumstances which occur at the same time: both plans P_1 and P_2 are executed in anyorder (i.e. are considered simultaneously); one of them, say P_1 has *wait for C* execution strategy; sub plan C is activated by P_2. If now C aborts then the plan P_1 has not to terminate because C was not activated by P_1, i.e. plan C must remember the parent which activated it. This is done by introducing the new state variable *C_my_selector*. Following SMV code illustrates how this is reflected in the overall SMV model:

```
DEFINE

Cp_consider_condition := (P2p_control_consider_signal = C) | (P2p_control_consider_signal = all) |
                         (P1p_control_consider_signal = C) | (P1p_control_consider_signal = all);
Cp_activate_condition := ((P2p_control_activate_signal = C) | (P2p_control_activate_signal = all) |
                          (P1p_control_activate_signal = C) | (P1p_control_activate_signal = all));
Cp_parentterm_condition := (Cp_my_selector=P2) & (P2p_is_terminated|P2p_state=inactive)|
```

```
                          (Cp_my_selector=P1) & (P1p_is_terminated|P1p_state=inactive);

P1p_waitfor_condition := (Cp_state=completed & Cp_my_selector=P1);
P1p_waitfor_condition := (Cp_state=completed & Cp_my_selector=P1);

P2p_waitfor_condition := (Cp_state=completed & Cp_my_selector=P2);
P2p_waitfor_condition := (Cp_state=completed & Cp_my_selector=P2);

VAR

Cp_my_selector: {none,P1,P2};

ASSIGN

   init(Cp_my_selector) := none;
   next(Cp_my_selector) :=
   case
       P2p_state=activated : P2;
       P1p_state=activated : P1;
       1 : Cp_my_selector;
   esac;
```

Assuming that all patient data parameters are finite and no Asbru plan conditions are time annotated, we have seen that to this point we don't need any abstraction at all in order to construct the finite state model of the protocol. We distinguish between statical parameters which are assigned at the beginning of plans execution and dynamic parameters which are assigned during the execution of some *ask-plan*. For example, parameter diagnosis is assigned just in the next step after initialization and the parameter *more-then-focal-tumor* is asked later by the ask-plan *mcte2*:

```
VAR
 diagnosis : {metastasized-BC,locoregional-recurrent-BC,locally-advanced-BC,
              operable-BC,operable-invasive-BC,DCIS,unknown};
 more-than-focal-tumor : {unknown,0,1};
ASSIGN
 init(diagnosis) := unknown;
 next(diagnosis) := case
   diagnosis=unknown : {metastasized-BC,locoregional-recurrent-BC,locally-advanced-BC,
                        operable-BC,operable-invasive-BC,DCIS};
   1 : diagnosis;
 esac;

 init(more-than-focal-tumor) := unknown;
 next(more-than-focal-tumor) := case
   more-than-focal-tumor=unknown & mcte2p_state=activated : {0,1};
   1 : more-than-focal-tumor;
 esac;
```

On the first view it looks a bit inefficient to assign the value *unknown* to the the patient parameter *diagnosis* in the initialization and the nondeterministic assignment in the next step. But writing down *init(diagnosis) :=* {*metastasized-BC,..*}*;* would result in a quite different model as in the case of the definition

above. The definition *init(x) := {1,2};* means that the Kripke structure representing our model has two initial state: one with*x=1* and the other with *x=2*. This affects verification of the CTL properties according to the CTL semantics: CTL formula φ holds for the Kripke structure *iff* it holds for all initial states s_0.

4.2 Environment, macro- and micro-steps

Although, until now we have not applied any abstraction, but abstraction is needed for practically every Asbru model. An abstraction is always applied in the model of *user-performed*-plan. On the other hand, if we have a time annotated Asbru condition, then we also always apply an abstraction. Both abstractions present our standard abstraction and our incorporated in the SMV model compiler.

The reason why we need such an abstraction is the issue of modeling *time*. Modeling of time in Asbru usually requires some kind of history, i.e. in order to evaluate some time annotated conditions a plan must access the history. Our goal is an abstraction, which can be constructed automatically for any given Asbru model. In order to construct a finite state model an appropriate abstraction which eliminates time and history is needed. We use a simple abstraction that maps all time annotations to atomic propositions which logical value is randomly assigned in every macro step by the environment. Those random inputs can eventually generate behavior in the abstract model that is not present in the concrete model. This abstraction preserves only ACTL properties, as it is an over-approximation, which adds extra behavior to the abstract model (The logic ACTL is the set of all well-formed state formulas from CTL [36] containing no existential operators (EX and EU)).

The main weakness of this abstraction is the potential generation of false negatives during the verification of universal properties which must be eliminated by refining the model (CEGAR [37]). On the other hand the important advantage is its automatic generation. By this abstraction we shift the complexity of time and history to the environment. According to the state chart semantics inputs from the environment happen in the first micro-step of every macro-step and provide the required information about the patient needed for the controls of plans, e.g. up-to-date value of blood pressure or information about change of the bilirubin level in blood over the period of 6 hours after the start of the plan.

The concept of environment is, for example, implemented in the semantics of state charts. Usually environment aggregates the abstracted parts of the concrete model and plays an important in the verification. We implemented this concept in our SMV model by introducing a predicate *tick* which defines the beginning of the new macro step, i.e. *tick* becomes true if the system reaches a stable state (no transition can be triggered). This can be modeled as a negation of the disjunction of all possibilities were some state chart transition can be

triggered:

```
tick := !(
    (loadp_state = inactive & loadp_consider_condition)|
    (loadp_is_in_selection_phase & loadp_reject_condition)|
    (loadp_state = considered & loadp_filter_condition)|
    (loadp_state = possible & loadp_setup_condition)|
    (loadp_state = ready & loadp_activate_condition)|
    (loadp_state = activated & loadp_complete_multi_condition)|
    (loadp_state = activated & loadp_abort_multi_condition)|
    (loadp_state = aborted & loadp_retry_condition)|
...
    );
```

Almost all Asbru models contain so called *user-conditions* which are evaluated by the external signal from a doctor at some time point of the plan execution. Such conditions can mostly be found in the definitions of user performed plans. Since we do not model the time and use a standard abstraction instead, this exactly the point where an environment plays role. According to the concept of the environment external signals are send to the system only at the beginning of every macro step. These signals represent an abstract part of the system. In case of *user-conditions*, signals which evaluate the user condition are non-deterministically sent every time the variable *tick* becomes true. The main problem about such modeling is the termination of plans. It is possible that, for example, user performed plans stuck in the state *Activated* waiting for the completion signal in vain. If we want to avoid such execution traces we must add some fairness assumption to the model. In order to do this we add lines like *FAIRNESS plan_state=inactive* to the SMV model.

Chapter 5

Verification process

The abstraction we use allows us to generate the SMV model automatically. On the other hand it can introduce unrealistic behavior, which has an impact on the verification process. Luckily this problem occurs only if the Asbru model contains non-trivial (we consider the time annotation NOW as a trivial one) time-annotated conditions. So, in this chapter we consider exactly this situation and discuss how the abstraction of time annotations can affect the verification process. Otherwise (if an Asbru model contains no non-trivial time annotations) we do not bother about false-negatives or false-positives.

Let us consider the case where the finite state model contains some unrealistic traces added by an abstraction. Due to property preservation considerations we examine only ACTL properties although it is also indirectly possible to verify ECTL properties. Fig. 5.1 illustrates the general scheme of the verification process.

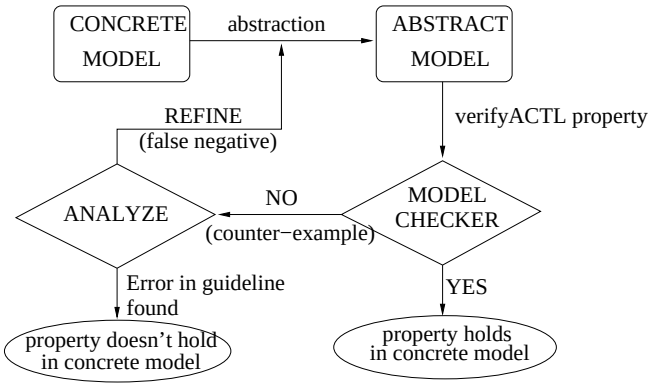


Figure 5.1: Verification in case of an over-approximation.

If ACTL property is proved to be false the corresponding counter example is generated. In the next step we have to analyze this trace to find out whether it is a real bug in the concrete model or just some unrealistic trace added by the over-approximation. We have also to verify ECTL properties since most implementation level (structural) properties are existential properties, e.g. satisfiability of Asbru conditions or non-redundancy of plans. If, for instance, a ECTL formula $EF\varphi$ must be verified, we verify first the ACTL formula $AG\neg\varphi$. If it is *true* then original formula is *false*. On the other hand, if a counter example is found then we analyze whether it is realistic one. If the found counter example trace is realistic then the original formula is *true* and generated counter example is the trace that satisfies the original formula.

Chapter 6

Compiler

Already at the beginning of model checking task in the project, as we were constructing the first prototypical SMV model for the jaundice protocol, we realized that this process can be done automatically. The Asbru model of the jaundice protocol contains approximately 40 plans. So we started with the small part of the whole hierarchy containing only 7 plans (*regular-treatments-star* hierarchy). Already this small number of plans caused several problems concerning error-prone human work. As we moved on to the higher levels in the hierarchy and the number of plans to be modeled was doubled, we still tried to construct the SMV model manually. But after spending half an hour on analyzing the counter example trace to find out that we have forgotten the negation operator in the filter condition we decided to write a compiler.

Writing a compiler for the construction of SMV models is a very crucial and responsible task. Because the confidence of SMV models depends on it. The compiler gets an Asbru model as an input and produces an SMV model as an output. With other words our compiler should apply a standard abstraction in an automatic manner on Asbru models defined by means of algebraic specifications in the theorem prover KIV. Another possibility would be, to write a general compiler for parallel programs, which are used to define the Asbru semantics in KIV. This compiler can be used on the already abstracted Asbru model (represented as a parallel program in KIV). Because of pragmatic considerations we settled for the first possibility (compiler with incorporated standard abstraction for Asbru models) as a more efficient way of constructing suitable SMV models.

The central step in writing the compiler of SMV models is to define a mapping of a finite state transition system (modeling an Asbru model) into an SMV model. This task is not as trivial as it sounds at first. The reason for this is a quite complicated general Asbru semantics as well as not always unambiguous way of encoding into SMV syntax, e.g. the transition relation can be defined using SMV-keywords *init,next* or alternatively by using SMV-keyword *TRANS*. Next

we will provide a detailed insight into the realization of these ideas. First we will see how the compiler generally works, i.e. how it gets an Asbru model from the theorem prover KIV. Then we will consider the SMV model for the part of the breast cancer case study as a running example. Using this example in order to explain the mapping from the finite state model on the document which can be used as an input for the SMV model checker. We will also describe the general structure of SMV models. And at the end we review the PPL source code of the compiler.

6.1 How does it works?

The algebraic specification of Asbru model contains in particular definitions of all Asbru plans (with all relevant data, like conditions, plan type etc.) and corresponding patient parameters. This is exactly what the compiler needs as an input. The compiler doesn't cares about the implementation of the semantics in KIV, i.e. it doesn't have to parse and interprete it. It implements its own semantics of Asbru as a finite state transition system by the application of a standard abstraction. This standard abstraction is assumed to comply to the original Asbru semantics implemented in KIV (see the corresponding chapter on Asbru finite state model).

Asbru plans are defined in KIV using abstract data type *asbru-def* which is constructed as a tuple of a set of conditions, plan type, a couple of flags for execution strategy, a list of sub plans and the waiting for sub plans completion strategy, see figure below:

```
asbru-def
  = mk-asbru-def( . .filter     : abstract-asbru-condition;
                  . .setup      : abstract-asbru-condition;
                  . .suspend    : abstract-asbru-condition;
                  . .reactivate : abstract-asbru-condition;
                  . .abort      : abstract-asbru-condition;
                  . .complete   : abstract-asbru-condition;
                  . .type       : plan-type;
                  . .retry      : bool;
                  . .subplans   : string-list;
                  . .waitfor    : wait-for;
                  . .opt-wf     : bool;
                );
```

As an example consider the plan *locally-advanced-BC* from the chapter 3 in the breast cancer guideline, see listing below. It is defined in KIV as an axiom *load-def*. All the compiler has to do is to search for axioms like *asbru(..)=mk-asbru-def(..);* in the theorem base and extract the relevant information from them. According to the axiom definition the plan *locally-advanced-BC* has a filter condition *"diagnosis=locally-advanced-BC"* and no setup-, abort- or complete-condition (default keywords in the corresponding place in the tuple), is not retried in case of rejection or abortion, has two sub plans *intm* and *lintm*, waits

only for the completion of one of the sub plans and doesn't wait for optional plans:

```
;;plan-locally-advancedBC
load-def:
     asbru('load')
     = mk-asbru-def
     (mk-aasbruc(mk-acond(lambda pdh, vh, ash, as, ac. pdh[ac]['parameter-diagnosis'] .val =
                                                  locally-advanced-BC,default-ata ),false, false),
      setup_condition,
      suspend_condition,
      resume_condition,
      abort_condition,
      complete_condition,
      anyorder,
      false,
      'intm' + 'lintm' ',
      wait-for-n(1, 'intm' + 'lintm' '),
      false);
```

Different patient parameters, which occur in the conditions are mostly finite (Asbru is very suitable for data abstraction) and are defined in data specifications:

```
data specification

diagnosis-scale = DCIS | operable-invasive-BC | locally-advanced-BC |
                    locoregional-recurrent-BC | metastasized-BC;

variables  sd, sd1, sd2, sd3 : diagnosis-scale;
           Sd, Sd1, Sd2, Sd3 : diagnosis-scale flexible;

end data specification
```

After all these information is extracted from KIV, the compiler has to translate the finite transition system due to the standard abstraction into the input document for the model checker using the SMV syntax. The general structure of the SMV document is explained in the next section.

6.2 Structure of the SMV document

According to the SMV syntax, we distinguish between state variables and *define*-variables. A specific assignment to the state variables specifies a unique state in the system and vice versa. In contrast, define-variables are defined like macros which depend in a unique way on the current assignment of the state variables and don't affect the state space. They can be seen as mere state labels which are used just for comfort.

State variables in SMV are defined using *init* and *next* keywords, i.e. definition of initial states and transition relation. While defining transition relation the

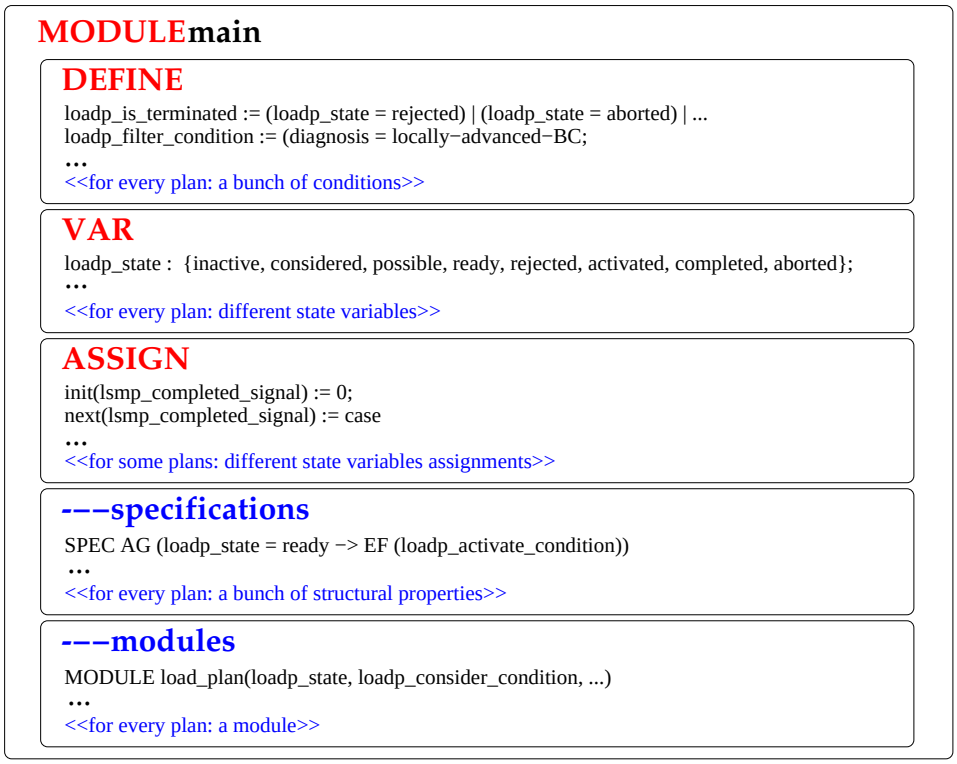


Figure 6.1: General structure of the SMV document

recursive definition is possible, e.g. *next(x) := x + 1;*. The definition of *define*-variables can not be recursive. Consequently it is natural to represent plan state variables as state variables. The same holds for patient parameters for example. On the other hand conditions is typical candidate to be declared as a define variable. Figure 6.1 shows the general structure of the SMV document: DEFINE (contains define variables declarations), VAR (declarations of state variables), ASSIGN (contains definition of initial states and transition relation for state variables), a list of specification (CTL properties to be verified) and a list of modules (containing a module for each plan). A module for each plan simply contains definition of the transition relation for plan state variable.

The document structure on the Fig. 6.1 corresponds to the CMU SMV CTL model checker syntax which is the original tool in the family of SMV model checkers. We have adopted the compiler also for the Cadence SMV syntax in order to use extended features of this LTL model checker like *assume .. using .. prove ..* paradigm. This technique allows us to add different assumptions (as LTL formulas) to the model. This is in particular very useful in case of medical guidelines as it allows to integrate different medical background knowledge in

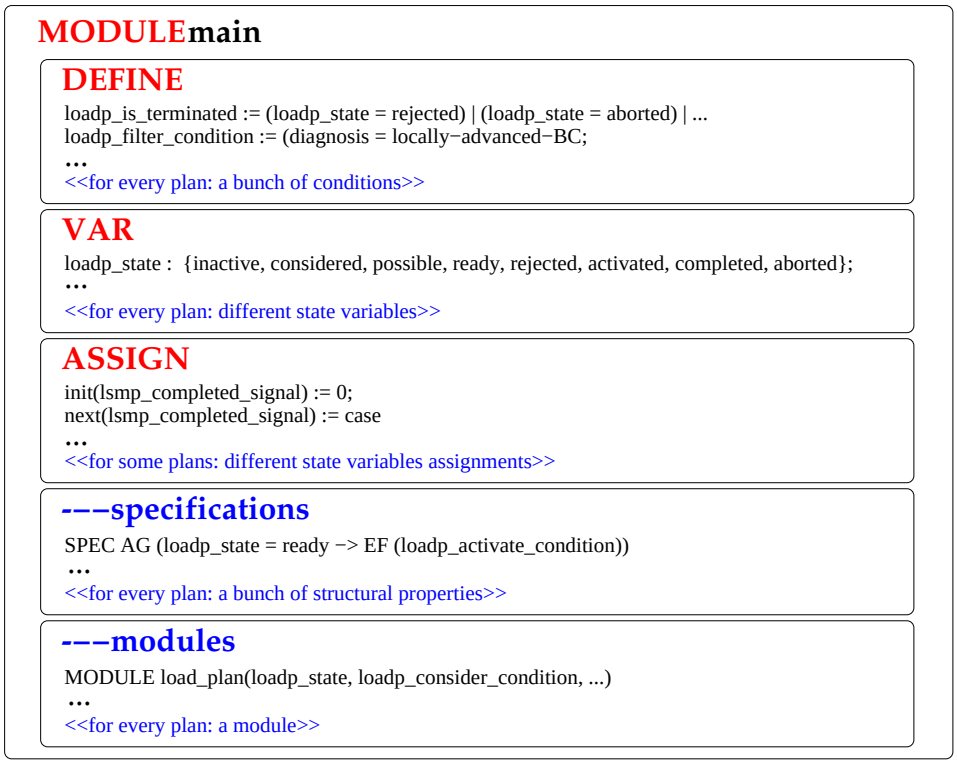


Figure 6.2: General structure of the SMV document (Cadence SMV syntax)

the model without complicated changes in the model itself (see the chapter on the background knowledge verification). See Fig. 6.2 for the Cadence SMV document example.

During the compilation process the compiler processes the list of Asbru plans, which are acquired from KIV, generates the code for each plan and places it in the corresponding sections. The concrete example of the generated SMV code (CMU SMV syntax) for the plan *locally-advanced-BC* from the chapter 3 in the breast cancer guideline is presented in the appendix.

6.3 Compiler as a PPL program

We implemented the compiler in the functional language PPL (LISP branch) which was the most suitable decision since the theorem prover KIV is implemented in LISP. The general structure of the functional program is shown in Figure 6.3.

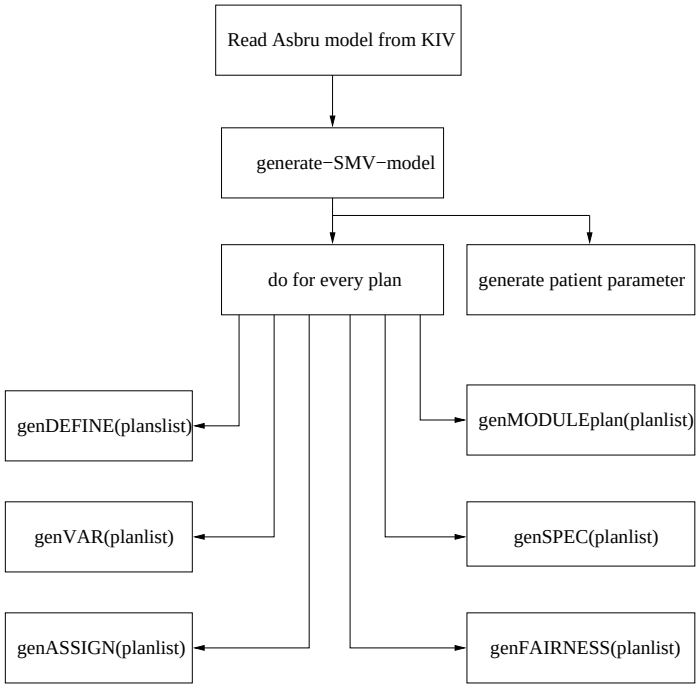


Figure 6.3: Internal structure of the PPL-program implementing compiler

Chapter 7

Properties

Every medical guideline must fulfill a number of properties and criteria. We discern between *structural* and *medical* properties [38]. Structural properties regard to the correctness of a concrete implementation of a guideline while medical properties refer to the correctness of a medical guideline itself. In the following section 7.1 we describe, what structural properties we use. Medical properties can be expressed in the Guideline Description Language (GDL), which was developed in the Protocure II project. In the section 7.2 we describe how GDL properties can be translated to SMV. Medical properties in general and the results of model checking structural and medical properties are presented in detail in Deliverable D4.2c of the Protocure II project [4].

7.1 Structural Properties

Structural properties are also called implementation level properties, as they refer to a concrete implementation of a guideline. The main goal of using them is to validate the correctness of a specific implementation of a guideline.

The translation of natural language based medical guidelines to a formal model is a huge and difficult step. Therefore, it is important to find and eliminate errors introduced into the model during the translation. Structural properties are used, to verify that the model is sane in principle, e.g., that all plans have the chance to get executed at least under some circumstances. This helps to eliminate errors introduced into the model during the translation process and to increase the confidence of the created model.

Usually, for a given guideline modellingion language (e.g. Asbru), all structural properties should be similar for all different models that are specified in this language. In our case, this guidline modelling language is Asbru.

7.1.1 Properties we use

In [38] a number of properties for Asbru-Plans are defined, which specify when an Asbru-plan is "*legal* and *meaningful*". The structural properties we used are derived from the properties presented in [38] and translated to SMV properties.

The structural properties we use in detail are the following:

Termination Every Asbru-plan must always terminate eventually.

No eternal waits It is possible during execution of an Asbru-plan that it waits for an external signal. It should not be possible that the plan waits eternally for this signal.

Completion It must be possible for every plan in the model to reach the state *completed*.

Activation Every plan in the model must be able to be activated.

Satisfiability It must be possible to satisfy every non-default Asbru-condition.

Valid sequences of states Every valid sequence of states of a plan must be possible in the model.

Usually, a violated structural property gives a hint at the kind of error in the model. So, the *termination*, *no eternal waits* and *completion* properties make it possible to detect, if a model gets stuck in a deadlock. As user performed plans are not required to terminate eventually, the violation of the properties *termination* and *no eternal waits* must not be an error in the model. If one of these properties is not satisfied, the error must be examined more closely by adding suitable fairness conditions about termination of user performed plans (see chapter 4.2).

The *activation* property helps to detect "dead code" fragments. These are usually the result of unsatisfiable or incorrect conditions. This kind of property is also often violated, if the subplan execution order or the *wait-for* condition of the superplan is incorrect. Figure 7.1 illustrates such an error.

Satisfiability properties make it possible to detect incorrect conditions. Violation of properties of the type *valid sequences of states* can have different reasons. Also, a violation of this property type must not be an error. E.g., it is allowed that a certain plan is never rejected in a model. However, this property type is very suitable to give a quick overview over all possible execution sequences of all plans, and if their actual behavior matches with their intended behavior.

7.1.2 Formalisation as SMV-Properties

In order to verify structural properties with SMV, it is necessary to specify them as temporal logic formulas. Cadence and NuSMV SMV are both able to

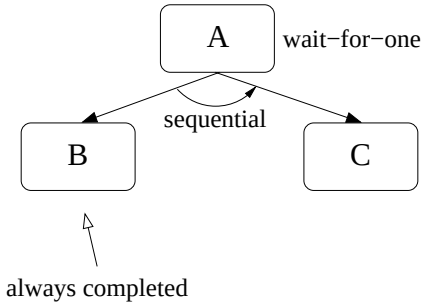


Figure 7.1: Example of an sequential execution, where plan C is never activated

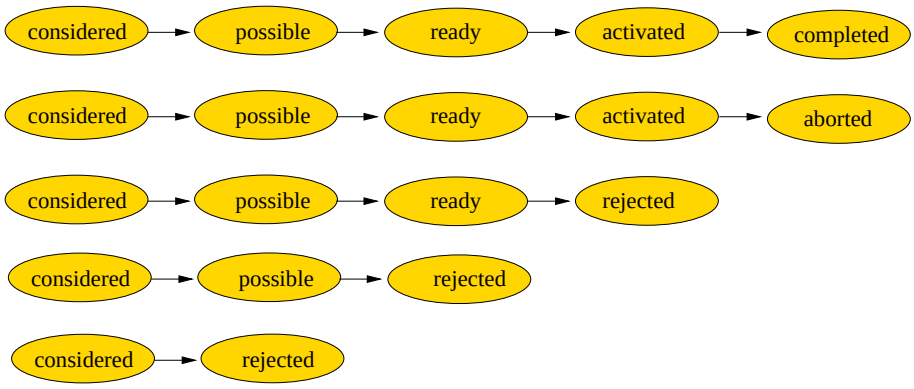


Figure 7.2: Valid sequences of states for Asbru-plans

verify linear time (LTL) and branching time (CTL) formulas. As some of the structural properties are existential (e.g. the completion property), we have chosen CTL for the formalisation of all structural properties.

In the following we describe how we formalize each structural property and present the according SMV-specification:

Termination

In every state the plan terminates or is inactive eventually in all continuations:

```
AG AF(P_state = completed | P_state = aborted |
      P_state = inactive | P_state = rejected)
```

No eternal waits

Only in the following states, a plan P can wait for external signals: *possible*, *ready* and *activated*. Therefore, we generate for any of these three

states S a property, which states the following: Every time P is in state S, in all continuations the according condition is true or the plan terminates. As example the according SMV-property for state *possible*:

```
AG(P_state = possible  -> AF(P_setup_condition | P_is_terminated))
```

The other two formulas can be defined analogously.

Completion

There exists a trace, where plan reaches the state *completed*:

```
EF P_state = completed
```

Activation

There exists a trace, where the plan reaches the state *activated*:

```
EF P_state = activated
```

Satisfiability

Every plan has four conditions: *filter-*, *setup-*, *complete-* and *abort-*condition. For every of these four conditions C There exists a trace, where the condition C is evaluated (i.e. the according plan is in a state where C is evaluated) and C is satisfied. As example the according SMV-property for the *setup-condition*:

```
EF (setup-condition & P_state=possible)
```

The other three formulas are be defined analogous.

Valid sequences of states

Figure 7.2 shows all valid sequences of states an Asbru-plan can pass through. We use the until operator, to test if such sequences exist. For example, to verify that the state sequence *inactive-considered-possible-ready-activated-completed* is possible, we use the following SMV-specification:

```
EF(P_state = inactive   & E(P_state = inactive   U
   (P_state = considered & E(P_state = considered U
    (P_state = possible   & E(P_state = possible   U
     (P_state = ready      & E(P_state = ready      U
      (P_state = activated  & E(P_state = activated  U
       P_state = completed)))))))))
```

The formulas for all other valid sequences of states are be defined analogous.

7.2 Translation Schemata from GDL to SMV-Properties

Originally, model checking of Asbru was supposed to be used mainly for checking structural properties. But it has turned out, as we have shown in deliverable D4.2c of this project, model checking is also very suitable for verification of medical properties. Medical properties are usually given in a natural language, their ad-hoc translation to SMV-propeties is a non-trivial and error prone task. To aid in the task to translate medical properties to formal properties, which can be used in verification tools, the guideline description language GDL was invented and presented in the master thesis of R. Stegers [39].

In this section, we want to describe first approaches, how GDL properties could be translated into SMV properties. As GDL formulas has many special constructs, we do not consider each of them here and limit this section only to general cases of GDL.

In this section we first give a short overview of GDL properties and their translation to KIV properties. Then we present two different translation GDL-to-SMV translation pattern. The first is a direct translation, which uses only temporal logic (TL) formulas to express the properties in SMV. The second pattern uses the GDL-to-KIV translation pattern, resulting in properties which are very similar to KIV properties. It uses TL-fomulas and special trigger variables to express the properties in SMV. The section concludes with an example translation, which shows the application of the second pattern.

7.2.1 Overview of GDL properties and the translation into KIV properties

Here, we can only give a small overview of GDL. For details of GDL, see [39]. GDL, and the translation of GDL to KIV are described there in detail.

The translation process of GDL begins with a property given in natural language and transforms this property in several successive steps into an attached GDL property. This attached GDL property is a formal representation of the original property and can be mechanically translated into the input language of verification tools like KIV or SMV.

In general, an attached GDL property has the following pattern[1]:

Goal *Goalname*
    Precondition
        preCondition
    Time-specification
        From *startEvent*

[1] In GDL, several variants of the schema presented here are allowed, which we do not discuss here.

Until *endEvent*
behaviour
eventOrCondition

In this schema, *preCondition* restricts the patient group for which the property should hold. *startEvent* and *endEvent* are used to define an interval, in which the property should hold. In *eventOrConditon* the relevant event or condition is specified. The *behaviour* specifies how *eventOrCondition* should be interpreted over the interval defined by *start-* and *endEvent*. The following keywords are allowed to specify the behaviour:

$$
\begin{aligned}
<\textit{behaviour}> := \ & \textit{'maintain-during-period'} \\
& | \ \textit{'avoid-during-period'} \\
& | \ \textit{'observe-during-period'} \\
& | \ \textit{'achieve-at-end'} \\
& | < \textit{'sub-goal'} >
\end{aligned}
$$

As example, the behavior *observe-during-period* means that *eventOrCondition* should occur at least once if *eventOrCondition* is a condition in the specified interval. If *eventOrCondition* is an event, the operand requires an addition count specificaion (e.g. observe-during-period = 1 or observe-during-period > 5). In this case the event must be observed the specified number of times in the interval.

The translation from GDL to KIV presented in [39] uses three kinds of KIV TL-formulas. The first are called *check-formulas*, which are the actual formulas for the property which have to be proven. A second kind of formula is called *test-formula*. Test-formulas are included into the check-formulas. They are usually very simple and mostly consist of a single flag that is set, if a certain signal (e.g., an event) did occur in the past. The last kind of formulas are called *support-formulas*. They are added to the antecedent of the KIV formula, that is they affect the model behavior but they have not to be proven. Support-formulas are mainly used to set the flags that are used in the test-formulas, e.g., to signal the occurrence of the *Start-* or *EndEvent* or the occurrence of *eventOrCondition*.

The actual pattern of a check-formula is dependent of the behavior of the attached GDL-property. We considere here only the most common form of check-formulas which has the following pattern:

$$\Box\ (\mathit{preCond}\ \rightarrow\ \mathit{cond})$$

Here, the formulas *preCond* and *cond* are predicate logic formulas which consists normally of one or more test-formulas.

Most support-formulas consist of an initialization of the trigger variable and an if-then-else like formula, which detects the occurence of a specific event of satisfaction of a specific condition. It has usually the following schema:

$$
\begin{aligned}
& \neg \mathit{Trigger}, \\
& \Box\ (\mathit{Event}\ \supset\ \mathit{Trigger}'' = \mathbf{True}\ ;\ \mathit{Trigger}'' = \mathit{Trigger})
\end{aligned}
$$

This KIV-formula sets *Trigger* to true as soon as *Event* occurs, otherwise it leaves *Trigger* unchanged. Initial *Trigger* is false.

7.2.2 Direct GDL-to-SMV translation pattern

One way to automate the translation of GDL properties into SMV has been devised, which uses the observation that there are a relatively low number of different behaviours are possible in GDL. The idea of the direct translation is to find a temporal formula pattern for every behaviour possible in GDL. Table 7.1 presents the TL-patterns we found.

Nr.	GDL Behaviour	SMV Property
1	MAINTAIN-DURING-PERIOD (StartEvent, EndEvent, Condition)	$AG\ (StartEvent \rightarrow A\ (Condition\ W\ EndEvent))$
2	OBSERVE-DURING-PERIOD (StartEvent, EndEvent, Event)	$AG\ (StartEvent \rightarrow A\ (\neg EndEvent\ U\ Event))$
3	AVOID-DURING-PERIOD (StartEvent, EndEvent, Condition)	$AG\ (StartEvent \rightarrow A\ (\neg Condition\ U\ EndEvent))$
4	ACHIEVE-AT-END (StartEvent, EndEvent, Event, Condition)	$AG\ (StartEvent \rightarrow A\ (\neg EndEvent\ U\ (Event \wedge Condition)))$
5	SEQUENCE/SUBGOAL (StartEvent, EndEvent, Event1, Event2)	$AG\ (StartEvent \rightarrow A\ ((\neg EndEvent\ U\ Event1) \wedge (\neg Event2\ W\ (Event1 \wedge \neg Event2))))$

Table 7.1: List of GDL to SMV direct translations

This line of research has been suggested by Dwyer et al [40]. It includes formulating equivalent formulations of the most essential temporal properties in various property specification languages. If GDL is viewed as one of these languages, and these mappings are established, medical properties, which were previously specified in other languages can be expressed in GDL.

Examples for the application of this patterns are shown in chapter 7.2 of the deliverable D4.2c of the Protocure II project [4].

7.2.3 Indirect GDL-to-SMV translation pattern using the GDL-to-KIV pattern

Another possibility for translating GDL properties to SMV is to use the translation pattern GDL-to-KIV. For this translation, we take the translation to KIV-properties as an intermediate step and translate these into SMV. This proceeding simplifies the task to ensure that the translated properties in SMV are semantically the same as the translated KIV properties.

Up to a some exceptions, e.g., if the included test-formulas consider time, check-formulas can be translated directly from KIV formulas to SMV LTL-properties. For these check-formulas that can not translated directly (e.g., if, as mentioned above, they consider time), a suitable abstraction must be found.

Assuming that *preCond* and *cond* can be translated into SMV, the check-formula of the preceding section can be translated into an equivalent SMV LTL-formula. The corresponding LTL-formula in nuSMV syntax would be:

```
LTLSPEC  G ( preCond -> cond)
```

A bit more difficult to include into the SMV model are the support-formulas. It is important that we model the behavior or the trigger varialbe in the same way as they do in KIV. A possible solution to do this, is to include support-formulas into the SMV model with the following translation schema. First, we declare a new variable for the trigger:

```
  trigger : boolean;
```

Then we can use the state transition definition of SMV to specify the possible behaviors of the trigger variable:

```
init(trigger) := false;
next(trigger) := case
   cond : true;
   1    : trigger;
esac;
```

A disadvantage of this pattern, compared with the direct translation pattern, is that, as we add some new state variables, we increase the size of the model and the complexity of model checking this model. Furthermore, the translation process is more complex, as we need the GDL-to-KIV translation as an intermediate step. However, there are also several advantages of this pattern compared to the direct GDL-to-SMV translation pattern. One advantages is that the resulting properties are semantically equivalent to the properties translated to KIV. This is useful if a property should be examined with both verification techniques. Also, due to the inclusion of the trigger variables into the model, the counterexamples for violated properties are better understandable. This is especially important, as, once we have the model and the properties, the analysis of counterexamples is the most difficult and time consuming task in verification

Goal Example 4
Precondition
always-true
Time-specification
From Transition ch1-treatment **enter** active
Until Transition masectomy-proper **enter** active
Observe-during-period
patient-information-reconstruction = completed

Figure 7.3: Attachment - Example 4 (BC Ch 1 23.11.2005) [39]

with model checking. Also, it is very hard to come up with a direct translation for difficult GDL-properties, e.g., properties containing behaviors with several sub-goals or properties that must observe a specific number of envents (e.g., *observe-during-period* > 5). Here, the indirect translation offers more flexibility.

Another possible pattern would be the usage of module checking to restrict the model with the support formulas. While this approach would make the translation of TL support formulas into the SMV more direct, modular model checking has usually a higher complexity than normal model checking.

7.2.4 Example for indirect GDL-toKIV translation

For an example of the indirect transformation technique, we use the following property from the breast cancer case study, chapter 1:

"The posibillity of breast reconstruction should be discussed with all patients prior to masectomy."

This example is used as running example in [39] ("Example 4"). In this work it is shown how this property can be attached and translated into a KIV-formula. The resulting attached GDL property is shown in figure 7.3, the translation into KIV-formulas in figure 7.4.

The support-formulas in the antecedent (that are all formulas before the "$\vdash$"-character) can be translated and included into the model according to the schema presented in the previous section. For the doubly-primed variables in the KIV-formulas, which denotes the variable value in the next temporal logic step, the *next* statement is used, which does exactly the same in SMV. First, we must add the variable declaration of the new trigger-variables we need to the SMV model:

```
--variable declaration
startEventTrigger : boolean;
endEventTrigger   : boolean;
observed          : boolean;
```

```
 (:  Initial system state :)

¬ StartEventTrigger, ¬ EndEventTrigger, ¬ Observed,

 (:  Start event detection :)

□ (    ¬ StartEventTrigger
     ∧ AS['ch1-treatment']] ≠ active(@, @, @)
     ∧ AS''['ch1-treatment']] ≠ active(@, @, @)
   ⊃ StartEventTrigger''
   ; StartEventTrigger'' = StartEventTrigger ),

 (:  End event detection :)

□ (    StartEventTrigger
     ∧ AS['masectomy-proper']] ≠ active(@, @, @)
     ∧ AS''['masectomy-proper']] ≠ active(@, @, @)
   ⊃ EndEventTrigger''
   ; EndEventTrigger'' = EndEventTrigger ),

 (:  Support part of the behavior :)

□ (    StartEventTrigger ∧ ¬ EndEventTrigger
     ∧ AS''['patient-info-reconstruction']] = completed
   ⊃ Observed''
   ; Observed'' = Observed )

 ⊢

(:  The check part of the behavior :)

□ ( ¬ EndEventTrigger ∧ EndEventTrigger'' ⇒ Observed'')
```

Figure 7.4: Translation - Example 4 (BC Ch 1 23.11.2005) [39]

The resulting SMV-code for the state transition relation of these variables, which has to be included into the SMV-model, is:

```
--start event detection
init(startEventTrigger) := false;
next(startEventTrigger) := case
  !startEventTrigger & ! ch1_treatment_state = active
         & next(ch1_treatment_state = active):   true;
  1    : startEventTrigger;
esac;

--end event detection
init(endEventTrigger) := false;
next(endEventTrigger) := case
  startEventTrigger & ! masectomy_proper_state = active
         & next(masectomy_proper_state = active):   true;
  1    : endEventTrigger;
esac;

--support part of the behavior
init(observed) := false;
next(observed) := case
  startEventTrigger & !endEventTrigger
         & masectomy_proper_state = completed:   true;
  1    : observed;
esac;
```

The *check*-formula in the succedent (the formula after the "⊢"-character) is specified as NuSMV-LTL property. Here we use the *next*-TL operator (written as "`X`" in SMV) for the doubly-primed KIV-variables. The SMV-property after translation is as follows:

```
LTLSPEC  G (( !endEventTrigger & X endEventTrigger ) -> X observed);
```

Chapter 8

Experimental Results

As the verification results of model checking guideline properties are presented in Deliverable 4.2c of the Protocure II project, we present here only general issues and experimental results about the complexity of applying model checking to Asbru plans.

8.1 Application of Symbolic CTL Model Checking to Asbru Verification

In this Section we use Cadence SMV on Asbru models. Model checking with NuSMV is very similar, as both tools share the same input syntax and many options are the same or have at least an corresponding equivalent and we did not notice an remarkable performance difference between both tools. All test were performed on a 2.2 Ghz standard PC with 2 GB RAM.

For our experiments, we chose the latest version of chapter 1 of the breastcancer case study. This chapter consists of 50 Asbru-plans and is therefore one of the biggest chapters of the case study. The whole SMV model for chapter consists of more then 6200 lines of code including the 570 structural properties, which are automatically generated and included in the model by the compiler. The computation af the initial configuration, the transition relation and the reachable states takes alltoghther approx. 133 seconds. This computation is necessary before any verification. The verification of all 570 structural properties takes approx. 20min

8.2 Application of Bounded Model Checking to Asbru Verification

In previous deliverables, we have found many anomalies in the models, suggesting that, even after a careful review by the modeller, the model of a guideline can still contain errors. However, even after the extensive testing phase, it is difficult to say when the model is correct. Bounded model checking is a technique for detection of problems in the system, but it may be possible that it tells you the system complies with a certain property, while in fact it does not. However, given that there might be anomalies in both the model or the specification of properties, the same holds for other types of verification. Moreover, there are immense amounts of properties, making it impossible anyway to check that the guideline is "correct". Hence, all verification tools should be seen as techniques for finding problems in the guideline, rather than showing that the guideline is correct. Preliminary effort shows that BMC could be an interesting technique to explore further.

Small experiments were done using bounded model checking in the context of model checking Asbru plans. Given the nature of the Asbru semantics, we can observe that each plan has at most 6 states it can be in (`inactive`, `considered`, `possible`, `ready`, `activated`, `completed`). Of course, other states can occur in case the plan is rejected or aborted, however, in that case, the plan does not complete. Hence, in case of n plans executed in sequence, the number of states is restricted by $6n$. A rough estimate, going through the models of the breast cancer guideline, shows that most plans can be reached with less than roughly 15 plan executions. Therefore, most counter-examples will be within this range, making it, at least in theory, worthwhile to apply BMC to Asbru verification of breast cancer.

As an example, we could for example investigate whether or not the patient will ever be informed. The plan that takes care of this is `patient-information` abbreviated to 'pin'. We specify that 'pin' is never activated, i.e.:

$$\mathtt{G}\,\mathtt{pin} \neq \mathtt{activated}$$

Using bounded model checking (using NuSMV in this case), we find:

```
-- no counterexample found with bound 0
-- no counterexample found with bound 1
-- no counterexample found with bound 2
(...)
-- no counterexample found with bound 12
-- no counterexample found with bound 13
-- no counterexample found with bound 14
-- specification   G (!pinp_state = activated) is false
```

i.e., the smallest counter example can be found after 15 steps in this case. Often, in verification of Asbru plans, it is also useful to provide a lower-bound. For

example, suppose we have n sequential plans and our property refers to the last plan. Clearly, we do not expect counterexamples smaller than roughly $6n$. Therefore, the iteration could start at a higher bound.

Now consider the following property:

$$\mathsf{G}(\texttt{treatment} = \texttt{activated} \rightarrow \mathsf{F}\texttt{treatment} = \texttt{possible})$$

Clearly, this is false, as after activation of the treatment, it cannot be restarted at any point. At first sight, it might seem that this cannot be proven by BMC, as we need to look arbitrarily long into the future to make sure treatment is never possible again. However, SMV counter models may contain loops, hence, it is still able to deduce that after activation, possible will never occur again in the model. Hence, we can still find a finite counter model:

```
-- no counterexample found with bound 0
-- no counterexample found with bound 1
(...)
-- no counterexample found with bound 8
-- no counterexample found with bound 9
-- specification G (treatmentp_state = activated
->  F treatmentp_state = possible)  is false
```

It remains future work to provide a more elaborate study to establish whether the anomalies found in the selected guideline properties, as provided in Deliverable 4.2c of the Protocure II project, can be found more easily using BMC rather than BDD techniques.

Chapter 9

Model Checking with Background Knowledge

Background knowledge is most intuitively expressed in LTL. Therefore, we use LTL model checking as well as bounded model checking for the verification with background knowlede. For accomplishing this, there are two options

1. A translation from LTL to the state machine, i.e., combining the guideline and additional knowledge into one automaton
2. LTL model checking

The first option was attempted using standard LTL to Büchi automatons [41, 42] on the diabetes case study (see Deliverable 4.2c), however, even for the limited amount of background knowledge that is used there, the SMV model explodes, which makes it infeasible to use for model checking. Of course, assuming PSPACE $\neq$ P, which is very likely, it is difficult to design really efficient algorithms to reduce the explosion of this translation.

Another approach for the verification of Asbru with background knowledge is the use of modular model checking (see chapter 3.6). We applied modular model checking to the breast cancer guideline using background knowledge with respect to daily medical practice. Here, we introduce the syntax used in SMV for the specification of modular proofs based on the SMV language definitions[1].

Assertions about the execution of the program described by the SMV model can be written down as:

```
id: assert f;
```

[1] http://www.cs.indiana.edu/classes/p515/readings/smv/CadenceSMV-docs/smv/language/language.html

where f is a linear temporal formula. If a trace in the SMV model does not satisfy f, then this is called a failure. In this case, the assertion has an identifier "id", which may be used to refer to this assertion.

Modular verification of $[\varphi]M\langle\psi\rangle$ is written down as follows:

```
using id prove SPEC g;
```

where id is an assertion. It is also possible to verify assertions using other assertions, e.g.,

```
using id prove id2;
```

to verify that assertion identified by "id2" holds under the assumption that assertion "id" holds. An example of this technique used to verify the guideline using background knowledge can be found in Deliverable 4.2c.

Chapter 10

Summary

Our experiences with model checking showed how helpful this technique is for all kinds of analysis of formal models. In the first phase of using model checking for the verification of Asbru we mainly concentrated on the verification of structural properties of Asbru models. Already at this stage we achieved very promising results, enhanced our techniques and gained lots of experiences from verification process. The next phase started with application of the already validated technique to another case study called "breast cancer". Although it was not our goal in the beginning we have also shown that model checking can also be successfully applied for the verification of medical properties as well as for other kinds of analysis of Asbru models, e.g. critiquing etc.

Our choice in the beginning for a "push-down" approach, i.e. automatic model generation using standard abstraction, turned out to be a very flexible and efficient solution. The tool we developed for the compilation of SMV models from Asbru models in KIV made the whole process much more efficient. Otherwise, it were even impossible to construct some bigger SMV models with more than 5000 lines of code without some automatization support.

The general idea behind model checking to find as many errors in the design as possible in the short period of time proved itself to be very useful not only in the Asbru verification but also surprisingly in the modeling stage of the overall framework. If we just review the metaphor of our project "Asbru model - program" we recognize that constructing Asbru models is somehow like writing programs (yet parallel programs which is even more challenging). Of course by using the Asbru interpreter tool the modeler can detect some errors. But testing can not be compared with the state of the art verification technique allowing to check the most forgotten cases. So we propose for the future to integrate such quick check in the modeling process as well as a pre-check for new models before starting an interactive verification.

The approach we presented here uses the standard abstraction for the construction of a finite state model without considering any model of time. Using this

model we are able to verify a broad class of properties. Nevertheless, we are not able to find errors concerning timing issues in time annotations. Therefore, an interesting direction for the future work would be modeling of time annotations using one of the numerous automatic techniques and tools for the verification of real time systems. Another promising direction would be the better integration of both verification techniques, i.e. automatic and interactive. Currently we use them practically in parallel. The integration of theorem proving in the model checking process in order to tackle the state explosion problem or vice versa, using model checking for more efficient proof construction are interesting alternatives to our approach which can be investigated in the future.

Appendix A

Compiler

```
MODULE main
-- chapter 3 : plan name abbreviations for
--load  plan-locally-advancedBC       locally-advanced-bc

DEFINE
--locally-advanced-bc plan
loadp_is_terminated := (loadp_state = rejected) | (loadp_state = aborted) |
                                                  (loadp_state = completed);
loadp_is_aborted := (loadp_state = rejected) | (loadp_state = aborted);
loadp_consider_condition := 1;
loadp_filter_condition := diagnosis = locally-advanced-BC;
loadp_setup_condition := 1;
loadp_activate_condition := (1);
loadp_parentterm_condition := 0;
loadp_reject_condition := loadp_parentterm_condition | (loadp_state = considered &
                                                        !loadp_filter_condition);
loadp_abort_multi_condition := loadp_abort_condition | loadp_parentterm_condition |
                                                       loadp_child_is_aborted;
loadp_abort_condition := 0;
loadp_complete_multi_condition := (loadp_complete_condition & loadp_waitfor_condition) |
                                                              (!loadp_ifthen_condition);
loadp_complete_condition := 1;
loadp_waitfor_condition := (intmp_state=completed) | (lintmp_state=completed);
loadp_is_in_selection_phase := (loadp_state = considered) | (loadp_state = possible) |
                                                              (loadp_state = ready);
loadp_retry_condition := 0;
loadp_child_is_aborted := (intmp_is_aborted) & (lintmp_is_aborted);
loadp_ifthen_condition := 1;


--variable tick=1 indicates a macro step
tick := !(
    (loadp_state = inactive & loadp_consider_condition)|
    (loadp_is_in_selection_phase & loadp_reject_condition)|
    (loadp_state = considered & loadp_filter_condition)|
    (loadp_state = possible & loadp_setup_condition)|
    (loadp_state = ready & loadp_activate_condition)|
```

```
    (loadp_state = activated & loadp_complete_multi_condition)|
    (loadp_state = activated & loadp_abort_multi_condition)|
    (loadp_state = aborted & loadp_retry_condition)|
    ...
    );


VAR
loadplan : load_plan(loadp_state,loadp_parentterm_condition,loadp_consider_condition,
                     loadp_filter_condition,loadp_is_terminated,loadp_is_in_selection_phase,
                     loadp_setup_condition,loadp_reject_condition,loadp_activate_condition,
                     loadp_abort_multi_condition,loadp_complete_multi_condition,loadp_retry_condition);
loadp_state : inactive,considered,possible,ready,rejected,activated,completed,aborted;

loadp_plan_control : load_plan_control(loadp_state,loadp_control_consider_signal,
                                       loadp_control_activate_signal,intmp_state,
                                       intmp_retry_condition,lintmp_state,lintmp_retry_condition);
loadp_control_consider_signal : none,intm,lintm,all;
loadp_control_activate_signal : none,intm,lintm,all;

-----------------------------------------------------------
-------------------- PATENT DATA-------------------------
-----------------------------------------------------------
diagnosis : metastasised-BC,locoregional-recurrent-BC,locally-advanced-BC,operable-BC,
             operable-invasive-BC,DCIS,unknown;


ASSIGN

 init(diagnosis) := unknown;
 next(diagnosis) := case
   diagnosis=unknown : metastasised-BC,locoregional-recurrent-BC,locally-advanced-BC,
                       operable-BC,operable-invasive-BC,DCIS;
   1 : diagnosis;
 esac;

--property TERMINATION
SPEC AG AF (loadp_state=completed | loadp_state=aborted |
            loadp_state=inactive  | loadp_state=rejected)

--property ASBRU-CONDITION

SPEC EF (loadp_filter_condition   & loadp_state=considered)
SPEC EF (loadp_setup_condition    & loadp_state=possible)
SPEC EF (loadp_complete_condition & loadp_state=activated)
SPEC EF (loadp_abort_condition    & loadp_state=activated)

--property TRACES
--trace 1: inactive -> considered -> possible -> ready -> activated -> completed
SPEC EF(loadp_state = inactive &
        E(loadp_state = inactive   U (loadp_state = considered &
        E(loadp_state = considered U (loadp_state = possible &
        E(loadp_state = possible   U (loadp_state = ready &
        E(loadp_state = ready      U (loadp_state = activated &
        E(loadp_state = activated  U loadp_state = completed))))))))))

--trace 2: inactive -> considered -> possible -> ready -> activated -> aborted
```

```
SPEC EF(loadp_state = inactive &
        E(loadp_state = inactive   U (loadp_state = considered &
        E(loadp_state = considered U (loadp_state = possible &
        E(loadp_state = possible   U (loadp_state = ready &
        E(loadp_state = ready      U (loadp_state = activated &
        E(loadp_state = activated  U loadp_state = aborted))))))))))

--trace 3: inactive -> considered -> rejected
SPEC EF(loadp_state = inactive &
        E(loadp_state = inactive U (loadp_state = considered &
        E(loadp_state = considered U (loadp_state = rejected)))))

--trace 4: inactive -> considered -> possible -> rejected
SPEC EF(loadp_state = inactive &
        E(loadp_state = inactive U (loadp_state = considered &
        E(loadp_state = considered U (loadp_state = possible &
        E(loadp_state = possible U (loadp_state = rejected)))))))

--trace 5: inactive -> considered -> possible -> ready -> rejected
SPEC EF(loadp_state = inactive &
        E(loadp_state = inactive   U (loadp_state = considered &
        E(loadp_state = considered U (loadp_state = possible &
        E(loadp_state = possible   U (loadp_state = ready &
        E(loadp_state = ready      U loadp_state = rejected)))))))))

--property COMPLETED
SPEC EF loadp_state = completed

--property ACTIVATED
SPEC EF loadp_state = activated

--property WAIT-STATES
SPEC AG(loadp_state = possible  -> AF(loadp_setup_condition       | loadp_is_terminated))
SPEC AG(loadp_state = ready     -> AF(loadp_activate_condition    | loadp_is_terminated))
SPEC AG(loadp_state = activated -> AF(loadp_abort_multi_condition |
                                      loadp_complete_multi_condition))

FAIRNESS loadp_state = inactive


MODULE load_plan(loadp_state,loadp_parentterm_condition,loadp_consider_condition,
                 loadp_filter_condition,loadp_is_terminated,loadp_is_in_selection_phase,
                 loadp_setup_condition,loadp_reject_condition,loadp_activate_condition,
                 loadp_abort_multi_condition,loadp_complete_multi_condition,loadp_retry_condition)
ASSIGN
 init(loadp_state) := inactive;
 next(loadp_state) := case
   loadp_parentterm_condition : inactive;
   loadp_is_terminated & loadp_retry_condition : considered;
   loadp_state = inactive & loadp_consider_condition : considered;
   loadp_is_in_selection_phase & loadp_reject_condition : rejected;
   loadp_state = considered & loadp_filter_condition : possible;
   loadp_state = possible & loadp_setup_condition : ready;
   loadp_state = ready & loadp_activate_condition : activated;
   loadp_state = activated & loadp_complete_multi_condition : completed;
   loadp_state = activated & loadp_abort_multi_condition : aborted;
   1 : loadp_state;
```

```
 esac;

MODULE load_plan_control(loadp_state,loadp_control_consider_signal,loadp_control_activate_signal,
                         intmp_state,intmp_retry_condition,lintmp_state,lintmp_retry_condition)
ASSIGN
 init(loadp_control_consider_signal) := none;
 next(loadp_control_consider_signal) := case
   loadp_state = activated : all;
   !loadp_state = activated : none;
 esac;

 init(loadp_control_activate_signal) := none;
 next(loadp_control_activate_signal) := case
   loadp_state = activated :  case

     loadp_control_activate_signal = none : case
       intmp_state = ready & lintmp_state = ready : intm,lintm;
       !intmp_state = ready & lintmp_state = ready : lintm;
       intmp_state = ready & !lintmp_state = ready : intm;
       !intmp_state = ready & !lintmp_state = ready : none;
       1 : loadp_control_activate_signal;
     esac;

     loadp_control_activate_signal = intm & (intmp_state = aborted | intmp_state = completed) :
     case
       lintmp_state = ready : lintm;
       !lintmp_state = ready : none;
       1 : loadp_control_activate_signal;
     esac;

      loadp_control_activate_signal = lintm & (lintmp_state = aborted | lintmp_state = completed) :
     case
       intmp_state = ready : intm;
       !intmp_state = ready : none;
       1 : loadp_control_activate_signal;
     esac;

     1 : loadp_control_activate_signal;
   esac;
   !loadp_state = activated : none;
 esac;
```

Bibliography

[1] Clarke, E.M., Emerson, E., Sistla, A.: Automatic verification of finite-state concurrent systems using temporal logic specifications. ACM Transactions on Programming Languages and Systems (TOPLAS) **8**(2) (1986) 244–263

[2] Dunets, A.: Verification of medical guidelines by model checking. Master's thesis, University of Augsburg, Germany (2005) In German.

[3] Bäumler, S., Balser, M., Dunets, A., Reif, W., Schmitt, J.: Verification of medical guidelines by model checking - a case study. In: SPIN. (2006) 219–233

[4] Deliverable D4.2c: Formal Verification of Selected Guideline Properties, Protocure Project-II, IST-FP6-508794 (2006) `http://www.protocure.org/`.

[5] Marcos, M., Balser, M., ten Teije, A., van Harmelen, F.: From informal knowledge to formal logic: A realistic case study in medical protocols. In: Proceedings of EKAW, Springer (2002) 49–64

[6] Ben-Ari, M., Manna, Z., Pnueli, A.: The temporal logic of branching time. Acta Informatica **20** (1983)

[7] Clarke, E.M., Emerson, E.A.: Design and synthesis of synchronization skeletons using branching time temporal logic. In: Logic of Programs: Workshop. Number 131 in LNCS, Yorktown Heights, NY, Springer (1981)

[8] Emerson, E.A., Clarke, E.M.: Characterizing correctness properties of parallel programs using fixpoints. In: Automata, Languages and Programming. Number 85 in LNCS, Springer (1980) 169–181

[9] Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. MIT Press (2000)

[10] Lichtenstein, O., Pnueli, A.: Checking that finite state concurrent programs satisfy their linear specification. In: POPL '85: Proceedings of the 12th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, New York, NY, USA, ACM Press (1985) 97–107

[11] Vardi, M.: On the complexity of modular model checking. In: Logic in Computer Science. (1995) 101–111

[12] Vardi, M.: Branching vs. linear time: Final showdown. Lecture Notes in Computer Science **2031** (2001) 1–22

[13] Alur, R., Henzinger, T., Kupferman, O.: Alternating-time temporal logic. Lecture Notes in Computer Science **1536** (1998) 23–60

[14] van der Hoek, W., Wooldridge, M.: Tractable multiagent planning for epistemic goals. In: Proceedings of the First International Conference on Autonomous Agents and Multiagent Systems (AAMAS-02). (2002)

[15] Büchi, J.: On a decision method in restricted second order arithmetic. In: Proceedings of the International Congress on Logic, Methodology and Philosophy of Science, Stanford University Press (1960) 1–11

[16] Vardi, M.Y., Wolper, P.: Reasoning about infinite computations. Inf. Comput. **115**(1) (1994) 1–37

[17] Holzmann, G.J.: The model checker spin. IEEE Trans. on Software Engineering **23**(5) (1997) 279–295

[18] Burch, J.R., Clarke, E.M., McMillan, K.L., Dill, D.L., Hwang, L.J.: Symbolic model checking: 10^20 states and beyond. Inf. Comput. **98**(2) (1992) 142–170

[19] McMillan, K.L.: Symbolic Model Checking. Kluwer Academic Publishers (1993)

[20] Bryant, R.E.: Graph-based algorithms for boolean function manipulation. IEEE Transactions on Computers **35**(8) (1986) 677–691

[21] Cimatti, A., Clarke, E., Giunchiglia, F., Roveri, M.: Nusmv: a reimplementation of smv. Technical Report 9801-06, IRST, Trento, Italy. (1998)

[22] McMillan, K.L.: The smv system. Online Manual (2000) http://www.cs.cmu.edu/ modelcheck/smv/smvmanual.ps.

[23] de Moura, L., Owre, S., Rueß, H., Rushby, J., Shankar, N., Sorea, M., Tiwari, A.: SAL 2. In Alur, R., Peled, D., eds.: Computer-Aided Verification, CAV 2004. Volume 3114 of Lecture Notes in Computer Science., Boston, MA, Springer-Verlag (2004) 496–500

[24] Amnell, T., Behrmann, G., Bengtsson, J., D'Argenio, P.R., David, A., Fehnker, A., Hune, T., Jeannet, B., Larsen, K., Möller, M.O., Pettersson, P., Weise, C., Yi, W.: Uppaal - Now, Next, and Future. In Cassez, F., Jard, C., Rozoy, B., Ryan, M., eds.: Modelling and Verification of Parallel Processes. Number 2067 in Lecture Notes in Computer Science Tutorial, Springer–Verlag (2001) 100–125

[25] Henzinger, T.A., Ho, P.H., Wong-Toi, H.: HYTECH: A model checker for hybrid systems. International Journal on Software Tools for Technology Transfer **1**(1–2) (1997) 110–122

[26] Biere, A., Cimatti, A., Clarke, E., Strichman, O., Zhu, Y.: Bounded Model Checking. Volume 58 of Advances in Computers. Academic Press (2003)

[27] Biere, A., Cimatti, A., Clarke, E., Fujita, M., Zhu, Y.: Symbolic model checking using sat procedures instead of bdds. In: Design Automation Conference (DAC'99). (1999)

[28] Kupferman, O., Vardi, M.: Modular model checking. Lecture Notes in Computer Science **1536** (1998) 381–401

[29] Grumberg, O.: Abstractions and reductions in model checking. In: Marktoberdorf summer school, Nato Science Series (2001)

[30] S. Graf, H. Saidi: Construction of abstract state graphs with PVS. In Grumberg, O., ed.: Proc. 9th INternational Conference on Computer Aided Verification (CAV'97). Volume 1254., Springer Verlag (1997) 72–83

[31] Clarke, E.M., Grumberg, O., Long, D.E.: Model checking and abstraction. ACM Transactions on Programming Languages and Systems **16**(5) (1994) 1512–1542

[32] Godefroid, P., Huth, M., Jagadeesan, R.: Abstraction-based model checking using modal transition systems. Lecture Notes in Computer Science **2154** (2001) 426+

[33] Dams, D., Gerth, R., Grumberg, O.: Abstract interpretation of reactive systems. ACM Trans. Program. Lang. Syst. **19**(2) (1997) 253–291

[34] M. Balser, C. Duelli, W.R.: Formal semantics of asbru - v2.12. Technical report, University of Augbsurg, Computer Science Department (2006)

[35] Balser, M., Duelli, C., Reif, W., Schellhorn, G.: Verifying concurrent systems with symbolic execution. Journal of Logic and Computation **12**(4) (2002) 549–560

[36] Emerson, E.A.: Temporal and modal logic. In van Leeuwen, J., ed.: Handbook of Theoretical Computer Science: Volume B: Formal Models and Semantics. Elsevier, Amsterdam (1990) 995–1072

[37] Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: Computer Aided Verification. (2000) 154–169

[38] Duftschmid, G., Miksch, S.: Knowledge-based verification of clinical guidelines by detection of anomalies. OEGAI Journal **18**(2) (1999) 37–39

[39] Stegers, R.: From natural language to formal proof goal: Structured goal formalisation applied to medical guidelines. Master's thesis, Free University Amsterdam, department of artificial intelligence (2006) Available from: `http://www.stegers.info/Ruud/MastersThesis.pdf`.

[40] Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Patterns in property specifications for finite-state verification. In: ICSE '99: Proceedings of the 21st international conference on Software engineering, Los Alamitos, CA, USA, IEEE Computer Society Press (1999) 411–420

[41] Vardi, M.: An automata-theoretic approach to linear temporal logic. In Moller, F., Birtwistle, G., eds.: Logics for Concurrency, Springer Verlag (1996) 238–266

[42] Gastin, P., Oddoux, D.: Fast ltl to büchi automata translation. In: CAV '01: Proceedings of the 13th International Conference on Computer Aided Verification, London, UK, Springer-Verlag (2001) 53–65