Process-based Derivation of Requirements for Medical Devices

Heather M. Conboy
Dept. of Computer Science
University of Massachusetts
Amherst, MA 01003 USA
hconboy@cs.umass.edu

George S. Avrunin
Dept. of Computer Science
University of Massachusetts
Amherst, MA 01003 USA
avrunin@cs.umass.edu

Lori A. Clarke
Dept. of Computer Science
University of Massachusetts
Amherst, MA 01003 USA
clarke@cs.umass.edu

ABSTRACT

One goal of medical device certification is to show that a given medical device satisfies its requirements. The requirements that should be met by a device, however, depend on the medical processes in which the device is to be used. Such processes may be complex and, thus, critical requirements may be specified inaccurately or incompletely, or even missed altogether. We are investigating a requirement derivation approach that takes as input a model of the way the device is used in a particular medical process and a requirement that should be satisfied by that process. This approach tries to produce a derived requirement for the medical device that is sufficient to prevent any violations of the process requirement. Our approach combines a method for generating assumptions for assume-guarantee reasoning with one for interface synthesis to automate the derivation of the medical device requirements. The proposed approach performs the requirement derivation iteratively by employing a model checker and a learning algorithm. We implemented this approach and evaluated it by applying it to two small case studies. Our experiences showed that the proposed approach could be successfully applied to abstract models of portions of real-world medical processes and that the derived requirements of the medical devices appeared useful and understandable.

Categories and Subject Descriptors

D.2.1 [**Software Engineering**]: Requirements/Specifications; D.2.4 [**Software Engineering**]: Software/Program Verification – model checking

General Terms

Design, Verification

Keywords

Requirement specifications, medical devices, medical processes, model checking, learning algorithms

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
IHI'10, November 11–12, 2010, Arlington, Virginia, USA.
Copyright 2010 ACM 978-1-4503-0030-8/10/11 ...$10.00.

1. INTRODUCTION

Medical devices need to be certified to show that they satisfy their specified requirements. But, often these requirements are determined with respect to a particular plan of use, referred to here as a medical process, without taking into account the various alternative ways in which the device may be employed. Even when these alternatives are taken into account, it may be challenging to determine the appropriate device requirements since the medical process may be large and complex, especially when exceptional conditions and concurrent activities are considered. Hence, it may be difficult to reason about all potential behaviors of the medical process and their interactions with the medical device. For clarity, we use the term *overall process* to refer to the combination of a device and a medical process in which that device is used. This paper describes an approach for automatically deriving the requirements for a device given an overall process model, composed of a model of how that device will be used in a particular medical process and a simple model of the device's behaviors, along with the requirements for that overall process.

As an example, consider the combination of an infusion pump and a medical process for an in-patient surgery where the infusion pump is used in that medical process to administer intravenous fluids and medications. These pumps are used over a wide range of dosages and rates, from a milliliter or two per hour to many liters per hour, and may have several channels infusing different medications. Errors in setting a pump can lead to the administration of 1000 times the intended dose of medication in a short period. In response to this risk, manufacturers have introduced a new generation of "smart" pumps. A smart pump would be programmed with a library giving the usual concentrations, dosing units, and dosing limits for the drugs in use in a particular area of the hospital, such as an operating room or an intensive care unit. The allowed drugs and dosing limits differ for different areas; for instance, the drug library for an operating room typically allows a wider range of dosing limits than that for an intensive care unit. The pumps may also include information about drug interactions and provide patient monitoring functions. The clinician using the pump selects the drug, concentration, etc. and the pump alerts the clinician if the dose exceeds the limits in the library, the drug is already being administered on another channel, or some other hazardous condition is identified.

One important requirement for any overall process in which an infusion pump is used is that a patient never be administered a drug overdose. This might be reflected as an overall

process requirement that states that if the selected dosage is outside the range, the pump must issue an alert. If the pump developers do not consider alternative usages where a pump can be moved from one area in the hospital to another, then the device requirement that the pump must be reconfigured when it is moved might be overlooked. Or, perhaps more likely, even after carefully considering such alternative usages, the device requirement might contain some subtle errors.

We are investigating a requirement derivation approach that takes a model of the overall process and a requirement that process is intended to satisfy. The proposed approach considers all potential behaviors allowed by the overall process model and outputs a derived requirement for the medical device that is sufficient to prevent any violation of the overall process requirement. This approach could identify inadequacies in existing device requirements or device requirements that have been missed.

Our approach builds on requirement derivation approaches developed for software engineering. Specifically to automate the derivation of the medical device requirements, we combine two previous approaches, an assumption generation algorithm and an interface synthesis algorithm, that both make use of model checking and learning algorithms. Model checking techniques typically take as input a system model and a requirement of that system and verify whether or not all potential executions of the system model satisfy that requirement. If not, a counterexample execution is provided that demonstrates how the system could violate its requirement. The proposed approach iteratively uses model checking to determine if the behaviors of the overall process that satisfy the current derived requirement satisfy the overall process requirement. If not, the learning algorithm refines the device requirement based on the generated counterexample. Although the approach is described from the perspective of deriving requirements for the device, this approach is more general in that it actually derives requirements about the interaction between the device and the medical process and, thus, could provide insights about the requirements for the device, for the medical process, or both. This paper describes this approach, the toolset developed to support this approach, and the results of a preliminary evaluation using two small case studies.

The remainder of this paper is organized as follows. Section 2 provides an overview of previous work on software-based requirement derivation approaches. The proposed approach is discussed in Section 3, and the toolset implementing it is described in Section 4. Section 5 summarizes our evaluation, and Section 6 discusses our contributions and some possible directions for future work.

2. RELATED WORK

In the medical domain, special purpose languages and notations have been developed to model medical guidelines and protocols. Peleg et. al [25] provide a recent survey of guideline and protocol models. These medical process models are usually expressive enough to easily capture the normal behaviors but can be cumbersome when expressing aspects such as exceptional situations and healthcare professionals communicating with each other and various software applications and devices. Moreover, many of these medical process models do not have precisely defined semantics so they can not be formally analyzed. Our approach requires medical process models that are both expressive and precise.

For hardware and software systems, there has been extensive previous work on model checking techniques, e.g., [11, 23]. Model checking techniques, however, suffer from the state explosion problem, where the size of the system model or the cost of the verification algorithm may grow exponentially with the size of the system, so model checkers incorporate a number of optimizations to ameliorate this state explosion. Model checking tools differ with regard to the modeling language, the requirement specification language, the verification algorithm, and the supported optimizations. Compositional verification approaches try another way to reduce the impact of the state explosion problem, by using a divide and conquer strategy to decompose the verification of the entire system into the individual verification of each component of that system. A system component, however, often satisfies a requirement only in certain environments. Assume-guarantee reasoning techniques, e.g., [26], have been developed to utilize an assumption about the environments in which a system component is used. But appropriate assumptions are often difficult to provide.

As stated in the introduction, our approach builds on assumption generation methods, e.g., [2, 8, 12], developed to support assume-guarantee reasoning techniques. The assumption generation methods mentioned here employ a learning algorithm and a model checker to learn the assumptions. Like these approaches, ours also considers a decomposition of the overall system model. We separate the overall process model into the medical process model and the device model, and then the assumption learned is a derived requirement of the device. Since the assumption generation methods are performing verification, these methods often stop learning the assumption after encountering the first execution of the system model that violates the given requirement of that system. In our case, this means that the learned assumptions would be too strong to be useful, and thus we extended these approaches.

For that extension, our approach builds on interface synthesis methods, e.g., [3, 6, 19]. These methods take as input a software component and a requirement of that component and output an interface that captures the most general way to use that component without violating the given requirement; conceptually, these methods are performing assumption generation where the system component is known but the particular environment in which that component is used is unknown. Unlike the assumption generation methods, the interface synthesis algorithms do not stop when the first violation is found and thus the interfaces should be weak enough to be useful requirements. While some interface synthesis methods use a combination of learning and model checking, as in the assumption generation techniques, other interface synthesis methods have been developed that are based on game theory or counterexample guided abstraction refinement, e.g., [6, 20]. In particular, the interface synthesis techniques employ different strategies to weaken the generated interfaces. Our approach uses the same strategy as the interface synthesis approach developed by Giannakopoulou and Păsăreanu [19], but differs with regard to the modeling language and the model checker employed.

Interactive requirement elaboration techniques, e.g., [1, 24], input a set of low-level requirements for a system, where those requirements are represented as conditional actions,

and then iteratively refine those requirements to meet a high-level system requirement. On each iteration, the user must provide positive and negative scenarios of the system's behavior and then must select from among refinements suggested based on those scenarios. The suggested refinements are computed by employing a learning algorithm. These techniques and our proposed requirement derivation approach share the goal of deriving requirements that help to ensure the overall system meets its requirement. But our approach focuses on the interactions between the device and the medical process and does not need to be provided with additional information from the user on each iteration.

Previous work has analyzed medical process models by applying theorem proving, e.g., [28], and model checking, e.g., [9, 10, 15]. With the model checkers, if a medical process model may violate its requirement then a counterexample is generated that demonstrates that violating behavior. The counterexample can be used to modify the behavior of the medical process. In our work, if an overall process model, composed of a model of a given device and a model of a particular medical process that uses that device, may violate its requirement then a derived device requirement is produced that restricts the interactions of the medical process and the device to ensure that the overall process requirement is satisfied. The derived requirement can be used to modify the behavior of the medical process, the device, or both.

3. REQUIREMENT DERIVATION APPROACH

Assume that for a given overall process, we know the overall process requirements and understand the medical process that defines how each device should be used. What are the requirements on the devices that will assure that the overall process satisfies its requirements? To simplify the problem, we assume that all human agents in the medical process (e.g., doctors, nurses, technicians, etc.) perform their activities correctly and that there is a single device and a single overall process requirement. The process-based requirement derivation approach we present here takes as inputs an overall process model and one of its requirements. In more detail, the overall process model is composed of a formal model of the medical process in which the device is represented only by the interface it presents to the rest of the medical process and a simple device model that, in the most permissive case, essentially allows the device to behave in an arbitrary fashion. For the infusion pump example, for instance, the medical process model would describe the activities of the medical personnel and include a representation of the way in which they set the pump and receive alerts from it. But, the pump device model could abstract away any details about the internal logic by which the pump determines when to issue alerts, etc.

At a high-level, the requirement derivation algorithm first determines if a derived requirement for the device is needed to assure that the overall process model satisfies its requirement. If so, it creates a very permissive initial version of that requirement, i.e., one that allows essentially arbitrary behavior of the pump. This algorithm then tries to find a stronger requirement that ensures that the overall process requirement will be satisfied. At the same time, it tries not to overrestrict the behavior of the device. This is because the derived requirement of the medical device must be restrictive enough to prevent violations of the overall process requirement but permissive enough to be useful as a device requirement. For instance, a derived requirement of the infusion pump that prevents the pump from administering any medication ensures that an overdose is not administered, since the pump can administer no doses at all. But that derived requirement is too restrictive to be useful since it also prevents the administration of necessary medications at the correct dosage.

The iterative improvement is done by first attempting to disallow behaviors of the overall process model that violate the given requirement of that process. This is called "strengthening" the derived requirement. For the pump example, for instance, a behavior in which a dose that exceeds the library limits is entered and the pump does not issue an alert would violate the overall process requirement and should be disallowed by the derived requirement. A model checker is employed to query the overall process model for counterexamples that cause a violation of the overall process requirement. A learning algorithm, called the learner, is then employed to refine the derived requirement based on those counterexamples. Second, the algorithm attempts to allow as many behaviors of the overall process model as possible that do not violate the overall process requirement by modifying the device requirement to allow those behaviors. This is called "weakening" the derived requirement. For instance, a behavior in which a dose that is within the library limits is entered and the pump does not issue an alert should be allowed.

Figure 1 shows the requirement derivation algorithm as a flowchart. This algorithm has three main steps. At a high-level, the model checker is responsible for Step 1 and the Step 3 queries while the learner is responsible for Step 2 and the Step 3 refinements. In the following, each step is described in more detail.

Step 1 does an initial screening to determine if a device requirement should be derived. This step first employs the model checker to check if all of the potential executions of the overall process model satisfy its requirement. If that check succeeds, then this algorithm reports that "Overall process model satisfies its requirement; so a derived requirement is not needed." On the other hand, if that check fails for all of the potential executions of the overall process model, then restricting the behavior of the device cannot produce executions of the overall process model satisfying the requirement of that process. In this case, the algorithm reports that the "Overall process model violates its requirement; no derived requirement could lead to the overall process model satisfying its requirement." If the check succeeds for some of the potential executions of the overall process model, then this algorithm proceeds to Step 2 to propose an initial derived requirement and then to Step 3 to refine it.

Step 2 employs the learner to create an initial derived requirement. We currently create a very simple, permissive initial requirement that is described in more detail in the toolset section.

Step 3 iteratively performs the requirement derivation where each iteration has two phases. The goal of Phase 1 is to make the derived requirement strong enough so that if the device satisfies the derived requirement then the overall process model will satisfy its requirement. Phase 1 first performs Query 1, which employs the model checker to search for a counterexample where the overall process requirement is vi-

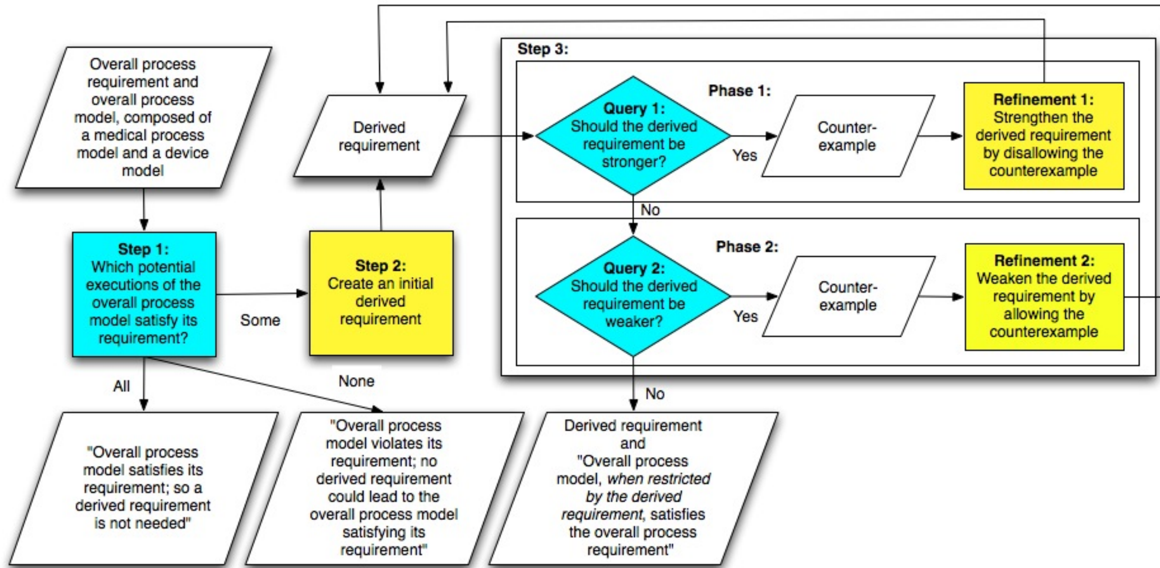

Figure 1: Flowchart for the requirement derivation algorithm

olated but the derived requirement is satisfied. Next, Refinement 1 checks whether such a counterexample is found. If so, the learner strengthens the derived requirement by disallowing the behavior corresponding to that counterexample. The algorithm then returns to the start of Step 3. For Refinement 1, if such a counterexample is not found then the algorithm proceeds to Phase 2.

The goal of Phase 2 is to weaken the derived requirement enough to make it a useful requirement. Phase 2 first performs Query 2, which employs the model checker to search for a counterexample where the overall process requirement is satisfied but the derived requirement is violated. Refinement 2 then checks whether such a counterexample is found. If so, the learner weakens the derived requirement by allowing the behavior corresponding to that counterexample. Since the derived requirement may now be too weak, this algorithm then repeats Step 3. If no such counterexample is found, the algorithm outputs the derived requirement and reports that the "Overall process model, *when restricted by the derived requirement*, satisfies the overall process requirement."

We note that the requirement derivation algorithm depends on the initial model for the device. Rather than use a permissive initial model of the device that allows it to behave in an essentially arbitrary fashion, one could start with a more concrete model that imposes reasonable restrictions on the device's behavior. This could be based on an existing device, or simply on plausible restrictions such as requiring the device to be turned on before it produces any output. While using more restrictive device models may decrease the derivation time and produce more understandable derived requirements (because they need to restrict less behavior), it may also hide issues that would only be exposed when considering more general device behavior.

The key difference between our approach and the assumption generation methods used for assume-guarantee compositional verification is in Phase 2. Once a derived requirement that is strong enough to ensure satisfaction of the overall process requirement has been found, the assume-guarantee reasoning approaches check whether the device model satisfies the derived requirement. Since the initial model for the medical device may be imprecise, we expect this check to fail, but if the counterexample showing how the device model could violate the derived requirement leads to a violation of the overall process requirement, the assume-guarantee approaches will simply report that the overall process model can violate the overall process requirement and stop. Since those approaches are simply trying to determine whether the composed system satisfies the given overall requirement, this is sufficient. But it may be that the derived requirement generated at this point is too strong, and that there are device behaviors that violate the derived requirement but do not lead to a violation of the overall process requirement. In this case, we need to weaken the derived requirement to allow these behaviors, and we adopted an approach from interface synthesis to accomplish this.

4. REQUIREMENT DERIVATION TOOLSET

We have constructed a prototype toolset implementing the approach described in the previous section. This toolset takes as input an overall process requirement specified as a finite state automaton (FSA) and an overall process model composed of a medical process model and a permissive device model. When a less permissive device model is desired, it also takes as input additional device requirements specified as FSAs. This toolset outputs whether or not the overall process model satisfies its requirement and, if not, then when possible it provides a derived requirement of the device, specified as an FSA, that prevents the overall process model from violating its requirement.

The requirements for a process or device describe how that process or device should behave, addressing critical aspects such as safety, security, and privacy. We use FSAs to specify

the requirements; they are expressive enough to capture a wide range of interesting requirements, they are supported by the PROPEL tool [13] we used to help elicit precise requirements from domain experts, and the learning algorithm we use learns a regular language.

The medical process model must reflect the complexity of medical processes, capturing the various responses to exceptional conditions, concurrent activities, and the communication between human agents and various software systems and devices. It must have precisely defined semantics to support formal analysis. We use the Little-JIL process definition language [7] since Little-JIL has successfully been used to model medical processes (e.g., [9, 10]), including such aspects of those processes as exceptional behavior and concurrency. Our group had previously developed a translator from Little-JIL to the input formalisms of two model checkers.

For the requirement derivation algorithm, we implemented a combination of the assumption-generation algorithm developed by Cobleigh et al. [12] and the interface synthesis algorithm developed by Beyer et el. [6]. We utilized the L^* learning algorithm [4, 27] with the FLAVERS model checker [16] for Little-JIL to answer queries.

In what follows, we first provide some background on FSAs, Little-JIL, FLAVERS, and the L^* algorithm. Then, we describe the requirement deriver tool in more detail.

4.1 Background

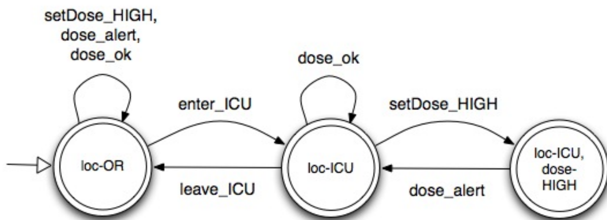

Figure 2: Pump example: Overall process requirement as an FSA

A requirement specified as an FSA captures the intended behaviors as a set of event sequences. Figure 2 shows an FSA representing the overall process requirement for the pump example described in the introduction. For brevity in the following, we shorten operating room to OR and intensive care unit to ICU. This overall process requirement says that when the pump is in an ICU and is set to deliver a dose over the allowed limit, it must issue an alert. For this requirement, the alphabet, or set of events, is: enter_ICU, leave_ICU, setDose_HIGH, dose_alert, dose_ok. There are 3 states and 6 transitions. Each state is shown as a circle. The states are labeled with the physical location of the pump in the hospital, specifically either loc-OR or loc-ICU, and may be labeled with the dose entered by a healthcare clinician, e.g., dose-HIGH. Since in the medical process model the healthcare practitioners first perform the surgery in the OR, the start state is “loc-OR” (designated by the incoming arrowhead). Additionally since the medical process model allows the pump to remain in the OR, state “loc-OR” is also an accepting state (designated by the inner concentric circle). Each transition from a state on a given event to a specified state is shown as an arc between those two states annotated with that event. To illustrate, the pump may be moved from the OR to the ICU so there is a transition from the start state “loc-OR” to state “loc-ICU” annotated with event enter_ICU. For this work, we use FSAs that are deterministic and total. To make the requirement FSAs total, we add a special violation state that is a trap state, meaning the violation state is non-accepting and for every event in the alphabet has a transition to itself. Therefore any event sequence that reaches the violation state will remain in the violation state. For simplicity, the figures that show the FSAs do not show the violation state. Thus if a state does not show a transition annotated with a particular event in the alphabet, then there is implicitly a transition on that event to the violation state. For example, state “loc-ICU, dose-HIGH” implicitly has a transition on event dose_ok to the violation state. One event sequence that satisfies this overall process requirement is enter_ICU, leave_ICU. Alternatively, an event sequence that violates this requirement is enter_ICU, setDose_HIGH, dose_ok.

The medical process model and the device model are described using the Little-JIL process definition language [7]. A Little-JIL process model precisely captures how to perform a particular task and contains three main components, a resource repository, an artifact collection, and a coordination specification. The resource repository defines which agents, either human agents or computational agents (e.g., hardware devices, software applications), perform the activities. The artifact collection defines what artifacts are consumed and/or produced by the activities. A coordination specification precisely defines how the agents perform the activities that consume and/or produce the artifacts. Since we are primarily interested in the humans’ interactions with the devices and each other, we focus here on the coordination specifications. The Little-JIL coordination specifications provide support for high-level language features such as abstraction, parallelism, synchronization, and exceptional conditions. A coordination specification has a visual representation that consists of a hierarchical decomposition of steps where each step represents an activity to be performed by a particular agent.

The FLAVERS model checker [16] verifies whether or not any potential execution of a system can violate a given property. The input to FLAVERS is a representation of the system as a collection of control flow graphs and a specification of the property as an FSA. The user can also specify additional constraints as FSAs; in that case, FLAVERS checks whether any potential execution adhering to those constraints can violate the property. FLAVERS analysis is conservative; to make the analysis tractable, the collection of control flow graphs may allow paths that do not correspond to actual executions of the system. Thus, if FLAVERS determines that no path can lead to a violation of the property, we know that no actual system execution can violate it. But when FLAVERS finds a path that leads to a violation, we do not know for certain that this path corresponds to an actual execution of the system.

The L^* algorithm learns an unknown regular language U over an alphabet Σ and returns a minimal deterministic FSA M such that the language $L(M)$ is equivalent to U. For the requirement derivation, U should correspond to a device requirement that is strong enough so that the overall process requirement is satisfied. In addition, U should be

weak enough so that the derived requirement is useful. The L^* algorithm learns by interacting with a "minimally adequate teacher," for brevity shortened here to teacher, that is essentially an oracle that is capable of answering queries about U. The teacher must be able to answer two types of queries, a membership query and an equivalence query. A membership query inputs an event sequence σ from Σ^* and outputs whether or not σ is in U. An equivalence query inputs an FSA M_i and checks whether or not $L(M_i)$ is equivalent to U. When $L(M_i)$ is equivalent to U, it outputs true. Otherwise, it outputs false and a counterexample event sequence from the symmetric difference of $L(M_i)$ and U, i.e., an event sequence that belongs to $L(M_i)$ but not U, or one that belongs to U but not $L(M_i)$.

4.2 Requirement Deriver Tool

At a high-level, the requirement deriver tool implements the requirement derivation algorithm described in the previous section by employing the L^* algorithm and a teacher that uses FLAVERS to answer the two query types. To be able to apply FLAVERS, we used the existing translation tool to construct FLAVERS input from Little-JIL. This translation supports a subset of Little-JIL and incorporates several optimizations. For the two case studies described in the next section, we needed to extend the existing translation since a larger subset of Little-JIL was needed to model the medical processes used in those case studies. Next, we briefly describe each step of the requirement derivation algorithm.

For Step 1, FLAVERS verifies whether or not the overall process model satisfies its requirement. The result of that verification provides enough information to ascertain if all, none, or some of the potential executions of the overall process model satisfy the given requirement. For Step 2, the L^* algorithm creates a very basic initial derived requirement. It considers an initial set of event sequences that contains the empty event sequence and every event sequence of length one from the alphabet. For each event sequence in that set, the L^* algorithm performs a membership query on σ that essentially checks whether or not σ is a prefix of any event sequence in regular language U that is being learned. If the answer to that membership query is true, then the initial derived requirement allows the behaviors corresponding to σ. Otherwise, the initial derived requirement disallows the behaviors corresponding to σ.

Step 3 performs the requirement derivation by employing the L^* algorithm. On each iteration i, Step 3 uses FLAVERS to check whether any execution of the overall process in which the device's behavior satisfies the current derived requirement M_i can violate the overall process property. (If constraints on the behavior of the device have been supplied by the user, these constraints are used by FLAVERS to restrict the executions considered.) If such an execution is found, the learning algorithm produces a new derived requirement M_{i+1} that excludes this execution and starts another iteration. If no such execution is found, it proceeds to Phase 2. In this phase, Query 2 checks whether any execution of the overall process in which the device's behavior violates M_i satisfies the property of the overall process. If such an execution is found, the learning algorithm produces a new derived requirement M_{i+1} that allows this behavior.

Query 1 corresponds to Oracle 1 of the assumption generation algorithm in [12]. Because the work in [12], however, was directed at compositional verification rather than requirement derivation, Oracle 2 of that paper simply checked whether one component of the system satisfies the analog of M_i. If a counterexample is found, Cobleigh et al. check whether that counterexample corresponds to a violation of the analog of the overall process requirement, in which case they report that the system under analysis violates the requirement being checked. For our purposes, it is necessary to keep refining the derived requirement in this case. Thus our Phase 2 is an adaption of the permissiveness check from [6], as is the interface synthesis method developed by Giannakopoulou and Păsăreanu [19].

5. CASE STUDIES

We applied the requirement deriver tool described in the previous section to two small case studies, one involving a smart infusion pump and one involving an implantable cardioverter-defibrillator (ICD). For each case study, we identified an overall process requirement (specified as an FSA) and constructed an overall process model (written in Little-JIL) that is a combination of a small medical process model and a simple device model. The requirement derivation toolset is implemented in Java. The experimental platform was a laptop PC with a 2.4 GHz processor and 4 GB of RAM. For each requirement derivation toolset run, we measured the space used in megabytes (MB) and the time taken in seconds. The approach, toolset, and case studies are explained in more detail in [14].

5.1 Pump Case Study

As mentioned in the introduction, the pump case study considers a device model for a pump and a medical process model for an in-patient surgery based on scenarios described in [5]. In general within the medical process model, we elaborated those steps where the healthcare practitioners interacted with the pump, based on a demonstration given by Professor Elizabeth Henneman from the University of Massachusetts School of Nursing. The overall process requirement was taken directly from the safety goals discussed in [5].

For the pump device model, we made several simplifying assumptions. There are only two drug libraries modeled, a drug library for an OR and a drug library for an ICU. Each drug library contains a single drug, and the only dosing parameters modeled are a minimum dosing limit and a maximum dosing limit. The drug doses are abstracted as either low or high. For the pump, we consider only the command *setLib* that inputs a care area and configures the pump with the drug library associated with that area and the command *setDose* that inputs a drug dose (either low or high), checks whether or not that dose is within the dosing limits for the configured drug library and if not reports a dose alert. The pump device model uses a FLAVERS constraint to capture the pump's internal behavior (which library is it configured for) and the logic of the check for potential overdoses and underdoses.

At a high-level of abstraction, a medical process for an in-patient surgery consists of five major phases: checking the patient into the hospital, performing the operation on that patient, administering ICU care if needed, monitoring the patient during recovery, and checking that patient out of the hospital. The patient is initially hooked up to the pump in the OR and then the patient and pump may be moved to

the ICU if needed. The following sequence of steps defines how to use the pump to administer an infusion in a given area in the hospital. First, a clinician has the option to set the pump to the drug library associated with that area. Next, the clinician must set the pump for the dosage to be infused by entering the appropriate number. Lastly, if the pump does not report a dose alert, then the medical clinician employs the pump to infuse the entered dosage. Otherwise if the pump does report a dose alert then the clinician decides either to restart this entire sequence, assuming the dosage was not entered correctly (i.e. mistyped a number), or to not administer the infusion at all until double checking with someone else that the entered dosage was the appropriate one. In total, the overall process model written in Little-JIL contains 76 steps; more details are provided in [14].

For the pump case study, the requirement derivation toolset was given the overall process requirement shown in Figure 2 and the overall process model described above that is a combination of the medical process model for an in-patient surgery and the device model of the pump, including the constraint that captures the pump's internal behavior. The toolset reports that the overall process model may violate its requirement and produces a derived requirement of the pump. The requirement derivation used 22 MB and took 152 seconds.

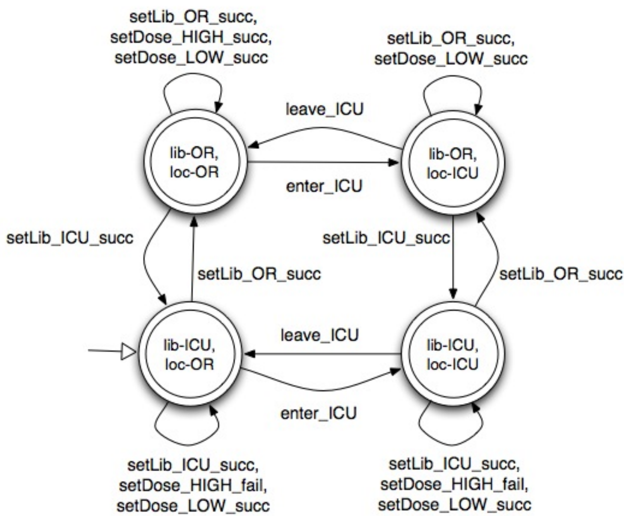

Figure 3: Pump derived requirement as an FSA

Figure 3 shows the derived requirement of the pump as an FSA. Each state is labeled with the configured drug library of the pump and the physical location of that pump. The pump is initialized with the most restrictive drug library, which in this model is the ICU's drug library. Within the in-patient surgery medical process, the pump is initially physically located in the OR as described above. Thus for the derived requirement of the pump, the start state is labeled "lib-ICU, loc-OR." Each transition is annotated with an event that corresponds to either a pump command (e.g., event setLib_OR_succ designates that the pump command *setLib* to the OR completed successfully) or moving the pump from one location to another (e.g., event enter_ICU designates when the pump moves from an OR to an ICU). Informally, the derived requirement of the pump states that after the pump is moved into the ICU that pump must be configured with the ICU's drug library before that pump is used to administer infusions in the ICU. This is illustrated by the event sequence enter_ICU, setDose_HIGH_fail and the event sequence setLib_OR_succ, enter_ICU, setLib_ICU_succ, setDose_HIGH_fail.

This derived requirement could be added to the set of requirements for a real-world pump being developed. Then to satisfy the pump derived requirement, the pump developers could modify the pump to make it location sensitive (e.g., by having the pump query a central computer in the hospital or employ radio-frequency identification tags) so the pump can ascertain the care area in which it is located and then configure itself for that area. Since the modified pump would satisfy the pump derived requirement, this would help to ensure that the overall process satisfies its requirement about a potential overdose leading to a pump alert. Alternatively the medical process model for an in-patient surgery could be modified so that after a pump is moved into the ICU the healthcare professionals always reconfigured that pump for the ICU before using it. Thus an overall process model, composed of the modified medical process model and the original pump device model, would satisfy the overall process requirement. In addition, the real-world medical process for an in-patient surgery would need to similarly be modified and the hospital administration would have to ensure that the healthcare professionals are adhering to this modified medical process.

5.2 ICD Case Study

The ICD case study considers an overall process model composed of a device model for an ICD and a medical process model that describes an ICD patient's care. The overall process requirement was taken directly from the security and privacy goals discussed in [21]. The ICD device model is based on the description provided in [22]. The ICD medical process model is based on the observations by Professor Kevin Fu and his colleagues of an implantation of a new ICD and a battery replacement for an existing ICD performed at a local area medical center [18]. The overall process requirement involves security, specifically this requirement specifies that "an outsider should not be able to trigger an ICD's test mode, which could induce heart failure" [21].

An ICD is implanted in a patient's chest cavity and is connected to that patient's heart with electrical leads. Once the ICD is implanted, an external programming device may be employed to access and program the ICD by sending radio commands from the programming device to the ICD. To simplify the ICD device model, we only modeled four commands. Initially, the ICD rejects commands from the programming device. A wand is employed to send the command to *activate* the ICD so that it accepts commands from the programming device. The wand may use a magnet or a radio signal for the activation. Additionally, there is a command to *deactivate* the ICD so that it returns to the mode in which it rejects further commands. The command *testmode* administers a shock to stop the heart and the command *readdata* reads the internal ICD settings and also telemetry about the heart. Additionally, the ICD device model employs a FLAVERS constraint to capture some of the ICD's internal behavior (whether the ICD is activated or deactivated) and the logic for whether or not to accept commands from a programming device. At a high-level, the ICD medi-

cal process model describes three alternatives usages of the ICD: an implantation performed in a hospital of an ICD in a patient, a follow up in a clinic, or an attack from outside a healthcare facility where the outsider attempts to trigger the ICD's test mode. We elaborated those steps where the humans interacted with the ICD. In total, the overall process model, composed of the simplified ICD model and the medical process model described above, contains 66 steps.

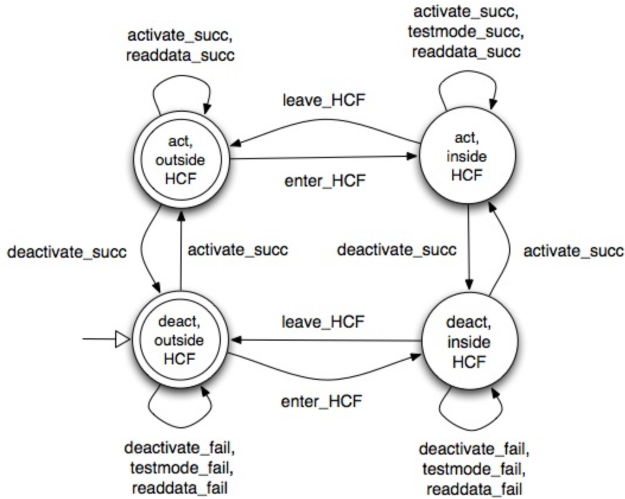

Figure 4: ICD derived requirement as an FSA

The requirement derivation toolset reports that the overall process model may violate its requirement and produced the derived requirement of the ICD shown in Figure 4. Each state is labeled with whether the ICD is activated or deactivated (either act or deact) and the physical location of that ICD (either inside or outside a healthcare facility). Initially, the ICD is deactivated as described above and is physically located outside of a healthcare facility (HCF) thus the start state is labeled "deact, outside HCF." As in the derived requirement of the pump, each transition is annotated with an event that corresponds to either an ICD command (e.g., activate_succ designates that the ICD command *activate* completed successfully) or the ICD switching physical locations (e.g., enter_HCF). The ICD derived requirement states that if the ICD is outside a healthcare facility (designated by the two states "deact, outside HCF" and "act, outside HCF") then the command *testmode* must not succeed (designated by the implicit transitions from each of those two states to the violation state on the event testmode_succ). In total, the requirement derivation toolset used 113 MB and took 260 seconds.

Additionally, the ICD derived requirement captures that if the ICD is deactivated (designated by the two states "deact, outside HCF" and "deact, inside HCF") then the command *testmode* does not succeed (designated by both those states having self-loop transitions on event testmode_fail). Therefore to satisfy the ICD derived requirement, the ICD device model or the ICD medical process model could be modified in such a way so that the ICD can never be activated outside of a healthcare facility (designated by state "act, outside HCF"). Specifically, the ICD derived requirement highlights two potential failures in the use of the ICD that could lead to the ICD being activated outside of a healthcare facility. There are the "intentional" failures due to the attacker activating the ICD (e.g., event sequence activate_succ, testmode_succ) and the "unintentional" failures due to a healthcare practitioner not deactivating the ICD before the patient leaves the healthcare facility (e.g., event sequence enter_HCF, activate_succ, leave_HCF, testmode_succ). The ICD device model could be modified so that after a certain condition is met the ICD must deactivate itself before the patient leaves the healthcare facility. For example, Halperin et al. [21] suggest that the condition could be either when the patient sits up or when the patient leaves the operating/exam room. After deriving this requirement, we learned that many ICDs deactivate themselves after a short period of time, apparently to reduce power consumption. This time is short enough (several minutes) to prevent, or at least greatly reduce, problems caused by both the unintentional and intentional failures. Alternatively, the medical process model for an ICD patient's care could be modified so that a medical professional must always deactivate a patient's ICD before that patient leaves the healthcare facility.

5.3 Discussion

This work explores a process-based requirement derivation approach that takes as input an overall process requirement and an overall process model composed of a model of the medical process and a model of the device, and, when possible, outputs a derived requirement for that device that prevents the overall process model from violating its requirement. To automate the requirement derivation, we combined a method for assumption generation with one for interface synthesis. The device requirements are iteratively derived using a learning algorithm and a model checker. This approach could be implemented for other process modeling languages, requirement specification languages, assumption generation algorithms, interface synthesis algorithms, learning algorithms, and model checkers. Our primary goal for this work was to provide "proof of concept" for such a requirement derivation approach by performing the preliminary evaluation on the two small case studies.

For this approach, we need medical process models that define how particular medical devices are used. To gain assurance about the quality of such medical process models, we could apply validation techniques such as manual reviews and formal methods (e.g., theorem proving and model checking). Although other work, e.g., [9, 10], has shown that the effort to define and validate medical process models is challenging and time consuming, this effort is worthwhile since these medical process models could also potentially be employed to train medical personnel, be the subject of other formal methods, produce simulation data, and support process guidance in the clinical setting. The medical processes in our preliminary evaluation were easily captured with Little-JIL. For each case study, the overall process model and the interactions between the medical professionals and the medical device were specified using Little-JIL's capabilities for parallelism and synchronization. Within the medical process models, we took advantage of Little-JIL's facilities for expressing abstraction and exceptional control flow.

Our preliminary evaluation applied this approach to relatively small parts of medical processes modeled at a high level of abstraction and used overall process requirements

involving both safety and security. The derived requirements for the medical devices are understandable and appear to be useful for providing insight into the interactions between the medical processes and medical devices. To further assist the medical device developers, the derived device requirement FSAs could be used to generate positive scenarios (i.e. sequences from the start state to an accepting state) and negative scenarios (i.e. sequences from the start state to a non-accepting state). Alternatively, a particular device model and/or medical process model could be validated against the derived device requirement by employing such techniques as testing or model checking. Any failed test results from testing or counterexamples from the model checker could then be used to localize the failure.

In general, the performance of the requirement deriver tool scaled well in terms of space and time when the overall process model had more details added but those additions did not involve interactions between the healthcare practitioners and the medical device (e.g., the in-patient surgery medical process had more details added about checking in a patient to the hospital). Since our tool combines a learning algorithm with a model checker, it benefited from the model checker's optimizations. On the other hand, this tool's performance did not scale as well when any additions involved interactions between the clinicians and the device (e.g., the medical process for an ICD patient's care had more details added to support a larger set of ICD commands). For each of the two case studies, the requirement deriver tool needed less than 150 MB of space and under 5 minutes for time. For this approach, the case studies suggest that space is more of an issue than time. In future work, the requirement deriver toolset's performance could be improved by employing additional optimizations supported by the learning algorithm, the model checker, or the translator from a process model to the input formalism of the model checker.

6. CONCLUSION AND FUTURE WORK

A medical device must be certified to gain assurance that it satisfies its requirements. But since that device may be used in alternative ways within a large and complex medical process, it may be challenging to accurately determine all of the necessary device requirements such that the overall process satisfies its requirements. Specifically, critical requirements of the medical device may be specified inaccurately or incompletely or critical requirements may be missed entirely. In this paper, our contributions are the proposed process-based requirement derivation approach, a toolset developed to support this approach, and a preliminary evaluation on two small case studies.

Based on this preliminary evaluation, we believe the approach is promising. For each case study, this approach derived a medical device requirement that was readily understandable and useful. A derived requirement for a particular medical device could be utilized to gain a better understanding of the interaction between that device and the given medical process in which that device is used. Then, the medical process model, device model, or both could be modified to satisfy the derived requirement to help ensure that the overall process model satisfies its requirement. In the future, the developers of real-world medical devices and medical processes could be provided with the derived device requirements that could illustrate when existing device requirements are either inaccurate or incomplete or when new device requirements are needed because they were missed.

For a better understanding of this approach, a more extensive evaluation is needed. Further evaluation should consider medical process models that are more detailed, a larger range of overall process requirements, and other medical processes than the two discussed here. For this work, we built on one assumption generation method and one interface synthesis method that employ a learning algorithm. Other methods, such as those that use game-theoretic approaches or counterexample-guided abstraction-refinement techniques, could also be used. In theory, such methods have the same worst case complexity. But in practice, these methods vary widely with regard to the performance (in terms of space and time) and the output derived requirements (in terms of size and complexity) and it would be interesting to compare the performance of tools based on such approaches.

As noted earlier, the derived requirements characterize the interaction between the rest of the medical process and the device and could be met by various combinations of device features and modifications to the medical process. Indeed, the approach could be used with a detailed model of the device (e.g., one constructed to represent the behavior of an existing device) and used to derive requirements that must be satisfied by medical processes in which the device is used. Such requirements could give a characterization of the class of medical processes in which it is safe to use the device. The devices we considered in our case studies are relatively small units, but the approach could also be applied to more complex devices or to larger software systems such as computerized order entry systems.

7. ACKNOWLEDGEMENTS

The authors thank Professor Kevin Fu and Professor Elizabeth Henneman, for their help with putting together the case studies, and Jamieson Cobleigh, for sharing his knowledge about the assumption generation methods. This material is based upon work supported by the National Science Foundation under Awards CCF-0820198, CCF-0905530 and IIS-0705772, and by a Gift from the Baystate Medical Center, Rays of Hope Foundation. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the NSF.

8. REFERENCES

[1] D. Alrajeh, J. Kramer, A. Russo, and S. Uchitel. Learning operational requirements from goal models. In *ICSE '09: Proc. of the 2009 IEEE 31st Int. Conf. on Software Eng.*, pages 265–275, Washington, DC, USA, 2009. IEEE Computer Society.

[2] R. Alur, P. Madhusudan, and W. Nam. Symbolic compositional verification by learning assumptions. In Etessami and Rajamani [17], pages 548–562.

[3] R. Alur, P. Černý, P. Madhusudan, and W. Nam. Synthesis of interface specifications for Java classes. *SIGPLAN Not.*, 40(1):98–109, 2005.

[4] D. Angluin. Learning regular sets from queries and counterexamples. *Inf. Comput.*, 75(2):87–106, 1987.

[5] G. S. Avrunin, L. A. Clarke, E. A. Henneman, and L. J. Osterweil. Complex medical processes as context for embedded systems. *SIGBED Rev.*, 3(4):9–14, 2006.

[6] D. Beyer, T. A. Henzinger, and V. Singh. Algorithms for interface synthesis. In W. Damm and H. Hermanns, editors, *CAV*, volume 4590 of *Lecture Notes in Computer Science*, pages 4–19. Springer, 2007.

[7] A. G. Cass, B. S. Lerner, S. M. Sutton, Jr., E. K. McCall, A. Wise, and L. J. Osterweil. Little-JIL/Juliette: a process definition language and interpreter. In *ICSE '00: Proc. of the 22nd Int. Conf. on Software Eng.*, pages 754–757, New York, NY, USA, 2000. ACM.

[8] S. Chaki, E. Clarke, N. Sinha, and P. Thati. Automated assume-guarantee reasoning for simulation conformance. In Etessami and Rajamani [17], pages 534–547.

[9] B. Chen, G. S. Avrunin, E. A. Henneman, L. A. Clarke, L. J. Osterweil, and P. L. Henneman. Analyzing medical processes. In *ICSE '08: Proc. of the 30th Int. Conf. on Software Eng.*, pages 623–632, New York, NY, USA, 2008. ACM.

[10] S. Christov, B. Chen, G. S. Avrunin, L. A. Clarke, L. J. Osterweil, D. Brown, L. Cassells, and W. Mertens. Formally defining medical processes. *Methods of Information in Medicine*, 47(5):392–398, 2008.

[11] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, 1999.

[12] J. M. Cobleigh, D. Giannakopoulou, and C. S. Păsăreanu. Learning assumptions for compositional verification. In *TACAS '03: Proc. of the Ninth Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, volume 2619 of *Lecture Notes in Computer Science*, pages 331–346, New York, NY, USA, 2003. Springer-Verlag Berlin Heidelberg.

[13] R. L. Cobleigh, G. S. Avrunin, and L. A. Clarke. User guidance for creating precise and accessible property specifications. In *SIGSOFT '06/FSE-14: Proc. of the 14th ACM SIGSOFT Int. Symp. on Foundations of Software Eng.*, pages 208–218, New York, NY, USA, 2006. ACM Press.

[14] H. Conboy. Process-based requirement derivation, Department of Computer Science, University of Massachusetts, Amherst, MA 01003 (UM-CS-2010-034), 2010.

[15] C. Damas, B. Lambeau, F. Roucoux, and A. van Lamsweerde. Analyzing critical process models through behavior model synthesis. In *ICSE '09: Proc. of the 2009 31st Int. Conf. on Software Eng.*, pages 441–451, Washington, DC, USA, 2009. IEEE Computer Society.

[16] M. B. Dwyer, L. A. Clarke, J. M. Cobleigh, and G. Naumovich. Flow analysis for verifying properties of concurrent software systems. *ACM Trans. on Software Eng. and Methodology*, 13(4):359–430, 2004.

[17] K. Etessami and S. K. Rajamani, editors. *Computer Aided Verification, 17th Int. Conf., CAV 2005, Edinburgh, Scotland, UK, July 6-10, 2005, Proceedings*, volume 3576 of *Lecture Notes in Computer Science*. Springer, 2005.

[18] K. Fu. Research notes about implantable medical devices, 2006.

[19] D. Giannakopoulou and C. S. Păsăreanu. Interface generation and compositional verification in JavaPathfinder. In *FASE '09: Proc. of the 12th Int. Conf. on Fundamental Approaches to Software Eng.*, pages 94–108, Berlin, Heidelberg, 2009. Springer-Verlag.

[20] D. Giannakopoulou, C. S. Păsăreanu, and H. Barringer. Assumption generation for software component verification. In *ASE '02: Proc. of the 17th IEEE Int. Conf. on Automated Software Eng.*, pages 3–12, Washington, DC, USA, 2002. IEEE Computer Society.

[21] D. Halperin, T. S. Heydt-Benjamin, K. Fu, T. Kohno, and W. H. Maisel. Security and privacy for implantable medical devices. *IEEE Pervasive Computing*, 7(1):30–39, 2008.

[22] D. Halperin, T. S. Heydt-Benjamin, B. Ransford, S. S. Clark, B. Defend, W. Morgan, K. Fu, T. Kohno, and W. H. Maisel. Pacemakers and implantable cardiac defibrillators: Software radio attacks and zero-power defenses. In *SP '08: Proc. of the 2008 IEEE Symp. on Security and Privacy*, pages 129–142, Washington, DC, USA, 2008. IEEE Computer Society.

[23] G. J. Holzmann. *The SPIN Model Checker*. Addison-Wesley, 2004.

[24] P. G. Kelley, P. Hankes Drielsma, N. Sadeh, and L. F. Cranor. User-controllable learning of security and privacy policies. In *AISec '08: Proc. of the First ACM Workshop on AISec*, pages 11–18, New York, NY, USA, 2008. ACM.

[25] M. Peleg, S. W. Tu, J. Bury, P. Ciccarese, J. Fox, R. A. Greenes, R. Hall, P. D. Johnson, N. Jones, A. Kumar, S. Miksch, S. Quaglini, A. Seyfang, E. H. Shortliffe, and M. Stefanelli. Comparing computer-interpretable guideline models: A case-study approach. *JAMIA*, 10:2003, 2002.

[26] A. Pnueli. In transition from global to modular temporal reasoning about programs. In K. Apt, editor, *Logic and Models of Concurrent Systems*, volume 13, pages 123–144, New York, NY, USA, 1984. Springer-Verlag.

[27] R. L. Rivest and R. E. Schapire. Inference of finite automata using homing sequences. In *STOC '89: Proc. of the 21st annual ACM Symp. on Theory of Computing*, pages 411–420, New York, NY, USA, 1989. ACM.

[28] A. ten Teije, M. Marcos, M. Balser, J. van Croonenborg, C. Duelli, F. van Harmelen, P. Lucas, S. Miksch, W. Reif, K. Rosenbrand, and A. Seyfang. Improving medical protocols by formal methods. *Artificial Intelligence in Medicine*, 36(3):193–209, 2006.