Slicing Behavior Tree Models for Verification

Nisansala Yatapanage[1], Kirsten Winter[2], and Saad Zafar[3]

[1] Institute for Integrated and Intelligent Systems, Griffith University, Nathan, QLD 4111, Australia
[2] School of Information Technology and Electrical Engineering, The University of Queensland, St.Lucia, QLD 4072, Australia
[3] Riphah International University, Rawalpindi, Pakistan

Abstract. Program slicing is a reduction technique that removes irrelevant parts of a program automatically, based on dependencies. It is used in the context of documentation to improve the user's understanding as well as for reducing the size of a program when analysing. In this paper we describe an approach for slicing not program code but models of software or systems written in the graphical Behavior Tree language. Our focus is to utilise this reduction technique when model checking Behavior Tree models. Model checking as a fully automated analysis technique is restricted in the size of the model and slicing provides one means to improve on the inherent limitations. We present a Health Information System as a case study. The full model of the system could not be verified due to memory limits. However, our slicing algorithm renders the model to a size for which the model checker terminates. The results nicely demonstrate and quantify the benefits of our approach.

1 Introduction

Detecting problems early in the software life cycle, in the modelling and design phase, would greatly reduce the costs involved. Formal models of systems allow for rigorous analyses to be conducted. The Behavior Tree graphical modelling language [1, 2], which has a formal semantics, has been proposed as a support for engineers handling the complexity of large systems. Behavior Tree models maintain a strong connection with the original textual requirements of the system. Behavior Trees can be automatically translated into model checking languages for verification [3–5].

Model checking [6] is an automated verification technique that exhaustively searches the state space to determine whether or not a given property holds for the system. A major limitation of model checking is what is known as the *state explosion problem*. It refers to the possibly exponential number of states produced from even a small number of system components. This prevents model checking from being applied to larger systems. The model checker may run out of memory resources before providing a result, or may take a prohibitively large amount of time to solve the verification problem.

This paper describes a technique for reducing Behavior Tree models prior to model checking, in order to reduce the time and memory resources required. We propose using a technique known as *slicing* [7], which has been traditionally applied to programs to aid in understanding and debugging (see [8] and [9] for comprehensive surveys).

The objective of this technique is to eliminate those parts of the program that are not relevant in the current context. In doing so slicing can significantly reduce the size of the investigated program.

Although most applications of slicing have been programs, slicing has also been applied to specifications. In particular, slices have been created from Z specifications [10, 11], hierarchical state machines in the RSML specification language [12] and Extended Finite State Machines [13]. The aim of these approaches is to support understanding.

Slicing in the context of automated analysis aims at eliminating those parts of the model that have no effect on whether the property to be checked is satisfied by the model or not. Thus, the approach requires a new type of *slicing criterion* which is derived from the property, often given as a temporal logic formula. This criterion may not refer to a specific program statement; instead several statements could be the slicing targets. In [14], Hatcliff et al. present the foundations of "property-directed" slicing in which formulas given in linear temporal logic (LTL) [15] without using the next state operator provide the slicing criteria.

Whereas Hatcliff et al. apply their technique to Java code, other approaches propose slicing for models of software or systems given in, e.g., the SPIN input language Promela [16], the SAL language [17] and timed automata as used by the UPPAAL tool [18]. Leuschel et al. [19] apply slicing to CSP models and Brückner and Wehrheim [20] investigate models written in CSP-OZ-DC, a language that combines Communicating Sequential Processes, Object-Z and Duration Calculus. Each of these approaches caters for the intricacies of the given modelling language. In our approach the characteristics of Behavior Trees define the avenue of how slicing can be defined such that properties are preserved. A major benefit of slicing is that it requires very little computational resources, as the slicing algorithm runs in close to linear time.

We have applied our slicing algorithm to the Behavior Tree model of a Health Information System in order to evaluate its effectiveness. The results are presented in this paper in the following way: Section 2 provides the reader with preliminaries on the Behavior Tree notation and on slicing in general. In Section 3 we define the main ingredients for Behavior Tree slicing. We introduce our case study in Section 4 and present the results of Behavior Tree slicing and the improvements gained when model checking in Section 5. Section 7 summarises and gives an outlook on future work.

2 Preliminaries

2.1 Behavior Trees

The *Behavior Tree* (BT) notation [1, 2] is a graphical notation to capture the functional requirements of a system provided in natural language. The strength of the BT notation is two-fold: Firstly, the graphical nature of the notation provides the user with an intuitive understanding of a BT model - an important factor especially for use in industry. Secondly, the process of capturing requirements is performed in a stepwise fashion. That is, single requirements are modelled as single BTs, called *individual requirements trees*. In a second step these individual requirement trees are composed into one BT, called the *integrated requirements tree*. Composition of requirements trees is done on

the graphical level: an individual requirements tree is merged with a second tree (which can be another individual requirements tree or an already integrated tree) if its root node matches one of the nodes of the second tree. Intuitively, this merging step is based on the matching node providing the point at which the preconditions of the merged requirement trees are satisfied. This structured process provides a successful solution for handling very large requirements specifications [1, 21].

The syntax of the BT notation comprises nodes and edges. A node can be either a state realisation describing a state change of a component or one of its attributes, or a selection, guard, or event, guarding the control flow within the BT. A node is specialised by its *Component name* C, *Behavior* B, *Type*, and set of *Flags*.

The *Behavior* is an identifier (describing a state, an event, or a channel name), or an expression (referring to component attributes).
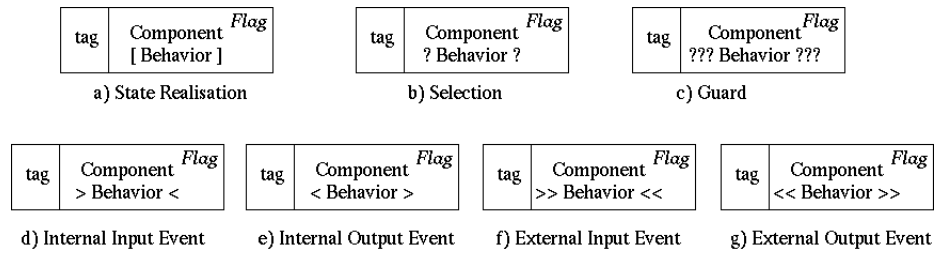
tag	Component ^{Flag} [Behavior]		tag	Component ^{Flag} ? Behavior ?		tag	Component^{Flag} ??? Behavior ???

      a) State Realisation   b) Selection   c) Guard

tag	Component ^{Flag} > Behavior <	tag	Component ^{Flag} < Behavior >	tag	Component ^{Flag} >> Behavior <<	tag	Component ^{Flag} << Behavior >>

d) Internal Input Event   e) Internal Output Event   f) External Input Event   g) External Output Event

Fig. 1. Different node types of the BT syntax

The *Type* of a BT node can be (c.f., Figure 1)

(a) a state realisation, modelling C being in a state if B is a state name, or updating C's attribute if B is an update expression over the attribute;

(b) a selection (or condition) on C's state if B is a state name, or a selection on the state of one of C's attributes if B is an expression over the attribute; in both cases, the control flow terminates if the condition is not satisfied;

(c) a guard; the control flow can pass the guard when C is in state B if B is a state name, or when the expression B over one of C's attributes is satisfied if B is an expression over the attribute; otherwise it is blocked until the state realisation occurs;

(d-e) an internal event modelling communication and data flow between components within the system, where B specifies an event; the control flow can pass the internal input/output event node when the event occurs (the message is sent), otherwise it is blocked until it occurs;

(f-g) an external event modelling communication and data flow between the system and its environment, where B specifies an event; the control flow can pass the external input/output event node when the event occurs (the message is sent), otherwise it is blocked until it occurs.

The control flow of the system is modelled by either a normal or a branching edge. A normal arrowed edge models *sequential flow* between two steps. If two nodes are connected by a line without an arrow head the two steps occur together atomically. Figure 2 shows the two types of branching edges: concurrent and alternative. *Concurrent branching* (Figure 2a) models threads running in parallel. As an example the threads in
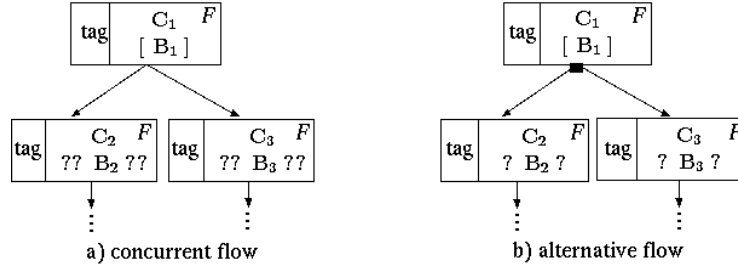
Fig. 2. Branching structures in the BT syntax

the figure start with a guard node. The branches, however, can start with any node type. We show only two sub-trees in the branching, although in general there may be more.

In *alternative branching* (Figure 2b), the control flow follows only one of the branches. Alternative branches can comprise either selections only (for example, as shown in Figure 2b) or only other node types but no selections. Alternative branching over selections operates as a non-deterministic choice over the branches with a satisfied selection condition B_i. If none of the selections is satisfied the behaviour terminates. Alternative branching over non-selections behaves like a non-deterministic choice that is unguarded.

Flags in a BT node can specify: (a) a *reversion* node, marked by '^', if the node is a leaf node, indicating that the control flow loops back to the closest matching ancestor node (a matching node is a node with the same component name, type and behaviour) and all behaviour started after the matching ancestor node is terminated; (b) a *referring* node, marked by '=>', indicating that the flow continues from the matching node; (c) a *thread kill* node, marked by '$--$', which kills the thread that starts with the matching node, or (d) a *synchronisation* node, marked by '=', where the control flow waits until all other threads with a matching synchronisation node have reached the synchronisation point.

As described in [22], the BT syntax also includes notation for standard set operations, some of which are shown in Figure 3 on page 4 (for a complete description of the syntax see [23]): Assume C has an attribute S which is a set, then a) models adding element x to S, b) removing element x from S, c) queries the cardinality of S, and d) queries membership of x in S. Union, difference and intersection of two sets S and T can be specified using the following syntax: $S := S + T$, $S := S - T$, and $S := S \times T$.
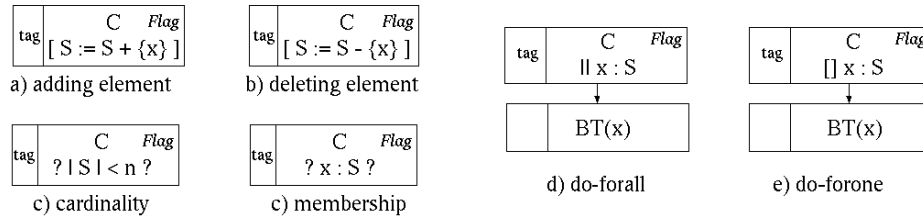

Fig. 3. Set Operations and Parametrisation of BTs

In addition to standard set operations, the syntax also provides constructs for para-meterisation such that a sub-BT is to be performed on all members or one member of a reference set S. Figure 3d) models execution of a sub-tree BT(x) for all members x of set S, and Figure 3e) models execution of BT(x) for some member x of set S (where x is chosen non-deterministically). These constructs are referred to as *do-forall* and *do-forone*, respectively. The component name can also be omitted.

Type declarations and other structural information about the system model are captured in a *Composition Tree* (CT). We do not introduce the notion of CTs here but refer the interested reader to [23].

The semantics of BTs is formalised in [24] using CSP_σ [25] which is an extension of CSP with state. An automated translation [3, 4] provides an interface to the model checker SAL [26], allowing for fully automated analysis of BT models.

2.2   Program Slicing

Program slicing is a technique which removes irrelevant parts of a program based on the dependencies between the program statements. The program is first transformed into a Control Flow Graph (CFG). The nodes in the CFG represent the program statements and the edges correspond to control flow. There is a single entry node and a single exit node for each graph. Branching statements, such as if or while statements, are represented as nodes in the CFG with two successors representing the true and false paths. All other statements are represented as nodes with a single successor. In many approaches to slicing, a structure known as a Program Dependence Graph (PDG) [27, 28] is then created using the CFG. The PDG is a directed graph with vertexes representing program statements and edges representing dependencies, such as control-flow and data dependencies. To determine the set of statements that may directly or indirectly affect a specified criterion, a simple reachability analysis is performed on the graph. The statements that were found to be irrelevant to the criterion can then be removed, producing the slice. The slice will always be a subset of or equal to the original program.

A formal foundation of control dependencies and slicing correctness can be found in the work by Ranganath et al. [29].

3   Slicing of Behavior Trees

Slicing of BTs is performed in a similar manner as program slicing. We utilise this technique to reduce the size of a BT model when applying model checking. Model checking is a fully automated process which, in our case, checks whether a BT model satisfies a given temporal logic property. We assume the property is specified in LTL_X [6], which is linear temporal logic (LTL) [15] minus the next step operator X. A slice of a model wrt. a given property is created in such a way that it preserves the property. Assuming this, it suffices to model check the smaller slice instead of the full model.

The first step in the overall slicing process is to create a *Behavior Tree Dependence Graph* (BTDG) from the given BT. The BTDG indicates all the dependencies that exist between the nodes of the BT. Each BT will have only one *general* BTDG which is

independent of the property to be verified but covers *all* dependencies between all components and attributes. Therefore this step only needs to be performed once per BT and the resulting BTDG can be used for creating slices for any LTL_X property as well as for any other analysis technique that relies on dependency graphs. The set of nodes that are relevant for a specific property are then identified using the BTDG. Finally, these nodes are re-formed into a tree, producing the *BT slice* for the given property.

3.1 Creating the BT Dependence Graph

The BTDG is similar to a Program Dependence Graph [27, 28]. Each node in a BTDG represents a node in the BT and each edge represents a dependency between two nodes. There are several types of dependencies that might be present in a BT, specifically: control, data, interference, synchronisation, message and alternative choice branching dependencies.

A BTDG is created not from the BT directly but from the *control flow graph of a BT* (CFG-BT). In the context of program slicing, a control flow graph indicates the flow of control between single program statements. A BT is almost a like a control flow graph as it shows the flow of control via sequential and branching edges. However, some parts of the control flow are denoted by boxes rather than edges, e.g. reversion and reference nodes. These have to be replaced by edges linking the predecessor node in the BT with its successor node. Moreover, selection, guard and (internal/external) input event nodes in a BT do not explicitly show the flow of control in the case where this respective condition fails (i.e., the behaviour in the negative case). We add edges to represent these cases: (a) each selection node has an additional outgoing edge to a terminal node representing termination when the selection condition is not satisfied, (b) guard and input event nodes have an additional edge that loops back to itself representing blocking behaviour (i.e., waiting) in the case where the guard or event is not available.

A *path* in a CFG-BT is a sequence of nodes. Let $Path(p,q)$ represent the path from node p to node q. If $k \in Path(p,q)$, then k is a node on the path. A path is called *maximal* if it either terminates (i.e. it has no successors) or contains an infinite loop (this definition was adapted from [29]).

We call two nodes p and q *matching*, denoted as $Matching(p,q)$, if they have the same component name, behavior and type. If $Concurrent(p,q)$ then nodes p and q are in concurrent threads of the CFG-BT. If $Alternate(p,q)$ then nodes p and q are in alternate branches of the same thread of the CFG-BT.

We now define the notion of *definition set* and *reference set* for BT nodes which are used to define dependencies between nodes. Intuitively, if a node updates/queries a component or attribute, the component or attribute is a member of the definition/reference set for that node. Assume in the following that C is a component, a and b are attributes of the component, s is a behavior, g is a guard, S and T are sets, x denotes an element of S, and m is a natural number.

Definition 1. (Definition Set)
$DEF(n)$ represents the set of components and attributes defined *at node n. Specifically, if n is of the form*

(a) (state-realisation) C[s] then $C \in DEF(n)$,

(b) (state-realisation of attributes) C[a := s] then $a \in DEF(n)$,
(c) (adding/removing an element from a set) C[S := S + x] or C[S := S - x] then $S \in DEF(n)$,
(d) (union, subtraction, intersection of sets) C[S := S + T] or C[S := S - T] or C[S := S $\asymp$ T] then $S \in DEF(n)$.

Definition 2. (Reference Set)
Let $REF(n)$ represent the set of components and attributes referenced at node n. Specifically, if n is of the form

(a) (selection/guard) C?g? or C???g??? then $C \in REF(n)$,
(b) (selection/guard over attributes) C?a = exp? or C???a = exp??? where exp is an expression or a behavior, then $a \in REF(n)$,
(c) (state-realisation of attribute) C[a := f(b)], where $f(b)$ is an expression over b, then $b \in REF(n)$,
(d) (selection over set predicates) C?x : S? or C?S = {}? or C?|S| $\bowtie$ m? where $\bowtie \in \{=, <, >, <=, >=\}$, then $S \in REF(n)$.

Based on these preliminary definitions we now define various types of dependencies between BT nodes from which the BTDG is built.

Control dependencies arise if a BT node controls the execution of another node. A node p in a BT is control-dependent on a node q if and only if there are two possible outcomes after executing node p: in one scenario all paths of execution lead to q and in the other scenario, there exists a path on which node q is never executed. This situation occurs when node p is either a guard, a selection or an external input event. For instance, in the case of a selection, if the selection is satisfied then the control flow proceeds to subsequent nodes, but if the selection is not satisfied then the control flow terminates. Thus, the execution of the subsequent nodes are controlled by the selection node. We exclude dependencies between nodes from alternative and concurrent branches as those are handled separately.

Definition 3. (Control Dependency)
For two nodes p and q in a CFG-BT, node q is control-dependent *on node p, denoted as $p \xrightarrow{cd} q$, iff:*

(i) node p has at least two successors m and n, where NOT(Alternate(m, n)) and NOT(Concurrent(m, n)),
(ii) for all maximal paths from node m, node q always occurs and
(iii) there exists a maximal path from node n on which node q never occurs.

If a node q in a BT queries the state of a component or attribute, it is *data-dependent* on any node p that updates the state of that component or attribute. That is, if for a node p a component or attribute c is in $REF(p)$, then p is data-dependent on a node q for which c is a member of $DEF(q)$, assuming c is not re-defined by another node on the path between p and q. This dependency occurs within a single thread as well as between parallel threads. In the latter case it is often also referred to as *interference dependence*.

These two dependencies are differentiated because interference dependency is intransitive (in contrast to all other dependencies). This results in slices that are not optimal as they contain nodes that are not relevant for the property and could have been sliced away. Krinke in [30] introduces the notion of *threaded witness* to determine whether a path in the dependency graph shows a true dependency. In our approach it remains future work to include a strategy for optimising the slicing algorithm that handles this problem.

The following definition combines data and interference dependency as both notions differ only in one condition: $NOT(Concurrent(p, q)$ for data dependency and $Concurrent(p, q)$ for interference dependency.

Definition 4. (**Data and Interference Dependency**)

For two nodes p and q in a CFG-BT, node q is data- or interference-dependent *on node p, denoted as $p \xrightarrow{dd/id} q$, iff:*

(i) *$\exists c \in DEF(p)$ such that $c \in REF(q)$ and*
(ii) *$\forall k \in Path(p, q), c \notin DEF(k)$.*

Message dependence is similar to data dependence. All internal input nodes are message-dependent on internal output nodes with the same message name. (Note that for external input/output nodes we do not have a similar dependency as the sender/receiver of the message is outside the scope to the system (i.e., external).)

Definition 5. (**Message Dependency**)

For two nodes p and q in a CFG-BT, node q is message-dependent *on node p, denoted as $p \xrightarrow{md} q$, iff:*

(i) *$Type(p) = InternalOutput$ and $Behavior(p) = m$ and*
(ii) *$Type(q) = InternalInput$ and $Behavior(q) = m$.*

All sets of synchronising nodes are dependent on each other, because each node must wait for all the others before proceeding.

Definition 6. (**Synchronisation Dependency**)

For two nodes p and q in a CFG-BT, node q is synchronisation-dependent *on node p, denoted as $p \xrightarrow{sd} q$, iff:*

(i) *$Flag(p) = Synchronisation$ and $Flag(q) = Synchronisation$ and*
(ii) *$Matching(p, q)$.*

For BTs an extra dependency type arises that is not normally found in program slicing. A BT can have alternative choice branching points. At each of these points, each branch is dependent on whether or not the other branches have executed. If one branch executes then all others are terminated. Therefore, there is a dependency between the root nodes of all branches in an alternative branching construct.

Definition 7. (**Alternative Dependency**)

For two nodes p and q in a CFG-BT, node q is alternative-dependent *on node p, denoted as $p \xrightarrow{ad} q$, iff nodes p and q:*

(i) have the same parent node and
(ii) are connected by an alternative choice branching point.

Note that synchronisation as well as alternative dependency are symmetrical relations between BT nodes.

3.2   Creating the Slice

After the BTDG has been created for a given BT, it can be used to create the slice for any given LTL_X property. The *slicing criterion* consists of all state-realisation nodes which update the state of one of the components or attributes mentioned in the temporal logic property. If we lift the notion of reference set to temporal logic formulas such that for any formula p, $REF(p)$ denotes the set of components and component attributes that are mentioned in p, then the set of nodes building the slicing criterion for p is characterised as $SliceCrit(p) = \{n : BTNode \mid \exists c \in REF(p) \cdot c \in DEF(n)\}$. Starting from each of these nodes in $SliceCrit$, the BTDG is traversed in reverse, collecting every node that is encountered via dependency edges. During the traversal we check if a node has been visited already in order to avoid an infinite traversal due to symmetric or circular dependencies. The algorithm is in most cases linear and in the worst, but unrealistic, case quadratic in the number of nodes of the BTDG. Any nodes that were not encountered during the traversals can be removed from the BT, because they are irrelevant for the property.

The final set of relevant nodes is often a disjoint set of sub-trees. These must be re-formed into a tree. Every node that is missing its parent node is joined to its closest remaining ancestor, becoming one of its children. However, if the node was originally part of an alternative or concurrent branch and one or more of the other branches are still present in the tree, then the root nodes of each branch are joined to a $blanknode$. A $blanknode$ is a place-holder for the concurrent or alternative branching construct of the original BT and is used in order to preserve the structure. The $blanknode$ then becomes a child of the closest remaining ancestor of the branching nodes.

The reversion and reference nodes are then added back to the slice. These nodes represent jumps to other locations in the tree. The nodes are placed at the bottom of the same threads that they belonged to in the original BT. However, if an entire sub-tree no longer exists in the slice, then any reversions/reference nodes in that sub-tree are left out.

If the BT contains *do-forall* and *do-forone* nodes, then these are added back to the slice, unless these nodes are no longer relevant. The nodes become obsolete if the sub-tree below no longer mentions the member of the reference set (i.e., the parameter).

The final slice can only be used in place of the original BT for model checking the desired property if our slicing algorithm, which is based on the above definitions, is correct. We have formally proved the correctness of our approach and summarise the proof idea in the following. Similar to [29] we base our proof on the notion that nodes in the slicing criterion of a property p, $SliceCrit(p)$, are *observable* steps whereas nodes not in the slicing criterion are silent or *stuttering* steps. The idea of our proof is to show that our algorithm produces a slice that is stuttering-equivalent to the original BT. We base our proof on the theorem on stutter equivalence and LTL_X equivalence

([6], chapter 7) which states that for any paths σ_1, σ_2 and any LTL_X formula φ (over the same set of atomic propositions) $\sigma_1 \mathbin{\hat{=}} \sigma_2 \implies \sigma_1 \models \varphi$ *iff* $\sigma_2 \models \varphi$, where $\hat{=}$ denotes stuttering equivalence between two paths. That is, if two models are stuttering equivalent we can imply that both of them satisfy or dissatisfy the same properties.

Using these results it suffices in our context to show that for any behaviour in the original BT there exists a stuttering equivalent behaviour in the slice that is generated wrt. a formula φ. With the definitions as given in Section 3.1 a proof by contradiction provides the desired result. We assume that there exists an execution trace in the original BT for which there does not exist a stuttering-equivalent trace in the slice. Therefore the trace of the original BT must at some point contain an observable node that is not able to execute at the equivalent point in any of the traces of the slice. However, we have shown that such a node cannot exist according to the definitions of slicing and dependencies. We then assume that the converse is true: that there exists an execution trace in the slice that does not have a stuttering-equivalent trace in the original BT. We have also shown this to be impossible. Thus, a generated slice is always stuttering-equivalent to the original BT.

4   Case Study: the e-Health System

The e-Health system presented here has been adapted from a real case study presented in [31]. It is based on the automation requirements for an aged care facility. The facility provides accommodation and care for elderly residents. The facility is administered by a manager. The residents are visited by doctors regularly. Managers, residents and doctors can view and edit relevant electronic records. The resident's data is made up of the following elements. (1) At the time of admission the personal details of the residents are entered. These details include basic personal data (name, sex, etc.), medical details (blood group, allergies, etc.) and details of any nominated responsible person. (2) Before admission a legal agreement must also be signed electronically. (3) A care plan is initiated after he/she has been admitted to the facility. (4) Past medical records are entered by the manager of the facility. After each examination the doctor adds an entry to the medical records of the residents. The doctor can also add private notes to the medical records.

We have created a BT model of the e-Health system. Fig 4 shows an overview of the model, which is too large to be shown in further detail here. The BT describes the behaviour of the nominated responsible people, the managers, the doctors and the residents, each in a separate thread. A responsible person can give consent to add a doctor or sign an agreement on behalf of the patient. A manager can perform various tasks, including adding or viewing a patient's personal details, managing care plans, adding medical records and deleting data. A doctor can view or add medical records if he/she has permission to do so. Doctors, residents and temporary doctors can be added to the various access control lists, thereby granting them access to a patient's files.

In order to provide the reader with an idea of the structure of the tree, a section of the manager thread is shown in greater detail in Figure 5. The manager thread begins with "for all managers", "choose a resident", followed by "choose a data set". At this point there is an alternative choice between each of the possible actions that a manager
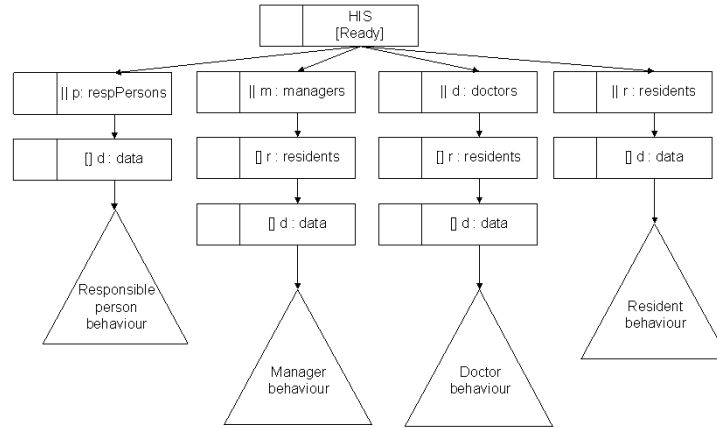
Fig. 4. Overview of the e-Health System BT

can perform. We show only the action of the manager viewing personal details. If the current data belongs to the current resident, then the manager can view the data. The behaviour then reverts to a higher node, to allow managers to perform other actions.

A number of access control requirements have been defined for the security of electronic records and to ensure the privacy of the residents. We have formulated these requirements as LTL theorems, shown below. To comply with different regulations, the resident records must be kept for a period of nine years. After the completion of the nine year period the record can be deleted. This requirement is formalised as theorem 1. The LTL operator, G(p), is used to state that p is always true. The second requirement concerns the ability of managers to add medical records. To protect the privacy of the residents, the access to medical records is limited only to the residents themselves or to the nominated doctors. However, the manager is allowed to enter past medical records at the time of admission to the facility. After the admission, the manager is not allowed to add or view the records. The formula states that medical records can only be added by a manager if the patient associated with the data has not yet been admitted. The last property states that a resident can view his/her private notes only if he/she has been assigned to the record's access control list (ViewNotesACL).
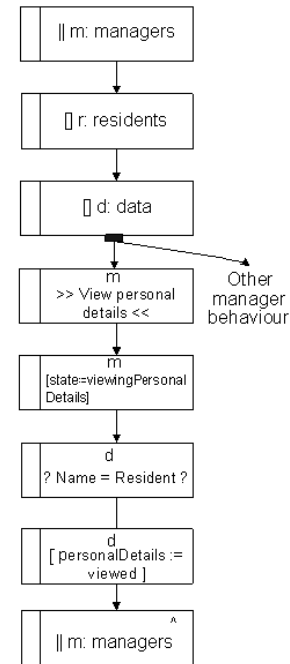

Fig. 5: Manager thread

1. Data can only be deleted by the manager if its leave date is greater than nine years.
   $\forall d : data, \forall m : managers,$
   $\mathbf{G}\,((d.deleted = true \;\wedge\; m.state = deletingData)$
   $\qquad \implies\; d.leaveDate = greaterThanNineYears)$

2. Medical records can only be added by the manager if the patient associated with the data has not yet been admitted. $\forall d : data, \forall m : managers,$
$\mathbf{G} \, ((d.medicalRecords = added \, \wedge \, m.state = addingMedicalRecords)$
$\qquad \implies \quad d.admitted = false)$

3. A resident can only view his/her private notes if he/she is in the set ViewNotesACL. $\forall d : data, \forall r : residents,$
$\mathbf{G} \, ((d.privateNotes = viewed \, \wedge \, r.state = viewingPrivateNotes)$
$\qquad \implies \quad (r : d.ViewNotesACL)$

5   Slicing the model of the e-Health system

In this section we compare the execution times for model checking the e-Health system BT with the slices for each property listed in Section 4. First, the BTDG for the e-Health system is created, according to the procedure outlined in Section 3. This BTDG is then used for creating each of the slices. The slicing criterion for each case is derived from the variables mentioned in the property. For example, the slicing criterion for theorem 1 is the set $\{d.deleted, m.state, d.leaveDate \mid d \in data \wedge m \in manager\}$. Starting from each of the state-realisation nodes that involve these components/attributes, the BTDG is traversed in reverse, collecting all relevant nodes.

The size of the resulting slices varies depending on the property. The only relevant part for theorem 1 is the manager's behaviour, so a significant proportion of the BT can be sliced away. The other two properties involve several parts of the BT, so the corresponding slices are larger than for theorem 1. For example, theorem 3 involves the set ViewNotesACL. This set is referred to in several places in the resident behaviour thread. However, the set is updated in the doctor behaviour thread, so the validity of the property is dependent on both parts of the BT. Table 1 lists the number of nodes in each of the slices compared to the original BT.

	Original BT	Slice Th1	Slice Th2	Slice Th3
No. of nodes	125	36	116	73

Table 1. Number of nodes in the original BT and each slice

The e-Health system BT utilises set constructs, so the BT is expanded before model checking. Whenever a *do-forall* or *do-forone* node is encountered, its sub-tree is replicated for each element in the set. We have compared the execution times for model checking the original BT vs. the slices, for increasing cardinality of the sets. The results are presented in Table 2. The system maintains sets of doctors, residents, responsible persons, managers, data and logs.

The checks were run on a 1.2GHz UltraSPARC processor with 24GB of RAM, running Solaris 10. The original model could not be verified. Even with only a single user in each set, i.e. the smallest possible model, the model checker ran out of memory before providing a result. However, using the slices, verification was possible. For all three theorems, the model checker provided a result in only seconds when the sets

	1 All	2 D	2 M	2 R	2 RP	2 D, 2 M
Original Th1	MEM	-	-	-	-	-
Slice Th1	2.2s	2.4s	11.4s	8.7s	2.4s	11.4s
Original Th2	MEM	-	-	-	-	-
Slice Th2	70.8s	10.6hrs	35.4mins	1.2hrs	16.9mins	MEM
Original Th3	MEM	-	-	-	-	-
Slice Th3	10.7s	6.2mins	1.1min	8.9mins	19.4s	31.6mins

	2D, 2 R	2 M, 2 R	2 All, 1 data	2 All, 2 data	3 All, 1 data
Original Th1	-	-	-	-	-
Slice Th1	9.0s	1.4mins	1.8mins	23.5mins	2.1hrs
Original Th2	-	-	-	-	-
Slice Th2	MEM	MEM	MEM	-	-
Original Th3	-	-	-	-	-
Slice Th3	MEM	4.1hrs	18.6hrs	-	-

Table 2. Execution times for model checking the original BT and slices, where: MEM = out of memory, n All = n elements in each user set, 2 D = 2 doctors, 2 R = 2 residents, 2 M = 2 managers, 2 RP = 2 responsible people, 1 data = 1 data set and 1 log, 2 data = 2 data sets and 2 logs; and if a set is not specified then it contained 1 element.

contained one element each. As the number of elements were increased, the execution times increased, but a result could still be obtained for most combinations in which there were two elements in one or more of the sets.

The slice for theorem 1 did not contain any nodes involving doctors or responsible persons, so increasing these sets did not affect the execution time. When the number of managers and residents were increased, the execution time increased, reaching 11 seconds with 2 managers and 1-2 minutes with 2 managers and 2 residents. Even with 3 elements in each of the sets a result was obtained in 2 hours.

The size of the slice for theorem 2 is almost the same as the original BT. However, there was still significant improvement in model checking. It was possible to obtain a result with one element in each set in only 71 seconds. With two elements in a set, the execution times varied between 17 minutes for 2 responsible persons and 10.6 hours for 2 doctors. If two sets contained two elements, such as 2 doctors and 2 residents, the model checker ran out of memory.

Model checking the slice for theorem 3 took only 11 seconds for one element in each set and less than 10 minutes for two elements in a set. With 2 elements in each of the user sets, the model was still able to be verified although the execution time reached 18.6 hours.

6 Discussion

Slicing can be considered to be complementary to other state-space reduction techniques. The low computational cost needed for slicing makes it ideal to be performed

as an initial step prior to applying other reduction techniques. The extent to which slicing can reduce the verification time depends on both the model and the property. The more dependencies that exist between the components in the formula, the larger the slice. Despite this, in many cases slicing can improve the verification time. In particular, in concurrent models it is often the case that each thread represents a separate component that operates independently. In such cases, slicing is very effective as it reduces the parallelism in the model, which is one of the most computationally expensive aspects in model checking. Due to the low cost for applying slicing, it is worthwhile to try it, especially if the model is infeasible for a model checker.

7 Conclusion

We have developed a method for automatically reducing Behavior Tree models using the slicing. Slices of a BT model can be used for aiding in understanding and also for reducing models prior to verification. The case study we have presented in this paper demonstrates the benefits of slicing a model before model checking. The results show the improvements in execution time and memory usage. This enables the verification of models for which model checking was previously infeasible. Although the shape and size of a slice depend on the property to be checked, the low computational resources needed to compute a slice makes it ideal for aiding in model checking.

For future work, we plan to extend the slicing procedure to enable slicing wrt. full LTL (including X operator) and to explore further optimisations which could produce greater reductions in the model size.

References

1. Dromey, R.G.: From requirements to design: Formalizing the key steps. In: Proc. of Software Engineering and Formal Methods (SEFM 2003), IEEE Computer Society (2003) 2–13
2. Dromey, R.G.: Genetic design: Amplifying our ability to deal with requirements complexity. In: Models, Transformations and Tools. Volume 3466 of LNCS., Springer (2005) 95–108
3. Grunske, L., Lindsay, P.A., Yatapanage, N., Winter, K.: An automated failure mode and effect analysis based on high-level design specification with behavior trees. In Romijn, J., Smith, G., van de Pol, J., eds.: Proc. of Int Conf. on Integrated Formal Methods (IFM 2005). Volume 3771 of LNCS., Springer (2005) 129–149
4. Grunske, L., Winter, K., Yatapanage, N.: Defining the abstract syntax of visual languages with advanced graph grammars-a case study based on Behavior Trees. Journal of Visual Language and Computing **19**(3) (2008) 343–379
5. Colvin, R., Grunske, L., Winter, K.: Timed behavior trees for failure mode and effects analysis of time-critical systems. Journal of Systems and Software **81**(12) (2008) 2163–2182
6. Baier, C., Katoen, J.P.: Principles of Model Checking. MIT Press (2008)
7. Weiser, M.: Program slicing. In: Proc. of Int. Conf. on Software Engineering (ICSE'81). (1981) 439–449
8. Tip, F.: A survey of program slicing techniques. Journal of Programming Languages **3**(3) (1995) 121–189
9. Xu, B., Qian, J., Zhang, X., Wu, Z., Chen, L.: A brief survey of program slicing. SIGSOFT Softw. Eng. Notes **30**(2) (2005) 1–36

10. Oda, T., Araki, K.: Specification slicing in formal methods of software development. In: Proc. of Computer Software and Applications Conference (COMSAC 93), IEEE (2005) 313–319

11. Wu, F., Yi, T.: Slicing Z specifications. ACM SIGPLAN Notices **39**(8) (2004) 39–48

12. Heimdahl, M., Whalen, M.: Reduction and slicing of heirarchical state machines. In Jazayeri, M., Schauer, H., eds.: Proc. of European Software Engineering Conference (ESEC/FSE 97). Volume 1301 of LNCS., Springer-Verlag (1997) 450–467

13. Dorel, B., Singh, I., Tahat, L., Vaysburg, S.: Slicing of state-based models. In: Proc. of Int. Conf. on Software Maintenance (ICSM 2003), IEEE (2003) 34–43

14. Hatcliff, J., Dwyer, M., Zheng, H.: Slicing software for model construction. Higher-Order and Symbolic Computation **13**(4) (2000) 315–353

15. Emerson, E.A.: Temporal and modal logic. In van Leeuwen, J., ed.: Handbook of Theoretical Coomputer Science. Volume B. Elsevier Science Publishers (1990)

16. Millett, L., Teitelbaum, T.: Slicing promela and its applications to model checking, simulation and protocol understanding. In: Proc. of Int. SPIN Workshop. (1998)

17. Ganesh, V., Saidi, H., Shankar, N.: Slicing SAL. Technical report, Computer Science Laboratory (1999)

18. Thrane, C.: Slicing for UPPAAL. In: Ann. IEEE Conf. (Student Paper), IEEE (2008) 1–5

19. Leuschel, M., Llorens, M., Olivier, J., Silva, J., Tamarit, S.: The MEB and CEB static analysis for CSP specifications. In Hanus, M., ed.: Proc. of Logic-Based Program Synthesis and Transformation (LOPSTR'08). Number 5438 in LNCS, Springer (2008) 103–118

20. Brückner, I., Wehrheim, H.: Slicing an integrated formal method for verification. In: Formal Methods and Software Engineering. Volume 3785 of LNCS., Springer (2005) 360–374

21. Wen, L., Dromey, R.G.: From requirements change to design change: A formal path. In: Proc. of Int. Conf. on Software Engineering and Formal Methods (SEFM 2004), IEEE Computer Society (2004) 104–113

22. Zafar, S., Colvin, R., Winter, K., Yatapanage, N., Dromey, R.G.: Early validation and verification of a distributed role-based access control model. In: Proc. of Asia-Pacific Software Engineering Conference (APSEC 2007), IEEE Computer Society (2007) 430–437

23. Dromey, G.R.: Behavior Engineering. http://www.behaviorengineering.org

24. Colvin, R., Hayes, I.J.: A semantics for Behavior Trees. Technical Report SSE-2010-03, The University of Queensland (May 2010) http://espace.library.uq.edu.au/view/UQ:204809.

25. Colvin, R., Hayes, I.J.: Csp with hierarchical state. In Leuschel, M., Wehrheim, H., eds.: Proc. of Int. Conf. on Integrated Formal Methods (IFM 2009). Volume 5423 of LNCS., Springer (2009) 118–135

26. de Moura, L., Owre, S., Rueß, H., Rushby, J., Shankar, N., Sorea, M., Tiwari, A.: SAL 2. In Alur, R., Peled, D., eds.: Int. Conference on Computer-Aided Verification, (CAV 2004). Volume 3114 of LNCS., Springer (2004) 496–500

27. Ottenstein, K.J., Ottenstein, L.M.: The program dependence graph in a software development environment. SIGSOFT Softw. Eng. Notes **9**(3) (1984) 177–184

28. Ferrante, J., Ottenstein, K.J., Warren, J.D.: The program dependence graph and its use in optimization. ACM Trans. Program. Lang. Syst. **9**(3) (1987) 319–349

29. Ranganath, V.P., Amtoft, T., Banerjee, A., Hatcliff, J., Dwyer, M.B.: A new foundation for control dependence and slicing for modern program structures. ACM Trans. Program. Lang. Syst. **29**(5) (2007) 27

30. Krinke, J.: Static slicing of threaded programs. SIGPLAN Notices **33**(7) (1998) 35–42

31. Evered, M., Bögeholz, S.: A case study in access control requirements for a health information system. In: Proc. of Workshop on Australasian Information Security, Data Mining and Web Intelligence, and Software Internationalisation. Volume 32., Australian Computer Society, Inc. (2004) 53–61