Slicing Behavior Trees for Verification of Large Systems

Nisansala Prasanthi Yatapanage

B.E. (Honours Class I)

INSTITUTE FOR INTEGRATED AND INTELLIGENT SYSTEMS

SCIENCE, ENVIRONMENT, ENGINEERING AND TECHNOLOGY GROUP

GRIFFITH UNIVERSITY

Submitted in fulfillment of the requirements of the degree of

Doctor of Philosophy

October 2011

ABSTRACT

It is essential to ensure the correctness of software systems, especially for large and safety-critical applications. Detecting problems earlier in the software cycle, such as in the specification and design phases, would significantly reduce the costs involved. Rigorous automated approaches are ideal for detecting such problems. Model checking is an automated verification technique which exhaustively searches the state space to determine whether a model of the system satisfies a given property. However, model checking suffers from state explosion, preventing large systems from being verified.

The Behavior Tree specification language enables engineers to handle the complexity of large systems, by allowing them to focus on one requirement at a time. Behavior Trees maintain strong links to the original requirements of the system. There has been support for automatic translation of Behavior Trees into model checking languages. However, due to the state explosion problem, large Behavior Trees still cannot be verified.

Program slicing is a reduction technique which automatically removes irrelevant portions of the program, usually applied for improving understanding and debugging. In this thesis, a technique for reducing Behavior Trees prior to verification is proposed, based on the concepts of program slicing. The technique is shown to preserve all properties specified in the language CTL^*_{-X}, which is CTL^* without the *next* operator. Thus, a property will be proved on the sliced model if and only if it is proved on the original model. The slicing approach is demonstrated on two case studies, producing significant reductions in verification time.

A new optimisation technique is also proposed, to allow the Behavior Tree to be reduced even further, by eliminating nodes which are infeasible. The technique is able to reduce slices more than previous approaches. The optimisation technique is also shown to preserve CTL^*_{-X} properties.

No other slicing method is able to preserve properties which contain the *next* operator. Another contribution of this thesis is a novel method for producing slices that are able to preserve *full* CTL^*, including properties containing the *next* operator.

This technique is shown to be correct by establishing a new form of branching bisimulation, known as *next-preserving branching bisimulation*. Weak forms of bisimulation are normally unable to preserve properties containing the *next* operator. *Next-preserving branching bisimulation* is shown to preserve full CTL^*, which includes the *next* operator. The new form of bisimulation allows similar techniques to be developed for other modelling languages as well, since all that is required is to show the establishment of a next-preserving branching bisimulation.

The final outcome of the thesis is a slicing approach which can effectively reduce Behavior Trees, in order to allow large systems to be verified. Furthermore, the thesis gives useful theoretical results about property preservation of full CTL^* by *next-preserving branching bisimulation.*

This work has not previously been submitted for a degree or diploma in any university. To the best of my knowledge and belief, the thesis contains no material previously published or written by another person except where due reference is made in the thesis itself.

_____

Nisansala Yatapanage

This thesis is dedicated to my father Dr. Kamal Yatapanage, my mother Ranjani Yatapanage and my sister Sadhana Yatapanage.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

First of all, I would like to express my deepest gratitude to my supervisor Kirsten Winter, for all the time and effort she spent on selflessly checking and improving my proofs, reading through my thesis and discussing my ideas and for all the guidance she gave me. Her advice and encouragement throughout all stages of the project were invaluable and will always be of benefit to me.

Next, I would like to thank my former supervisor, the late Geoff Dromey, for giving me the opportunity to work with the Behavior Tree research group and for all the support and guidance he has given me over the years.

I am grateful to Abdul Sattar, for replacing Geoff as supervisor towards the end of my project and for always being supportive of me. I would also like to thank Larry Wen for taking over as secondary supervisor.

I would like to sincerely thank Natalie Dunstan, for always efficiently taking care of the administrative issues and ensuring that I had all the facilities I required.

Thank you to Peter Lindsay for giving me the opportunity to be a visiting researcher at the University of Queensland and for providing me with access to the UQ computing facilities to perform my case study experiments. I would also like to thank Saad Zafar for permitting me to use his Behavior Trees for my case studies.

I would like to thank the following Behavior Tree researchers for all the useful discussions over the years and for giving me the opportunity to work with them on many projects: (in alphabetical order) Rob Colvin, Geoff Dromey, Lars Grunske, Ian Hayes, Diana Kirk, Sentot Kromodimoeljo, Peter Lindsay, Toby Myers, John Seagrott, Cameron Smith, Larry Wen, Kirsten Winter and Saad Zafar. Thank you also to Michelle Christie for giving me encouragement over the years.

This work was supported by the following scholarships:
Australian Postgraduate Award (2010 - 2011)
Griffith University Postgraduate Research Scholarship (2008 - 2010)
Institute for Integrated and Intelligent Systems Top-Up Scholarship (2008)
Institute for Integrated and Intelligent Systems Postgraduate Research Scholarship (2007 – 2008)

PUBLICATIONS

Publications by the author arising from this work:

- Yatapanage, N., Winter, K., & Zafar, S. (2010). Slicing Behavior Tree models for verification. In: C. S. Calude & V. Sassone (Eds.), *Proceedings of the 6th IFIP International Conference on Theoretical Computer Science (TCS2010)*, (pp. 125-139): Vol. 323 of IFIP Advances in Information and Communication Technology, Springer.

Related publications by the author:

- Lindsay, P. A., Yatapanage, N., & Winter, K. (2011). Cut Set Analysis Using Behavior Trees and Model Checking. *Formal Aspects of Computing*, To Appear. (doi: 10.1007/s00165-011-0181-8).

- Grunske, L., Winter, K., Yatapanage, N., Zafar, S., & Lindsay, P. A. (2011). Experience with Fault Injection Experiments for FMEA. *Journal of Software: Practice and Experience, 41*(11), 1233-1258.

- Lindsay, P., Winter, K. and Yatapanage, N. (2010). Safety Assessment Using Behavior Trees and Model Checking, In: *Proceedings of the 8th IEEE International Conference on Software Engineering and Formal Methods (SEFM 2010)*, (pp. 181-190): IEEE Computer Society.

- Grunske, L., Winter, K. and Yatapanage, N. (2008). Defining the Abstract Syntax of Visual Languages with Advanced Graph Grammars - A Case Study Based on Behavior Trees. *Journal of Visual Languages and Computing, 19*(3), 343-379.

- Zafar, S., Colvin, R., Winter, K., Yatapanage, N., & Dromey, R. G. (2007). Early Validation and Verification of a Distributed Role-Based Access Control Model. In: *Proceedings of the 14th Asia-Pacific Software Engineering Conference (APSEC 2007)*, (pp. 430-437), IEEE.

- Wen, L., Lin, K., Colvin, R., Seagrott, J., Yatapanage, N., & Dromey, R. G. (2007). "Integrare" - A Collaborative Environment for Behavior-Oriented Design. In: *Proceedings of the 4th International Conference on Cooperative Design, Visualisation and Engineering (CDVE 2007)*, (pp. 122-131): Vol. 4674 of Lecture Notes in Computer Science, Springer.

- Grunske, L., Lindsay, P., Yatapanage, N. and Winter, K. (2005). An Automated Failure Mode and Effect Analysis based on High-Level Design Specification with Behavior Trees. In: *Proceedings of Integrated Formal Methods: 5th International Conference (IFM 2005)*, (pp. 129-149): Vol. 3771 of Lecture Notes in Computer Science, Springer.

1 INTRODUCTION

Software systems are everywhere. They range from large safety-critical applications, such as medical equipment or aircraft control, to small household items used every day. Designing these systems without any flaws is a challenge. The systems must provide some degree of reliability to the user, giving them the assurance that the functions will work as expected. For safety-critical applications, the guarantee of correctness is even more important. For these reasons it is important to ensure that the systems have been designed correctly.

Correcting software defects after the system has been implemented is significantly more expensive than if the defects are corrected in the specification and design phases. It is therefore preferable to locate as many defects as possible in these early stages of the software process. Informal specifications written in natural language are not ideal for locating such defects, as the text descriptions are often ambiguous and incomplete. The use of formal models solves these problems. Formal methods refers collectively to techniques which have a rigorous mathematical basis. Formal specification languages allow the system description to be specified in a precise unambiguous manner. By specifying the informal natural language requirements as a formal model, defects are much easier to identify.

Techniques such as manual inspections and testing are often applied in practice for locating defects. Even using a formal model, manual inspection is usually a tedious and error-prone task. Although testing often catches many defects, it is not an exhaustive technique, so some errors can still remain undetected. On the other hand, verification techniques, such as model checking, can provide a guarantee that a model satisfies its requirements.

Model checking (Clarke & Emerson, 1982; Quielle & Sifakis, 1982) is an automated technique which exhaustively explores all possible execution traces of the system. This provides an assurance that the system will behave as required under all circumstances. The overall process of model checking is shown in Figure 1. The system specification or design is first translated into a formal model. The requirements or properties which the system is required to fulfill are also expressed mathematically, in the form of a temporal logic formula. The formal model and the temporal logic formula are then given as inputs to the model checker. The model checker automatically searches the state space of the model, to determine whether or not the model satisfies the property in question. If the property does not hold, the model checker returns a counterexample, which is a trace where the violation occurs.

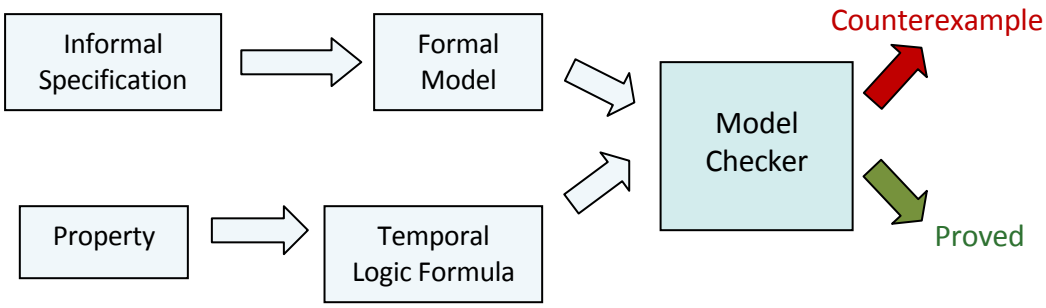


Figure 1. The Model Checking Process

Despite its advantages, model checking has a significant disadvantage: *state explosion*. State explosion refers to the exponential increase in the number of states as the model increases in complexity. The large number of states can cause the model checker to run out of memory resources or take an excessive amount of time to return a result. This can prevent many real-life systems from being model checked.

A number of methods exist for reducing state explosion, such as partial-order reduction (Peled, 1998), Binary Decision Diagrams (McMillan, 1992), abstraction (see for example Dams et al. (1997) and Clarke et al. (2001)) and slicing (Weiser, 1984). These can be divided into two classes: techniques which operate on the internal data structures of the model checker and techniques which reduce the model prior to running the model checker. Internal techniques, such as Binary Decision Diagrams and partial-order reduction, can provide significant reductions in the size of the state space. However, for large systems, the reductions provided may still be inadequate to allow model checking to be performed. The solution is to employ further reduction techniques in conjunction with the internal reduction methods. Techniques such as abstraction and slicing can effectively reduce the size and complexity of the model *prior* to sending it to the model checker.

Abstraction and slicing both produce smaller models; abstraction by representing several variables as a single abstract variable, and slicing by eliminating irrelevant portions of the model. The main difference is that an abstract model is always either an over-approximation or an under-approximation of the original model, so it is not both *sound* and *complete*, whereas the model produced by slicing, known as the *slice*, is both sound *and* complete. Over-approximations can produce counterexamples that are not valid traces of the original model, while under-approximations can fail to discover counterexamples that exist in the original. In contrast, a property holds on the slice if and only if it holds on the original model.

Another advantage is that slicing algorithms are computationally inexpensive. There is therefore no disadvantage in using slicing. If slicing is able to reduce the model, it will reduce the burden of the model checker. In the worst case, the slice returned will be the same size as the original model. Due to its low computational complexity, slicing can be used as a complementary method to the other approaches for reducing state explosion.

The aim of this thesis is to use slicing techniques to reduce specifications prior to model checking in order to alleviate the state explosion problem, allowing large systems to be verified. The specification language that will be used is *Behavior Trees* (Dromey, 2003, 2005), a language with a formal semantics and a graphical, tree-like notation that is easily understood by industry practitioners. The results from this thesis are applicable for slicing of models in other specification languages as well.

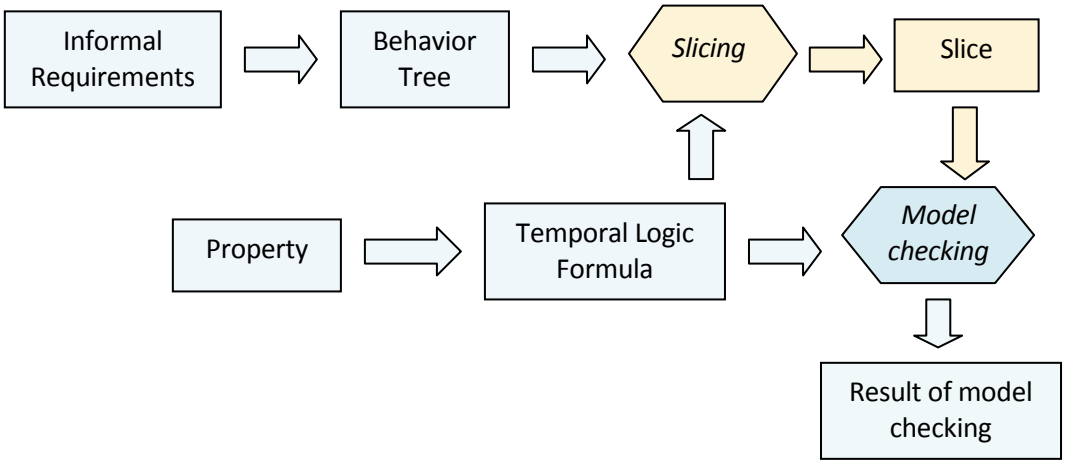


Figure 2. Model Checking Behavior Trees with Slicing

Figure 2 shows the proposed approach for using slicing to aid in the model checking of Behavior Trees. The informal requirements must first be translated into a Behavior Tree using the normal Behavior Engineering methodology. The property to be verified must be expressed as a temporal logic formula. Then, a slice is produced using the Behavior Tree and the temporal logic formula as inputs to

the slicer. The slice and the temporal logic formula are then given to the model checker, which then either states that the property is proved or returns a counterexample.

Behavior Trees

Behavior Trees is the specification language belonging to the Behavior Engineering methodology, proposed by Dromey (2003). Behavior Engineering is an approach for producing a formal specification out of informal, natural language requirements ("Behavior Engineering"). Behavior Engineering advocates a simple approach for creating a Behavior Tree out of requirements. Each individual requirement is modelled separately and the results are then merged together. This process assists users with managing large sets of requirements. The final result is a formal model which maintains traceability to the original requirements. Behavior Engineering has the following benefits (Dromey, 2003, 2005):

- The incremental approach for requirements translation enables users to handle the complexity of large systems by reducing the burden on their short-term memory.
- The approach effectively “bridges the gap between requirements and design” (Dromey, 2003, 2005), by providing a rigorous translation method.
- Behavior Trees maintain traceability to the original requirements by annotating the model with the identifiers of the requirements. This is useful to ensure that later stages of the design continue to preserve the original intentions of the requirements.
- The graphical notation enables users with no mathematical or formal methods background to create Behavior Trees with little difficulty. This is evident by the recent interest in Behavior Engineering by industry practitioners ("Representing Complex Systems," 2008).
- The output is a formal model, because Behavior Trees have a formal semantics (Colvin & Hayes, 2011).

Behavior Trees can be automatically translated into model checking languages for verification (Grunske, et al., 2008). This process has been used successfully for verifying several case studies (Grunske, et al., 2011). However, it inevitably cannot escape the state explosion problem. For large Behavior Tree specifications, the model checker often takes too long or runs out of memory and is unable to return a result. Zafar (2008) attempted to model check a case study of a hospital information system. He reported that after running for more than 270 hours, the model checker ran out of memory without providing a result. He finally resorted to reducing the model using his own knowledge of the system and was able to obtain a result. However, since the reduction was performed manually and not according to a defined approach, he had no guarantee that the results obtained using the reduced model applied to the original model. Examples such as this demonstrate the importance of a correct, automated approach for reducing Behavior Tree models.

1.1 Thesis Objectives

Since the Behavior Trees of most realistic systems are too large for the model checker to handle, users will be unable to use the benefits of Behavior Engineering for large systems. The primary goal of this thesis is to fill this gap in the Behavior Engineering methodology, by proposing the use of program slicing techniques to automatically reduce Behavior Tree specifications, for reducing the time and memory resources required by the model checker.

In addition to this, another objective of this thesis is to provide new slicing methods that are applicable for slicing of any programming or specification language. This includes an optimisation technique for obtaining further reductions to the slice. As well as this, the thesis describes a method for creating slices that are capable of preserving the *next* operator. Additionally, the thesis proposes a novel form of *branching bisimulation* under which the temporal logic CTL* is preserved, which includes the *next* operator. This is a useful theoretical result.

The thesis begins, in **Chapter 2**, with an introduction to the concepts required to understand the thesis topic: temporal logic, existing slicing methods, bisimulation and the syntax and semantics of Behavior Trees.

Program slicing techniques cannot be directly applied to Behavior Trees, due to the differences in semantics and structure between programs and Behavior Trees. For this reason, a new approach for slicing Behavior Trees has been developed and is presented in this thesis, in **Chapter 3**. The slicing approach utilises both adaptations of existing slicing techniques used in programs, as well as new methods for handling constructs that are specific to Behavior Trees. For this purpose, concepts used in program slicing, such as the *control flow graph*, the graph that represents the control flow in a program, must be adapted for Behavior Trees. A new form of control flow graph has been proposed, which incorporates concepts such as *alternative branching*, a branching construct unique to Behavior Trees. Slicing operates by identifying dependencies between the nodes of the model. The proposed slicing approach utilises adaptations of dependency types normally used in program slicing, such as control and data dependence, as well as new dependence types for synchronisation, communication using message passing and termination of threads.

Behavior Trees contain nodes which divert the control flow to other locations, known as *reversions* and *references*. A technique for removing unnecessary nodes of these types is presented. The technique ensures that the resulting slice will still preserve all necessary loops but will not contain unnecessary ones. Furthermore, this thesis presents a technique for merging the nodes of the slice into a syntactically correct Behavior Tree.

Polynomial-time algorithms for producing the slice have been developed, to ensure that the time for producing slices is kept to a minimum.

The slicing approach is only of use if the slices produce identical verification results to the original models. To confirm that this is the case, a proof of correctness is presented, which relates the slice to the original model using *branching bisimulation with explicit divergence*, a form of weak bisimulation that is known to preserve properties specified in the logic CTL^*_{-X}.

Chapter 4 presents an optimisation technique for reducing the model further, by removing *infeasible paths*. This technique is an improvement of an existing approach used for slicing programs. The approach given in this thesis can effectively produce smaller slices than the previous approach. As well as being useful for Behavior Trees, this optimisation technique is applicable to all forms of slicing, including slicing of programs. A proof of correctness using bisimulation is provided, to ensure that the reductions do not change the verification outcome.

All previous slicing approaches designed for verification are unable to preserve properties containing the temporal logic operator X. In **Chapter 5**, a novel technique is proposed, which produces slices that preserve full CTL^*, including formulas containing the operator X. This result is shown by the proposal of a new form of branching bisimulation, known as *next-preserving branching bisimulation*. A proof has been provided to show that next-preserving branching bisimulation preserves full CTL^*. This result is an essential contribution, as the weak forms of bisimulation are normally unable to preserve properties containing the X operator. Thus, these results are useful for many applications.

To confirm that the theoretical results of the previous chapters are applicable in practice, the results must be demonstrated on case studies. In **Chapter 6**, the Behavior Tree slicing approach is demonstrated on two case studies: a mine pump and a hospital information system. Both are case studies which originally could not be model checked in a reasonable amount of time. As will be seen, slicing was able to significantly reduce the model checking time, thus allowing verification results to be obtained for both case studies.

Finally, **Chapter 7** concludes the thesis and provides directions for future research.

The techniques given in each of the chapters can be composed together in various ways, according to the user's preference. After obtaining a slice using the general Behavior Tree slicing approach, this can be used directly for model checking. Alternatively, an optimised slice can be developed from it, using the approach given for removing infeasible paths, or the slice can be transformed into one that preserves the *next* operator. Another option is to compose all three techniques to create an optimised slice that preserves all formulas.

1.2 Related Work

Program slicing was originally developed by Weiser (1984) for aiding programmers in the debugging and understanding of large systems. There has been recent interest in applying slicing techniques for model reduction prior to verification.

This thesis presents the first approach for slicing Behavior Tree models. Previous attempts at reducing Behavior Tree models have involved manual changes made according to the user's knowledge of the system (Zafar, 2008). In contrast, the approach given in this thesis is fully automatic and the slice is guaranteed to preserve the same set of properties as the original.

The closest related work are approaches which slice specifications written in other languages for verification. Compared to slicing programs for debugging and understanding, slicing models for verification is a relatively new topic. Nevertheless, there have been approaches proposed by several groups for different specification languages. Millet and Teitelbaum (2000) sliced Promela models, which is the input language of the SPIN model checker. However, their approach was not guaranteed to preserve global properties, as they felt that property preservation was not essential for obtaining useful results. In a similar manner, Ganesh (1999) proposed a technique for slicing SAL models, the input language of the SAL model checking framework and Thrane and Sorensen (2008) gave a technique for slicing models for the UPPAAL model checker. Of these, Ganesh did not use temporal logic theorems for the slicing criterion, instead using the input and output variables in the model. Neither proved the correctness of their approach. Sabouri and Sirjani (2010) sliced Rebeca models, which is an actor-based specification language, although they did not prove the correctness of the approach either.

Brückner and Wehrheim (2005b) sliced Object-Z for verification and later extended the approach to CSP-OZ (Brückner & Wehrheim, 2005a), a language that combines Communicating Sequential Processes (CSP) and Object-Z. The approach was then extended to CSP-OZ-DC, a combination of CSP, Object-Z and Duration Calculus (Brückner, 2007). In their approach, the criterion contains the nodes that influence the events and states found in the theorem, expressed in Duration Calculus. This was one of the few approaches which included proofs of correctness. Similarly, Bordini et al. (2009) sliced agent-based systems written in the AgentSpeak language and proved that their approach preserved LTL_{-X} using stuttering equivalence.

Several other authors proposed slicing approaches for reducing state explosion but did not provide full proofs of correctness. Odenbrett et al. (2010) presented an approach for slicing AADL (Architecture Analysis and Design Language) specifications in order to reduce them prior to translation into Promela, the input language of the SPIN model checker. They claimed that CTL^{*}_{-X} properties were preserved, but left the proof for future work. Similarly, Schaefer and Poetzcsh-Heffter (2008) sliced specifications of adaptive systems as part of the MARS framework. They used *consistent bisimulation* to show that the approach preserves a variant of CTL* that does not contain the U or X operators, but details of the proof were not provided. Finally, van Langenhove and Hoojewijs (2006) presented an approach for slicing UML models for verification. The approach was claimed to preserve LTL_{-X} properties, although again a full proof was not given.

These are all approaches for slicing specifications in order to alleviate the state explosion problem. The technique presented in this thesis is the first which slices Behavior Tree models for this purpose. Since each slicing approach involves dependency types specific to the particular language, none of the other approaches can be directly applied to Behavior Trees. Furthermore, no other approach is able to preserve properties containing the X operator.

2 BACKGROUND

This chapter provides the background material for this thesis. Section 2.1 introduces model checking, temporal logic and bisimulation. Section 2.2 explains the concepts of program slicing, as well as existing approaches for slicing specifications and for model reduction in the context of verification. Section 2.3 introduces the Behavior Engineering methodology, including the Behavior Tree notation and the process of translating Behavior Trees to a model checking language.

2.1 Model Checking

Model checking is an automated verification technique, developed independently in the early 1980's by both Clarke and Emerson (1982) and Quielle and Sifakis (1982). The model checker takes two inputs: a model representing the system to be verified and a property to be checked. The model checker then systematically searches the state space of the model to determine whether or not the property is satisfied. If the property is not satisfied, the model checker returns a counterexample, which is a behaviour of the system that violates the property. The counterexample aids the user in determining the reason why the property does not hold on the model.

Each model checker has its own input language for specifying the model, usually describing the model as a *transition system*. The model describes the behaviour of the system. The property to be verified is a requirement which must hold on the model, for example a safety requirement. It is usually specified as a temporal logic formula.

2.1.1 Transition Systems

Transition systems describe the behaviour of systems. The behaviour is described in terms of states and transitions. Each system contains a set of atomic propositions. The states are differentiated by the atomic propositions which hold in each state. The transitions allow the system to evolve from one state to another. A transition system, otherwise known as a *Kripke Structure*, is a tuple $T = (\mathcal{S}, AP, \mathcal{L}, \mathcal{I}, \longrightarrow)$, where $\mathcal{S}$ is a set of states, AP is a set of atomic propositions, $\mathcal{L}$ is a labelling function which labels each state with the set of atomic propositions that hold in that state, $\mathcal{I}$ is a set of initial states and $\longrightarrow \subseteq \mathcal{S} \times \mathcal{S}$ is the transition relation. From a particular state, there may be multiple transitions emanating from it. In this case, the transition to execute next is chosen non-deterministically from the set of available transitions. An example of a simple transition system is shown in Figure 3.

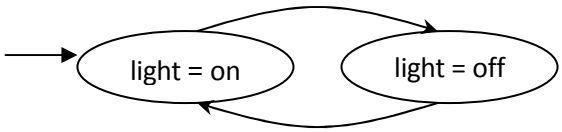


Figure 3. An Example Transition System

A *path* in a transition system is a sequence of states $\pi = < s_0, s_1, s_2, ... >$, where for every s_i, $s_{i+1} \in \pi$, where $i \geq 0$, $s_i \longrightarrow s_{i+1}$. Each path is either *finite* or *infinite*. A finite path is one which terminates at

some state. An infinite path does not terminate. A path is said to be *maximal* if it is either infinite or it ends at a state with no outgoing transitions. The suffix of π starting at the state s_j is denoted by $\pi[s_j\rangle\!\rangle$ and the prefix of π ending at s_j is denoted by $\pi\langle\!\langle s_j]$.

The goal of model checking is to determine whether or not the following holds, where $T = (\mathcal{S}, AP, \mathcal{L}, \mathcal{I}, \longrightarrow)$ is a transition system and φ is a temporal logic theorem:

$$T \vDash \varphi,$$

which represents that $\forall s_0 \in \mathcal{I}, s_0 \vDash \varphi$.

Another type of transition systems is known as *labelled transition systems*. These are the action-based counterparts of transition systems. A *labelled transition system* is a tuple $(\mathcal{S}, \mathcal{I}, A, \rightarrow)$, where $\mathcal{S}$ is a set of states, $\mathcal{I}$ is the set of initial states, A is a set of actions and $\rightarrow \subseteq \mathcal{S} \times A \times \mathcal{S}$ is the transition relation. The notation $s \xrightarrow{a} s'$ is used as a shorthand for $(s, a, s') \in \rightarrow$.

de Nicola and Vaandrager (1995) proposed *doubly-labelled transition systems*, which contain the information of both transition systems and labelled transition systems. A doubly-labelled transition system has both a labelling on states and a labelling on transitions and is defined as a tuple $(\mathcal{S}, AP, \mathcal{I}, \mathcal{L}, A, \longrightarrow)$, where $\mathcal{S}$ is a set of states, AP is a set of atomic propositions, $\mathcal{I}$ is the set of initial states, $\mathcal{L}: \mathcal{S} \rightarrow 2^{AP}$ is a labelling function which labels each state with the set of atomic propositions that hold in that state, A is a set of actions and $\longrightarrow \subseteq \mathcal{S} \times A \times \mathcal{S}$ is the transition relation. Doubly-labelled transition systems are useful for specifying systems that contain both state and event-based information and allow to translate from one to the other. For example, they were used in (ter Beek, et al., 2011) for modelling the state and event-based aspects of UML state machines.

2.1.2 Temporal Logic

The most commonly used temporal logics are Computation Tree Logic (CTL) (Clarke and Emerson (1982) and Linear Temporal Logic (LTL) (Pnueli, 1977). Both are subsets of the logic CTL* (Clarke et al. 1986).

CTL* is a logic which uses a branching time model. Branching time reflects the fact that many different paths are possible starting from any given state in the transition system, due to states having more than one outgoing transition. In CTL*, properties are expressed in terms of *state formulas* and *path formulas*. State formulas specify properties of states, while path formulas describe properties which hold on paths. The following definition gives the syntax of CTL* formulas. These definitions have been taken from Baier and Katoen (2008).

DEFINITION 1. CTL* SYNTAX

A CTL* *state formula* ψ is defined as follows, where $\mathsf{p} \in AP$ is an atomic proposition, ψ_1 and ψ_2 are state formulas and φ is a *path* formula:

$$\psi = true \mid \mathsf{p} \mid \psi_1 \wedge \psi_2 \mid \neg \psi_1 \mid \mathsf{E}\varphi$$

A CTL* *path formula* is defined as follows, where φ_1 and φ_2 are path formulas and ψ is a state formula:

$$\varphi = \psi \mid \varphi_1 \wedge \varphi_2 \mid \neg \varphi_1 \mid \mathsf{X}\varphi_1 \mid \varphi_1 \mathsf{U} \varphi_2$$

■

A state formula can be either the value *true*, an atomic proposition (p), the conjunction of two state formulas ($\psi_1 \wedge \psi_2$), the negation of a state formula ($\neg \psi_1$) or the existence of a path satisfying a particular path formula ($\mathsf{E}\varphi$). The operator E represents that "there exists" a path on which the given path formula holds. In addition, a derived operator A is often used, which represents that the given path formula holds "for all" paths. The operator A is derived from E as follows: $\mathsf{A}\varphi \equiv \neg \mathsf{E} \neg \varphi$. In a similar manner, the disjunction operator can be derived from conjunction: $\psi_1 \vee \psi_2 \equiv \neg (\neg \psi_1 \wedge \neg \psi_2)$.

A path formula can be either a state formula (ψ), the conjunction of two path formulas ($\varphi_1 \wedge \varphi_2$), the negation of two path formulas ($\neg \varphi_1$), the *next* operator ($\mathsf{X}\varphi_1$) or the *until* operator ($\varphi_1 \mathsf{U} \varphi_2$). The *next* operator specifies that the given path formula must hold at the next state on the path. The *until* operator is used to specify that a given path formula must hold until another given path formula holds, i.e. $\varphi_1 \mathsf{U} \varphi_2$ specifies that φ_1 holds until a state is reached where φ_2 holds. Additionally, there is a requirement that φ_2 does eventually hold. Two derived operators, F and G, are often used. The F operator specifies that a given path formula must eventually hold, at some state in the future along the path. F is derived from U as follows: $\mathsf{F}\varphi \equiv \mathit{true}\ \mathsf{U}\ \varphi$. The G operator specifies properties which must hold on *all* states along the path. G is derived from F as follows: $\mathsf{G}\varphi \equiv \neg\, \mathsf{F}(\neg\varphi)$. As was done for state formulas, the disjunction operator can be derived from conjunction.

The following definition gives the semantics for CTL* formulas, which explains under what circumstances a state satisfies each type of formula.

DEFINITION 2. CTL* SEMANTICS

Let $T = (\mathcal{S}, AP, \mathcal{L}, \mathcal{I}, \rightarrow)$ be a transition system. A CTL* state formula ψ holds in a state $s \in \mathcal{S}$, denoted $T, s \vDash \psi$, or simply $s \vDash \psi$, according to the following, where ψ_1 and ψ_2 are CTL* state formulas and φ is a CTL* path formula:

$s \vDash \mathit{true}$,

$s \vDash a \in AP$ iff $a \in \mathcal{L}(s)$,

$s \vDash \neg\, \psi_1$ iff $s \nvDash \psi_1$,

$s \vDash \psi_1 \wedge \psi_2$ iff $s \vDash \psi_1$ and $s \vDash \psi_2$,

$s \vDash \mathsf{E}\varphi$ iff there exists a path $\pi = < s_0, s_1, s_2, \ldots >$, such that $s_0 = s$ and $\pi \vDash \varphi$.

A CTL* path formula φ holds for a path $\pi = < s_0, s_1, s_2, \ldots >$, denoted $\pi \vDash \varphi$, according to the following, where φ_1 and φ_2 are CTL* path formulas and ψ_1 is a CTL* state formula:

$\pi \vDash \psi_1$ iff $s_0 \vDash \psi_1$,

$\pi \vDash \varphi_1 \wedge \varphi_2$ iff $\pi \vDash \varphi_1$ and $\pi \vDash \varphi_2$,

$\pi \vDash \neg\, \varphi_1$ iff $\pi \nvDash \varphi_1$,

$\pi \vDash \mathsf{X}\varphi_1$ iff $\pi[s_1\rangle\!\rangle \vDash \varphi_1$,

$\pi \vDash \varphi_1 \mathsf{U} \varphi_2$ iff $\exists j > 0$ such that $\pi[s_j\rangle\!\rangle \vDash \varphi_2$ and $\forall i$, where $0 \leq i < j$, $\pi[s_i\rangle\!\rangle \vDash \varphi_1$.

■

CTL is a subset of CTL* in which every path operator (X, U, F and G) must be immediately preceded by one of the state operators A or E. Thus, some CTL* properties are not expressible in CTL, such as E(Xp ∧ XXq). LTL is a subset of CTL* which does not include the E operator. All properties are implicitly expressed over all paths. In LTL it is not possible to specify that *there exists* a path where a property holds. Even so, there are LTL properties which cannot be expressed in CTL, such as FGp, which models fairness. The relationship between the three logics is shown in Figure 4. CTL*$_{-X}$, LTL$_{-X}$ and CTL$_{-X}$ refer to the variants of CTL*, LTL and CTL, respectively, that do not allow the use of the X operator.

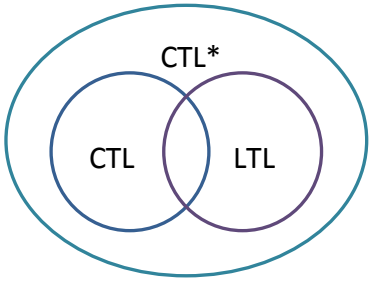


Figure 4. Relationship Between CTL*, CTL and LTL

2.2 Program Slicing

Slicing is an automated program reduction technique, which was originally proposed by Weiser (1981, 1984) for aiding developers in debugging and understanding large programs. Since then, program slicing has been investigated extensively (see Tip (1995) and Xu, et al. (2005) for comprehensive surveys), exploring its use for debugging as Weiser originally intended, as well as for new applications, including testing. The premise behind program slicing is simple: the program is reduced by automatically eliminating program statements that are irrelevant according to some specified slicing criterion, which traditionally consists of a program statement and a set of variables. The goal is to find the statements which are necessary in order to determine the values of those variables at the given point in the program.

Weiser's original approach (1981, 1984) involved solving dataflow equations in order to compute the set of relevant statements that form the slice. Most recent approaches instead use a *Program Dependence Graph* (PDG), first proposed by Ottenstein and Ottenstein (1984), which is a graph representing the dependencies between the program statements. The slice is computed by performing a backwards traversal of the graph, starting at the point of interest. Figure 5 gives an overview of the usual slicing approach for programs. The first step is to construct a *Control Flow Graph* (CFG) of the program, which is a graph showing the control flow between the statements of the program. This CFG is then used to create the PDG. A slicing criterion is supplied, commonly in the form $c = \{s, v\}$, where s is a statement in the program and v is a set of variables of interest.

Next, the node in the PDG which represents the program statement from the slicing criterion is identified. This node is used as the starting point for a backwards traversal of the PDG. The nodes which are encountered during the traversal form the slice; all other nodes are discarded.

Slicing is computationally inexpensive. Each phase of the slicing process can be achieved in polynomial time (Reps, et al., 1994). Additionally, the construction of the CFG and PDG need only be performed once per program. The construction of a new slice only requires a new traversal of the existing PDG, starting from a different node based on the new criterion.

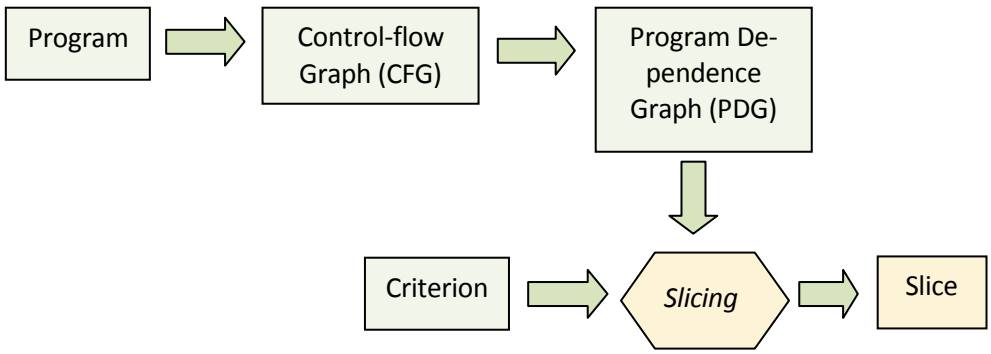


Figure 5. Overview of Program Slicing

2.2.1 Slicing of Sequential Programs

The first step of slicing is to create the Control Flow Graph (CFG) of the program. A CFG is a directed graph $G = < N, E, start, end >$, where N is a set of nodes, each representing a statement in the program, E is a set of edges representing the flow of control, such that $E = N \times N$, *start* is the node representing the start of the program and *end* is the termination node. An edge $e \in E$, where $e = (n_1, n_2)$, indicates that n_2 is one of the nodes which can execute immediately after n_1. The node n_2 is known as an *immediate successor* of n_1. Every node (except the end node) in the CFG has either one or two immediate successors. Sequential flow results in one successor, while branching conditions, such as *if* statements, result in two successors, one representing the case where the condition is true and the other where it is false. For this reason, the edges of a CFG may additionally be associated with a label indicating whether it is the *true* or *false* branch. The function *label*(e) returns the label associ-

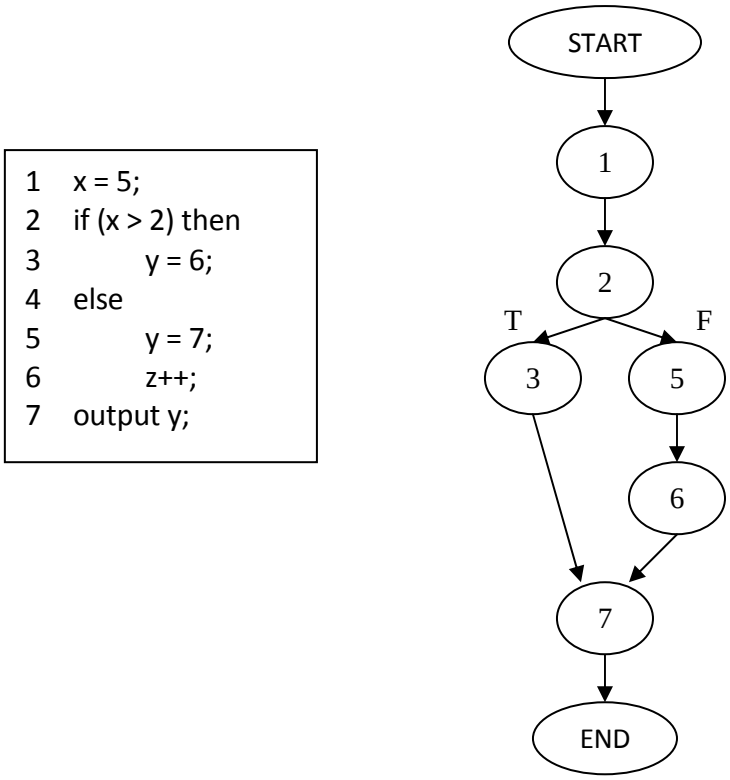


Figure 6. A Simple Program and its Corresponding CFG.

ated with the edge *e*. A *path* in a CFG consists of a sequence of nodes $< n_0, n_1, ..., n_k >$, where for every n_i, such that $0 \leq i < k$, $(n_i, n_{i+1}) \in E$. For every node *n* in the graph, there exists a path from *start* to *n* and a trace from *n* to *end*. The path from a node *m* to a node *n* is given by *path*(*m*, *n*). An example of a simple code fragment and the corresponding CFG is shown in Figure 6. Each node in the graph represents a statement of the program, as indicated by the numbers. The *if* statement at line 2 results in a branch in the graph. Both branches converge to node 7.

The Program Dependence Graph (PDG) (Ottenstein & Ottenstein, 1984) is constructed by identifying dependencies between program statements using the CFG. The PDG is also a directed graph $G = < N, E >$, where *N* is a set of nodes, each representing a statement in the program and *E* is a set of edges such that $E = N \times N$. The difference is that the edges represent a dependency between two nodes, instead of showing control flow. Unlike the CFG, there are no *start* or *end* nodes in a PDG. An edge $e \in E$, where $e = (n_1, n_2)$, indicates that n_2 is *dependent* on n_1. This means n_2 requires n_1 in order to execute. The two main types of dependencies in programs are control and data dependence.

Control Dependence

A control dependence from node n_1 to n_2 indicates that n_1 controls whether or not n_2 will be executed. There are numerous forms of control dependence in the literature. Many of these are essentially the same, despite having different names (Chen & Roşu, 2006). Control dependence definitions vary according to concepts such as how non-termination is handled. The traditional form of control dependence, as described by Tip (1995), is defined in terms of the notion of *post-dominance*. A node *n* is said to *post-dominate* a node *m* iff every path from *m* to *end* passes through *n*. The *immediate post-dominator* of *m* is the closest post-dominator. Using these concepts, control dependence is defined as follows.

DEFINITION 3. CONTROL DEPENDENCE
Node *n* is control-dependent on node *m* iff:

- $\exists \pi = path(m, n)$, where $\forall p \in \pi - \{m, n\}$, *p* is post-dominated by *n* and

- *m* is not post-dominated by *n*. ■

The first condition of the definition states that there is a path from *m* to *n* on which *n* always executes. The second condition states that there is at least one other path from *m* on which *n* is never reached. The node *m* thereby *controls* whether or not *n* can execute, by making the decision as to whether or not *n* will be bypassed.

Other variations of this definition exist which are essentially the same, for example the definition of *strong control dependence* given by Podgurski and Clarke (1990), which states that a node *n* is control-dependent on a node *m* if there is a path from *m* to *n* which does not contain the immediate post-dominator of *m*.

Example.

Control dependence can be illustrated with the example in Figure 6. Let *m* be node 2 and *n* be node 6. On the path from 2 to 6, node 5 is post-dominated by node 6, since every path from node 5 to *end* passes through node 6. This satisfies the first condition of Definition 3. Node 2 is not strictly post-dominated by node 6, due to the other path through node 3, thereby satisfying the second condition. Thus, node 6 is control-dependent on node 2.

The definition of strong control dependence produces the same result. Node 7 is the immediate post-dominator of node 2. There is a path from node 2 to node 6 which does not contain node 7, so node 6 is strongly-control-dependent on node 2.

■

Data Dependence

Data dependencies exist when one node modifies the state of a variable which is referenced by another node. Let $DEF(n)$ return the set of variables which are updated at node *n*, for example variables which are set to a new value. Let $REF(n)$ return the set of variables which are referenced at node *n*. An example is when an *If* statement consists of a guard which queries the state of a variable. The definition states that a node *n* is data-dependent on a node *m* if *n* references a variable that is defined at *m* and there exists a path between them on which the variable is not re-defined.

DEFINITION 4. DATA DEPENDENCE

Node *n* is data-dependent on node *m*, iff $\exists\, v$ such that:

- $v \in DEF(m)$,
- $v \in REF(n)$ and
- $\exists\, \pi = path(m, n)$, such that $\forall\, p \in \pi - \{m\}$, $v \notin DEF(p)$.

■

Example.

In the example in Figure 6, node 2 is data-dependent on node 1, because node 2 queries the state of variable x, while node 1 updates the state of variable x. The PDG of the CFG in Figure 6 is given in Figure 7. Every edge corresponds to a dependency between two of the nodes in the CFG.

■

Types of Slicing

The numerous slicing algorithms currently in existence can be classified in several different ways. Most slicing algorithms are either *static* or *dynamic*. For dynamic slicing (Korel & Laski, 1988), the criterion includes the input values for the program, whereas static slicing does not make any assumptions about the input values and thus considers all inputs. Therefore, dynamic slicing can produce smaller slices than static slicing can. This notion was formalised by Binkley, et al. (2006), where they showed that dynamic slicing is a weaker form of static slicing. They further proved the correspondence between weaker and stronger forms of slicing and the size of the slices produced. If a form of slicing is shown to be weaker than another, the smallest possible slices obtainable by the weaker

method (known as the *minimal* slices) will be smaller than the minimal slices of the stronger method. Thus, dynamic slicing produces smaller minimal slices than its static counterpart, as dynamic slicing is weaker. Despite this, for a particular input, the slice produced by dynamic slicing will not always necessarily be smaller than what would have been obtained using static slicing. The results can vary depending on the particular algorithm being used (Binkley, et al., 2006). Another form of slicing, known as *conditioned slicing*, lies between static and dynamic slicing. It is a form of static slicing which restricts the slice to statements which can execute under a specific condition.

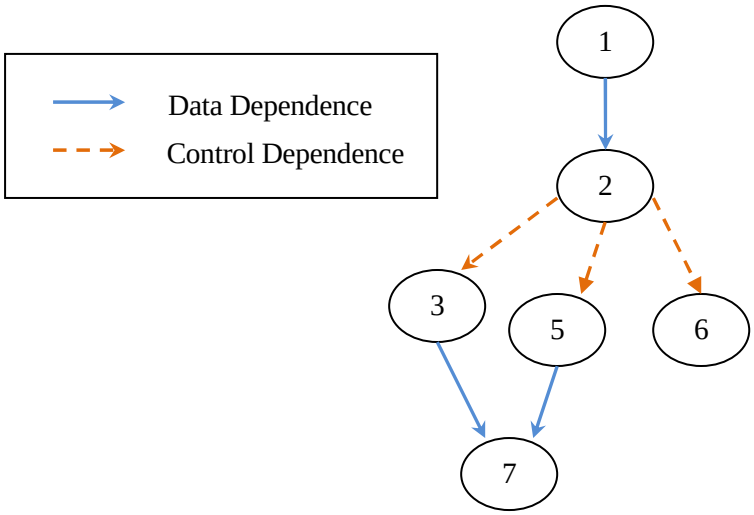


Figure 7. The PDG of the CFG in Figure 6.

Another classification is whether the slicing is conducted in the *forward* or *backward* direction. Forward slicing identifies all the statements which *can be influenced by* the given slicing criterion. Backward slicing identifies the statements which *can influence* the slicing criterion. Weiser's original algorithm is an example of static backward slicing.

2.2.2 Non-termination

Programs often contain loops which may potentially execute infinitely, thereby preventing control flow from reaching any successors beyond the loop. The execution of a subsequent statement *n* is controlled by the guard of the loop, *m*. The traditional definition of control dependence is non-termination insensitive, since the guard *m* may be post-dominated by *n*. That is, every path from *m* eventually leads to *n*, even though it passes through the loop. The possibility that the loop may execute infinitely is not taken into account. As a result, infinite loops such as this will not be identified by the control dependence relation and will therefore not be included in the slice. The resultant slice would exhibit different behaviour to the original program, because the trace in which the loop executes infinitely often will not be present in the slice.

Podgurski and Clarke (1990) defined *weak control dependence*, in order to incorporate then notion of non-termination. They used another form of post-dominance, known as *strong post-dominance*.[*] Recall that a node *n* post-dominates a node *m* iff every path from *m* to the end node must pass through *n*. A node *n strongly post-dominates* a node *m* iff there exists an integer $k \geq 1$, such that every path from *m* of length at least *k* must pass through *n*. The difference between the two forms of control dependence is only apparent in the presence of potentially infinite loops. Weak control dependence is non-termination sensitive. It therefore results in a greater number of control dependencies identified when computing the transitive closure of the CFG. Weak control dependence is defined as follows.

[*] Podgurski and Clarke referred to post-dominance and strong post-dominance as *forward dominance* and *strong forward dominance*, respectively.

DEFINITION 5. PODGURSKI & CLARKE'S WEAK CONTROL DEPENDENCE[†]
Node *n* is *weakly-control-dependent* on node *m*, iff *m* has two successors *p* and *q*, such that *n* strongly post-dominates *p* but does not strongly post-dominate *q*. ■

Chen and Roşu (2006) proposed a new form of control dependence which they called *termination sensitive control dependence*. It is designed to handle non-terminating loops by utilising termination information about each loop, given as annotations by the user. This form of control dependence lies between strong and weak control dependence, coinciding with the strong form when all loops are annotated as terminating and with the weak form when all loops are labelled non-terminating.

All of the definitions above require the control flow graph to have a single end state, a concept known as the *unique end node property* (Ranganath, et al., 2007). However, this is unsuitable for many modern programs, which often have multiple end points or no end points at all. This is often the case for reactive systems which infinitely cycle as they wait to receive input. One proposed solution is to modify the program in such a way as to ensure that it has a unique end node. Nevertheless, this solution is impractical for many modern program structures. Ranganath et al. (2007) defined new forms of both weak and strong control dependence which are suitable for systems with no unique end node. They defined control dependence over *maximal paths*, instead of paths which terminate at the end node. Similarly to maximal paths of transition systems, a maximal path of a control flow graph is a path that either terminates or contains an infinite loop. The definition for Ranganath et al.'s non-termination sensitive control dependence is given below.

DEFINITION 6. NON-TERMINATION SENSITIVE CONTROL DEPENDENCE
A node *n* is *non-termination sensitive control dependent* on a node *m* iff *m* has at least two successors *p* and *q* such that:

- for all maximal paths π_1 from *p*, node *n* always occurs and either *m* = *n* or *n* strictly precedes any occurrence of *m* in π_1 and
- there exists a maximal path π_2 from *q* on which either *n* does not occur or *m* strictly precedes any occurrence of *n* in π_2. ■

This new form of control dependence captures the same idea behind the traditional form of control dependence: a node *m* controls a node *n* if *m* leads to two branches, one which leads to *n* and one which causes *n* to be bypassed. The extra conditions in this new definition are for handling paths which do not necessarily terminate at a given end node, but instead revert and reach *m* again.

There are further variations on control dependence, such as *weak-order dependence*, proposed by Amtoft (2008), which identifies nodes that control the *order* in which nodes are executed, not just whether or not they are executed.

2.2.3 Slicing Concurrent Programs

The original slicing algorithms were designed for sequential programs. Concurrency presents significant challenges due to the complex interactions between threads. Cheng (1993) was the first to consider slicing of programs with concurrently executing threads. Cheng used Program Dependence Nets, which is an extension of Program Dependence Graphs. Cheng identified dependency types that are associated only with concurrent processes. Zhao (1999) proposed a similar approach, also based on PDG's, using multi-threaded dependence graphs for slicing concurrent Java programs. As for Cheng's approach, Zhao's graphs represented additional types of dependencies, which arise due to the interactions between concurrently executing threads. Zhao focussed on communication dependence and synchronisation dependence.

The techniques of Cheng and Zhao were both based on a graph reachability problem and therefore assumed transitivity for all dependence edges in the graph. This assumption is correct for control and

[†] Podgurski and Clarke used the term *direct weak control dependence* instead of weak control dependence and used the term *weak control dependence* to refer to the transitive closure of direct weak control dependence.

data dependence, as these operate on sequential paths. However, data dependence between parallel threads is intransitive, so this assumption can lead to inaccurate slices (Krinke, 1998).

Krinke differentiated between data dependence, which occurs between nodes in a single thread, and *interference dependence*, which occurs between nodes in parallel threads. He proposed a solution to handle the intransitivity of interference dependence based on the notion of a *threaded witness* in a threaded version of the CFG. A threaded witness is a sequence of statements which form a possible execution path of the program. Statements are not included in a slice if they do not form a threaded witness of the program.

Krinke's algorithm for computing threaded witnesses makes use of a tuple which records the last visited node for each thread. When a new node is reached in the backwards exploration of the dependency graph, the algorithm checks whether there is a valid path from the new node to the last visited node in that thread. If not, the new node is not included in the slice because it does not form a threaded witness. If there is a path, the new node is recorded in the tuple as the last encountered node for that thread.

Nanda and Ramesh (2000, 2006) showed the algorithm to be imprecise in the case of nested threads. Consider the example CFG shown in Figure 8. The CFG shown is a threaded CFG in the style used by both Krinke (1998) and Nanda and Ramesh (2000, 2006). It begins with an initial thread θ_0, which then spawns two new threads θ_1 and θ_2. After executing some behaviour, these two threads exit and the control flow converges back to θ_0. Threads θ_1 and θ_2 are known as nested threads. Since each thread is considered separately when computing the threaded witness, the algorithm maintains a tuple containing the last visited node for each of the three threads. Assume that the algorithm has just explored a node n_1 in θ_0. The tuple would now be $[n_1, \perp, \perp]$, where $\perp$ represents that no node has been encountered for that thread so far. If n_1 was interference-dependent on a node n_2 in θ_2, then this node will be included in the slice, because no node has so far been visited in that thread. This is imprecise because n_1 can never depend on n_2 which executes after it. The solution proposed by Nanda and Ramesh (2000, 2006) is to store information on the last node traversed in each set of sequentially operating threads, instead of in each individual thread. When a node is encountered, its label is not only recorded as the last node traversed in its own thread, but also in its parent and child threads. In the example, n_1 would be recorded in the place for thread θ_0, as well as for θ_2, i.e. the tuple would be $[n_1, \perp, n_1]$. Then when n_2 is explored, it will not be included in the slice because n_1 cannot be reached from n_2.

Nanda and Ramesh (2006) also showed Krinke's approach to be imprecise when handling threads in nested loops. Assume there is a node n_x with an ancestor n_y. Further assume that n_x is dependent on another node n_z. There are cases in which n_z cannot influence n_x because its update is always overidden by the ancestor n_y. In these cases, including n_z would decrease the precision of the slice.

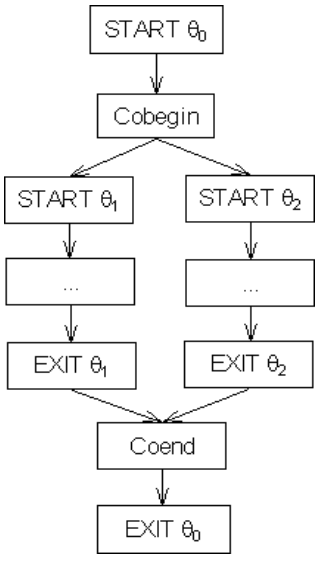


Figure 8. Example CFG of a Java Program, using the Threaded CFG style of Krinke (1998) and Nanda and Ramesh (2000, 2006).

Further reductions were suggested by Ranganath and Hatcliff (2004), who proposed a technique to identify interference edges which can be removed as they correspond to objects which are only ever accessed by a single thread. Since interference dependence occurs when an object is written in one thread and read by another, objects which are only accessed by a single thread cannot generate interference dependencies. This allows the number of interference dependence edges in the PDG to be reduced.

2.2.4 Program Slicing vs. Slicing of Models

Program slicing has been investigated extensively. In comparison, slicing of specifications has received less attention. The earliest attempt at slicing specifications was that of Oda and Araki (1993), for slicing Z specifications. Their purpose was to improve the understanding of large specifications. Wu and Yi (2004) later proposed another approach for slicing Z specifications using PDG's. Other similar approaches include slicing of UML models (Lano & Kolahdouz-Rahimi, 2010), CSP (Silva, et al., 2008), use case maps (Hassine, et al., 2005), hierarchical state machines expressed in the RSML language (Heimdahl & Whalen, 1997), Extended Finite State Machines (Korel, et al., 2003) and statecharts (Luangsodsai & Fox, 2010).

The traditional program slicing techniques can be adapted to suit various languages, including specification and modelling languages. The exact modifications that are required depends on the language of interest. It is usually necessary to consider the underlying semantics of the language to determine the types of dependencies. In some cases, the definitions of control and data dependence for programs can be easily adapted to the modelling language. For example, when slicing Extended Finite State Machine (EFSM) models, Korel et al. (2003) defined control and data dependencies in terms of states and transitions in the model, instead of nodes in the control flow graph. Similarly, Labbé and Gallois (2008) proposed an approach for slicing communicating automata specifications by adapting the control dependence definition of Ranganath et al. (2007). Again, they defined control dependence in terms of transitions in the specification, instead of nodes in the control flow graph. Furthermore, they incorporated Krinke's threaded witness approach (1998) into their definition for data dependence, by including the condition that there must be a valid path in the specification corresponding to each dependency path.

The modelling languages may contain constructs which do not match programming language concepts. In these cases, new dependence definitions may be necessary. Labbé et al. (2007) defined a new type of dependence known as *communication dependence*, in order to handle the communication which may occur between two automata. The *interference control dependence* of Luangsodsai and Fox (2010), used for slicing statecharts, also performs a similar purpose. Interference control dependence occurs when an event in a statechart is triggered by a parallel action. Wu and Yi (2004) introduced a new type of control dependence known as *logic dependence*, in order to represent the dependencies between post-conditions and pre-conditions in Z schemas. Brückner (2007) defined several new types of dependencies for slicing CSP-OZ-DC specifications, including *timing dependence*, the dependence arising from real-time properties specified in the DC part of the specification. In some cases, traditional dependencies may be found to be unnecessary. For example, interference dependency was found to be unnecessary when slicing Rebeca models (2010), as there are no shared variables between parallel objects. As these examples show, the definitions and algorithms for program slicing can be adapted to other types of languages. Each language would require different variations on the traditional program slicing definitions, according to the semantics of the language.

The purpose of performing the slicing is also relevant to how the dependencies should be defined. When slicing is used for debugging or understanding, the slice does not need to be *executable*. An executable slice is one which has the proper syntax of the programming language and so can be executed. When slicing for verification, the slice must be executable, in order for it to be used by the model checker. Another factor which is of more importance for slicing for verification is non-termination. As discussed by Ranganath et al. (2007), if the goal is only to aid the user in understanding large specifications, not for formal verification, then it might not be important to ensure that non-termination will be preserved.

Slicing Criterion

Slicing for reduction of models or programs prior to verification has been classified as *Proposition-Based Slicing* by Silva (2011). The main difference that arises when slicing for verification is the slicing criterion. When slicing for debugging or understanding, the criterion is usually given in the form of a program statement of interest and a set of variables. On the other hand, slicing used in the context of verification uses a criterion based on the temporal logic formula to be verified. This must be converted into the set of nodes from which the traversal of the dependence graph must begin. The approach can thus be classified as *simultaneous static slicing* (Danicic, et al., 1995), because there are several starting nodes for the slicing process. The most comprehensive work on slicing programs for verification is that of the Indus slicer (Dwyer, et al., 2006), which slices Java programs and operates as part of the Bandera model checking framework. The slicer incorporates concepts such as termination sensitive control dependence and interference dependence.

There has been recent interest in slicing models and specifications for verification. Several approaches have been proposed for various different languages, as discussed in Section 1.2. All of the existing approaches for slicing for verification use essentially the same process for deriving the starting nodes from the temporal logic theorem. First, the atomic propositions in the formula are identified. Then, any nodes which directly modify the truth value of one or more of these propositions become the slicing criterion. For example, Van Langenhove and Hoogewijs (2006), when slicing for verification of UML diagrams, considered the slicing criterion to be any states or transitions in the UML model which directly modify a variable in the temporal logic theorem.

This approach is the simplest and least computationally expensive, but it can sometimes lead to unnecessary nodes being included into the slice. Vasudevan et al. (2005) proposed a method for further reducing the slice if the property conforms to certain specific formats, such as LTL formulas in the format $\mathsf{G}(p \Rightarrow \mathsf{F}q)$. The slices were reduced by removing areas of the slice in which the property holds vacuously due to the antecedent of the formula evaluating to *false*.

Slicing Precision and Correctness

The most desirable slice would be the smallest possible slice: one which contains exactly all of the statements which influence the criterion and no more. This is known as the *optimal* slice. However, Weiser (1984) showed this to be undecidable, because the outcome of conditions, such as the guards of *if* statements, is undecidable in general. Müller-Olm and Seidl (2001) further showed that finding the optimal slice for multi-threaded programs without synchronisation or procedure calls is PSPACE-hard and for multi-threaded programs without synchronisation or loops it is NP-hard. They concluded that there can be no efficient optimal slicing algorithm for concurrent programs.

Since the optimal slice is unattainable, the goal in slicing is to obtain a slice which is as *precise* as possible while still maintaining correctness. Precision refers to the size of the slice. The fewer statements in the slice, the more precise it is. Nevertheless, precision cannot be achieved at the price of correctness. It is essential that the slice is correct. A *correct* slice is one which contains all of the statements which are relevant to the criterion. An incorrect slice is one which is missing some necessary statements and will therefore exhibit behaviour which is different than the original program. This requirement is especially important when slicing is applied for model checking, because the slice must satisfy the same properties as the original model. A slice which is correct but also contains unnecessary statements is referred to as a *conservative* slice. The largest conservative slice is the entire program itself.

An important consideration when creating a slicing algorithm is therefore to ensure that the slices produced are correct. Correctness can be defined in several ways. Weiser (1984) used a notion of projection to define correctness. A slice must exhibit a projection of the behaviour of the original program, with respect to the variables and the program statement given in the slicing criterion. Weiser's definition restricts slicing to programs with terminating behaviour.

Podgurski and Clarke (1990) defined correctness using what they refer to as *semantic dependence*. Semantic dependence occurs between two statements if a change in the semantics of one statement can affect the execution of the other. The observation of Weiser (1984) shows that identifying all such semantic dependencies is undecidable in general. However, Podgurski and Clarke demonstrated a

useful relation between semantic dependence and *syntactic dependence*. Syntactic dependence is simply the transitive closure of control and data dependencies. *Strong syntactic dependence* occurs between two statements if the statements are linked transitively by strong control dependence or data dependence, whereas *weak syntactic dependence* occurs if the statements are linked transitively by weak control dependence or data dependence. Podgurski and Clarke showed that semantic dependence implies weak syntactic dependence but not strong syntactic dependence. In other words, the possible non-termination of loops can result in a semantic dependence between two statements and must therefore be taken into consideration when slicing, in order to produce slices which are correct.

The notion of correctness is sensitive to the purpose of the slicing algorithm and the semantics of the language under consideration. For example, although Podgurski and Clarke's semantic dependence was sufficient for their purposes, Kumar and Horwitz (2002) found this definition to be insufficient for handling programs with unstructured control flow, such as *go-to* or *jump* statements. They provided an extension of semantic dependence which includes such jump statements.

When slicing is used for reduction of models for verification, the notion of correctness is related to the preservation of properties. It is essential that both the slice and the original model preserve the same set of properties, thereby guaranteeing that a verification result obtained using the slice is the same as what would have been obtained if the original model had been used. The correctness of the Indus slicer, which slices Java programs, was shown using a notion of projection (Hatcliff, et al., 1999; Hatcliff, et al., 2000), similar to Weiser's projection to demonstrate that their approach preserves LTL_{-X} properties. The difference between Weiser's projection and the projection of Hatcliff et al. is due to the slicing criterions. Weiser used slicing criterions that included both a program statement and a set of variables of interest. When slicing for verification, the criterion normally only consists of a set of nodes without any specific variables of interest. Thus, in the definition of projection given by Hatcliff et al., the variables of interest correspond to the ones referenced by the nodes in the criterion. This approach has also been used by other authors for discussing the correctness of slicing.

Similarly, Rakow (2008) developed an approach for slicing Petri Nets which is guaranteed to preserve LTL_{-X} properties. Wasserrab et al. (2009) provided a general framework for proving the correctness of slicing. They used the theorem prover Isabelle to verify their proof. The framework allows any backwards static intraprocedural slicing algorithm to be easily proved by simply showing that the CFG and the control dependence relation satisfy certain properties. They used a weak simulation relation to show the correctness of slicing. However, since their purpose was not verification, they did not relate the weak simulation relation to any form of temporal logic. The approach by Brückner (2007), for slicing CSP-OZ-DC, was shown to preserve properties specified in Duration Calculus. Ranganath et al. (2007) showed the correctness of their slicing approach by relating the original program and the slice using a weak bisimulation relation. Weak bisimulation is especially suited for slicing as it allows for the presence of invisible steps. The nodes in the slicing criterion can be considered to be the *observable* steps, while all others are *stuttering*, corresponding to the invisible steps in the weak bisimulation. In the next section, the concepts of bisimulation will be introduced.

2.2.5 Bisimulation

Bisimulation is a commonly used technique for proving equivalence between two transition systems. It can be divided into the *strong* and *weak* forms. Strong bisimulation requires that every step that is made in one transition system is matched by a step in the other system, as given by the following definition. In this section, all the definitions will be given in terms of doubly-labelled transition systems, as these will be used as in the next chapter to represent the underlying framework of Behavior Trees.

DEFINITION 7. BISIMULATION

Let T_1, T_2 be doubly-labelled transition systems such that $T_i = (\mathcal{S}_i, AP_i, \mathcal{J}_i, \mathcal{L}_i, A_i, \longrightarrow_i)$, for $i \in \{1,2\}$.

A relation $\mathcal{R}$ is a bisimulation iff for every $s \mathcal{R} t$, where $s \in \mathcal{S}_1$ and $t \in \mathcal{S}_2$, the following holds:

1a) $\forall s' \in \mathcal{S}_1$ and $a \in A_1$, such that $s \xrightarrow{a} s'$, $\exists t' \in T_2$ such that $t \xrightarrow{a} t'$ and $s' \mathcal{R} t'$ and

1b) $\forall t' \in \mathcal{S}_2$ and $a \in A_2$, such that $t \xrightarrow{a} t'$, $\exists s' \in T_1$ such that $s \xrightarrow{a} s'$ and $s' \mathcal{R} t'$.

T_1 and T_2 are bisimilar, denoted $T_1 \rightleftharpoons T_2$, iff there exists a bisimulation $\mathcal{R}$, such that $s_0 \mathcal{R} t_0$ for all $s_0 \in \mathcal{I}_1$ and $t_0 \in \mathcal{I}_2$.

■

On the other hand, the weak forms of bisimulation allow for the presence of invisible τ steps, which do not have to be matched by the other system. Since slicing removes transitions from the transition system, the resulting paths would contain fewer steps than the equivalent paths in the original. Strong bisimulation is therefore unsuitable for establishing the equivalence between a model and its slice. The stuttering transitions, which do not form part of the slicing criterion, can be thought of as invisible or silent steps which the slice is not required to emulate. Using this approach, the weak forms of bisimulation are appropriate for establishing equivalence for slicing.

Two common forms are *weak bisimulation* (Bloom, 1995) and *branching bisimulation* (van Glabbeek & Weijland, 1996). These forms of bisimulation are defined over labelled transition systems. Weak bisimulation, as defined in (Bloom, 1995) is a direct modification of strong bisimulation. Instead of requiring each step in one system to be matched by another, it requires that each observable step in one system be matched by an observable step in the other system, preceded by any number of invisible τ steps. This definition would appear to be an effective method for applying bisimulation concepts to systems with stuttering steps. However, there are cases where it is not suitable, as weak bisimulation cannot distinguish between two systems which perform identical steps but have different branching structures. In order to differentiate between two systems such as this, van Glabbeek and Weijland (1996) proposed an equivalence known as *branching bisimulation*. Branching bisimulation is similar to weak bisimulation, except that it differentiates between two systems that have different branching behaviour. Branching bisimulation is defined in the following definition, where $s \dashrightarrow s'$ denotes a stuttering step and $s \dashrightarrow^* s'$ denotes zero or more stuttering steps.

DEFINITION 8. BRANCHING BISIMULATION

Let T_1, T_2 be doubly-labelled transition systems such that $T_i = (\mathcal{S}_i, AP_i, \mathcal{I}_i, \mathcal{L}_i, A_i, \longrightarrow_i)$, for $i \in \{1,2\}$.

A relation $\mathcal{R}$ is a branching bisimulation iff for every $s \mathcal{R} t$, where $s \in \mathcal{S}_1$ and $t \in \mathcal{S}_2$, the following holds:

1a) $\forall s' \in \mathcal{S}_1$ and $a \in A_1$, such that $s \xrightarrow{a} s'$, either $s \dashrightarrow s'$ and $s' \mathcal{R} t$ or
$\exists t', t'' \in \mathcal{S}_2$ such that $t \dashrightarrow^* t'' \xrightarrow{a} t'$, $s \mathcal{R} t''$ and $s' \mathcal{R} t'$ and

1b) $\forall t' \in \mathcal{S}_2$ and $a \in A_2$, such that $t \xrightarrow{a} t'$, either $t \dashrightarrow t'$ and $t' \mathcal{R} s$ or
$\exists s', s'' \in \mathcal{S}_2$ such that $s \dashrightarrow^* s'' \xrightarrow{a} s'$, $s'' \mathcal{R} t$ and $s' \mathcal{R} t'$.

T_1 and T_2 are branching-bisimulation equivalent iff there exists a branching bisimulation $\mathcal{R}$, such that $s_0 \mathcal{R} t_0$ for all $s_0 \in \mathcal{I}_1$ and $t_0 \in \mathcal{I}_2$.

■

As with weak bisimulation, branching bisimulation requires every observable step in one system to be matched in the other system by a sequence of zero or more stuttering steps followed by a matching observable step. The difference is that the intermediate state t'', that is reached after the stuttering steps, has to be related to the first state in the original model, s.

De Nicola and Vaandrager (1995) showed the relation between branching bisimulation and its state-based counterpart, *stuttering bisimulation*, defined over Kripke structures. Stuttering bisimulation is

known to preserve CTL$^*_{-X}$ properties. De Nicola and Vaandrager used doubly-labelled transition systems to develop a conversion function to translate between labelled transition systems and Kripke structures. Using this, they established the equivalence between branching bisimulation and *divergence-blind stuttering bisimulation*. *Divergence* refers to infinite stuttering paths. These are ignored by divergence-blind stuttering bisimulation, which therefore preserves only a variant of CTL$^*_{-X}$ defined over finite paths only. As branching bisimulation is equivalent to divergence-blind stuttering bisimulation, it also only preserves CTL$^*_{-X}$ over finite paths. Since the usual definition of CTL* is defined over maximal paths, it is necessary to ensure that infinite stuttering paths are preserved. *Divergence-sensitive stuttering bisimulation* preserves CTL$^*_{-X}$ over maximal paths. De Nicola and Vaandrager showed that *divergence-sensitive branching bisimulation*, a variant of branching bisimulation, is equivalent to divergence-sensitive stuttering bisimulation and therefore preserves CTL$^*_{-X}$ over maximal paths.

For their definition of divergence-sensitive branching bisimulation, De Nicola and Vaandrager did not directly incorporate the notion of divergence into the definition. Instead, they created an extra state in the transition system to represent divergence and linked all divergent states to that new state. However, as noted by van Glabbeek et al (2009), using this method, *livelocked* states cannot be distinguished from *deadlocked* states. Livelocked states are ones which have self loops, while deadlocked states are ones which have no outgoing transitions. van Glabbeek and Weijland (1996) proposed a new version, called *branching bisimulation with explicit divergence*, which incorporates the divergence requirement into the definition itself, instead of making modifications to the transition system. The definition for branching bisimulation with explicit divergence is given in Definition 9, taken from van Glabbeek et al. (2009a). Branching bisimulation with explicit divergence was shown to preserve CTL$^*_{-X}$ by van Glabbeek et al. (2009b). This result has been re-stated as Theorem 1 below. They further showed that their definition is equivalent to other forms of divergence present in the literature (van Glabbeek, et al., 2009a).

DEFINITION 9. BRANCHING BISIMULATION WITH EXPLICIT DIVERGENCE

Let T_1, T_2 be labelled transition systems such that $T_i = (\mathcal{S}_i, AP_i, \mathcal{I}_i, \mathcal{L}_i, A_i, \longrightarrow_i)$, for $i \in \{1,2\}$.

A relation $\mathcal{R}$ is a branching bisimulation with explicit divergence iff for every $s \mathcal{R} t$, where $s \in \mathcal{S}_1$ and $t \in \mathcal{S}_2$, the following holds:

1a) $\forall s' \in \mathcal{S}_1$ and $a \in A_1$, such that $s \xrightarrow{a} s'$, either $s \dashrightarrow s'$ and $s' \mathcal{R} t$ or
$\exists t', t'' \in \mathcal{S}_2$ such that $t \dashrightarrow^* t'' \xrightarrow{a} t'$, $s \mathcal{R} t''$ and $s' \mathcal{R} t'$,

1b) $\forall t' \in \mathcal{S}_1$ and $a \in A_2$, such that $t \xrightarrow{a} t'$, either $t \dashrightarrow t'$ and $t' \mathcal{R} s$ or
$\exists s', s'' \in \mathcal{S}_2$ such that $s \dashrightarrow^* s'' \xrightarrow{a} s'$, $s'' \mathcal{R} t$ and $s' \mathcal{R} t'$,

2a) if there is an infinite path $s \dashrightarrow s_0 \dashrightarrow s_1 \dashrightarrow ...$, such that $s_i \mathcal{R} t$, $\forall i \geq 0$, then there exists an infinite path $t \dashrightarrow t_0 \dashrightarrow t_1 \dashrightarrow ....$, such that $t_j \mathcal{R} s_i$, $\forall i, j \geq 0$, and

2b) if there is an infinite path $t \dashrightarrow t_0 \dashrightarrow t_1 \dashrightarrow ...$, such that $t_i \mathcal{R} s$, $\forall i \geq 0$, then there exists an infinite path $s \dashrightarrow s_0 \dashrightarrow s_1 \dashrightarrow ....$, such that $s_j \mathcal{R} t_i$, $\forall i, j \geq 0$,

T_1 and T_2 are branching-bisimulation with explicit divergence equivalent, denoted $T_1 \triangleq T_2$, iff there exists a branching bisimulation $\mathcal{R}$, such that $s_0 \mathcal{R} t_0$ for all $s_0 \in \mathcal{I}_1$ and $t_0 \in \mathcal{I}_2$.

■

THEOREM 1. BRANCHING BISIMULATION WITH EXPLICIT DIVERGENCE PRESERVES CTL*-X

For two transition systems T_1 and T_2, $T_1 \triangleq T_2 \Rightarrow (T_1 \vDash \varphi \Leftrightarrow T_2 \vDash \varphi$, for all $\varphi \in \text{CTL}^{*}_{-X})$.

Proof.

This result was proven by van Glabbeek et al. (2009b). □

2.3 Behavior Trees

Formal methods are essential for building correct and safe systems. Despite this, there is often a large gap between the informal requirements provided by the user and the formal specification of the system. The Behavior[‡] Engineering process aims to close this gap (Dromey, 2003, 2005), by providing a rigorous translation scheme from the informal textual requirements to the formal model, and maintaining strong links between them. The Behavior Engineering process, created by Dromey (2003, 2005), consists of three types of models: Behavior Trees, Composition Trees and Structure Trees. As their names suggest, Behavior Trees model the behaviour of a system, Composition Trees define the composition, such as which components belong to the system, and Structure Trees model the structure, i.e. how the components fit together. In this thesis, only Behavior Trees will be considered, as this is the type of model which is used as the input for model checking. Behavior Trees have a formal semantics (Colvin & Hayes, 2011). The Behavior Engineering process is illustrated in Figure 9. The process begins with a set of informal textual requirements. Each requirement is first translated into an individual Requirement Behavior Tree (RBT). Next, all of the RBT's are merged together to create an Integrated Behavior Tree (IBT). The IBT is then transformed into a Design Behavior Tree (DBT), by making design decisions. This DBT can then be verified, by automatic translation into a model checking language (Grunske, et al., 2008), simulated (Wen, et al., 2007) or used in a model-driven engineering framework to produce an implementation (Myers, 2010).

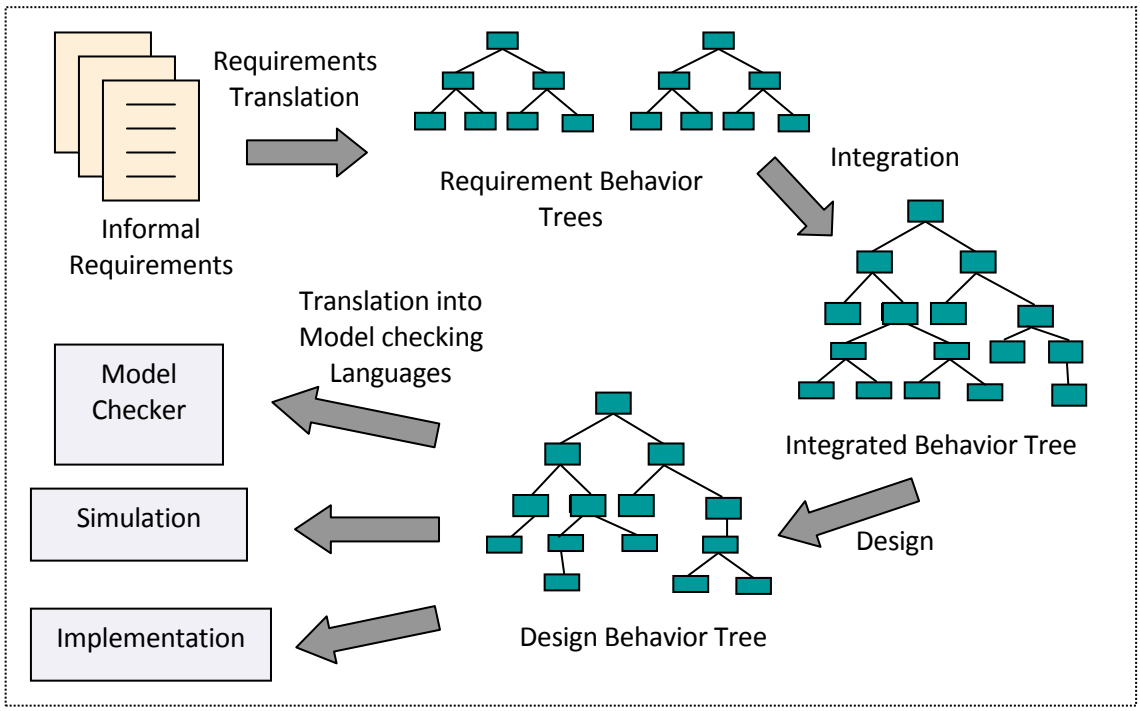


Figure 9. Behavior Engineering Process

[‡] The names Behavior Engineering and Behavior Trees have been trademarked using the American spelling and capitals for each word. When not referring to these proper nouns, the British spelling of *behaviour* will be used.

2.3.1 Behavior Tree Notation

Behavior Trees are graphical models in a tree-like form. Nodes are represented as rectangular boxes which contain information on the behaviour that has taken place, as well as an identifier linking it to the original textual requirement that it came from. Branches are used to model either alternative choices or parallel threads. Contrary to its name, however, Behavior Trees are not actual trees, but instead are directed cyclic graphs. This is due to the presence of *reversion* and *reference* nodes at the leaf nodes of threads, which transfer the control flow to another location in the tree. The full syntax of Behavior Trees is given in the Behavior Tree Notation Document v.1.0 ("Behavior Tree Group, Behavior Tree Notation v1.0," 2007). In this thesis, only the subset of the syntax that is used for translation into the model checking languages will be considered. For example, relational behaviour will not be covered as it is not currently included in the translation process. The treatment of the excluded language features remains as future work.

A Behavior Tree consists of nodes and edges. The parts of a node are shown in Figure 10. The *component* is the name of the component or attribute which is performing some behaviour. The name of the *behaviour* is given below it. The node *type* is indicated by the symbols on either side of the behaviour name. Some nodes may additionally have a *flag*[§]. The box on the left is the *requirements tag*, which is an identifier linking the node back to the original textual requirements of the system.

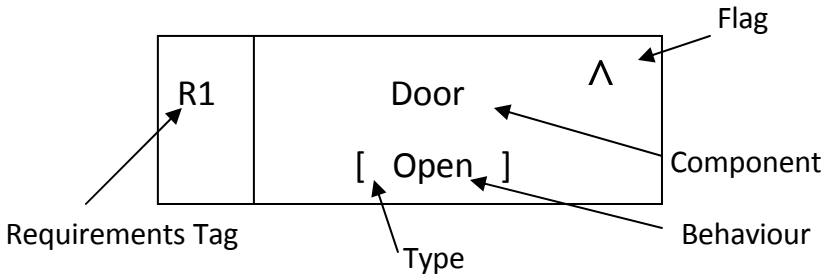


Figure 10. A Behavior Tree Node

Figure 11 shows the possible node types:

(a) *State-realisation:* The component C is updated to the state s.
(b) *Selection:* Control flow passes through this node if the component C is currently in the state s. If not, the thread terminates.
(c) *Guard:* Control flow passes through this node if the component C is currently in the state s. Unlike selections, if C is not in state s, the control flow waits at this location until the condition becomes true. Note that in the rest of this thesis, the guard will be referred to as *BTguard*, in order to avoid confusion with other notions of guards.
(d) *Internal Input Event:* The component C receives a message m from another component in the same Behavior Tree. There must be at least one internal output event node in the Behavior Tree sending the message m.
(e) *Internal Output Event:* The component C sends a message m to another component in the same Behavior Tree. There must be at least one internal input event node in the Behavior Tree receiving the message m.
(f) *External Input Event:* The component C receives a message m from the external environment.
(g) *External Output Event:* The component C sends a message m to the external environment.

[§] The *flag* was referred to as the *operator* in the Behavior Tree Notation Document v. 1.0 ("Behavior Tree Group, Behavior Tree Notation v1.0," 2007), but was known as the *flag* in previous work.

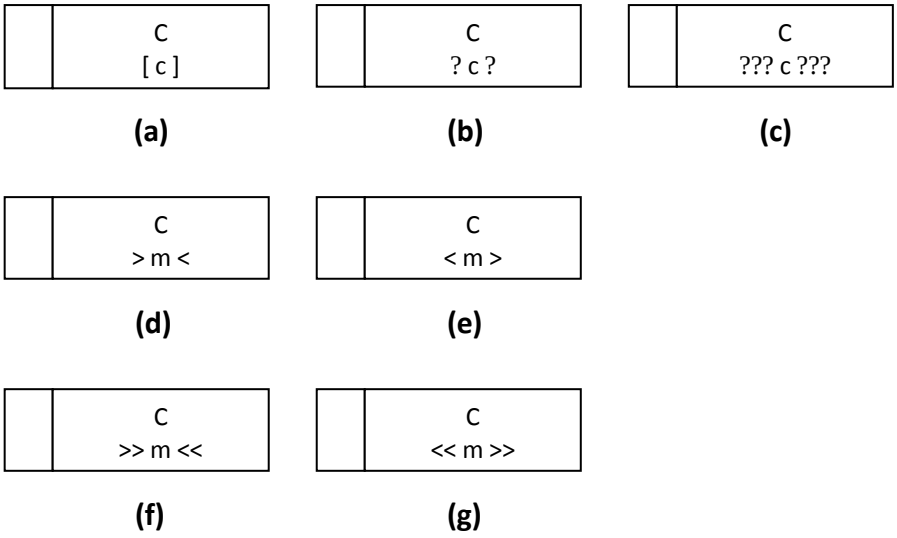


Figure 11. Behavior Tree Node Types

Additionally, nodes may refer to *attributes* of components, by replacing the behaviour name with an expression involving an attribute. For example, if a state realisation node has a component name of C and a behaviour of A := b, it represents that the attribute A of component C has realised state b. In addition to realisation of states, attributes can be assigned to numerical values using attribute expressions, such as A := A + 1. Selection and guard nodes can have any boolean expression involving an attribute, such as A > 5 or A = b.

Nodes which involve a conditional test, i.e. nodes of type selections, BTguards, internal input events and external input events, will be referred to in this thesis as *conditional* nodes.

The nodes are joined together using arrows, representing the control flow. Sequential flow is modelled using a normal arrow, as depicted in Figure 12 (a), while atomic connections are modelled using a straight line, as shown in Figure 12 (b). Atomic nodes represent uninterruptible sections of behaviour; i.e. no node in another thread can interrupt the atomic block.

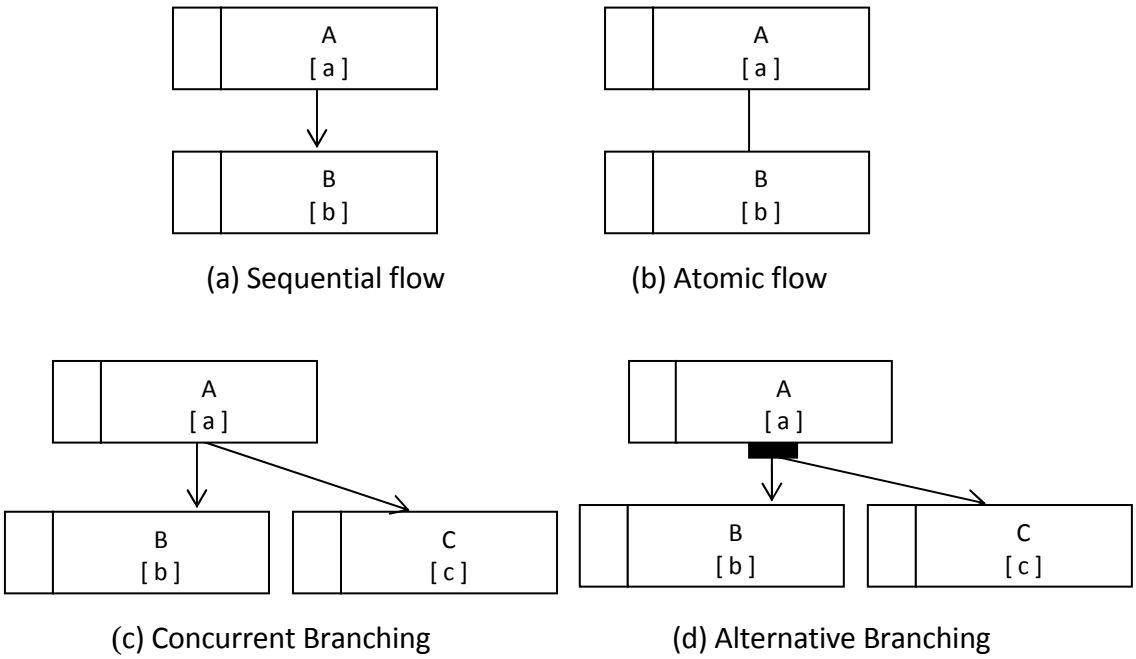


Figure 12. Sequential and Atomic Control Flow

There are two forms of branching in Behavior Trees: concurrent and alternative. Concurrent branching, as shown in Figure 12 (c) represents parallel threads. In the example, after the node A[a] executes, the two nodes B[b] and C[c] and their sub-trees execute in parallel, i.e. in any possible interleaving order. Alternative branching, shown in Figure 12 (d), represents a choice between two possibilities in a single thread. In the example, either the node B[b] or the node C[c] will be selected. When one branch is chosen, the other immediately terminates. As for concurrent branching, the nodes may be of any type. However, there is one restriction for alternative branching: either all the branches begin with a selection node or none of them.

Behavior Trees are designed to be able to model infinite behaviour. This is accomplished by the use of *reversion* and *reference* nodes, which cause the control flow to jump to another location in the tree. These two types of nodes are shown in Figure 13 (a) and (b), respectively. Reversion nodes are modelled using the "^" symbol and reference nodes are modelled using the "=>" symbol. Both reversion and reference nodes must be leaf nodes. Reversions cause the control flow to revert to a location higher up in the tree. The new location is referred to as the *target* of the reversion. The target must be an ancestor of the reversion node. Reference nodes are similar, however they cause the control flow to jump to another location which is not necessarily an ancestor. The target of a reference node must be in the same thread, but may be in an alternative branch. Reversion and reference nodes can be any type of node. The target nodes are identified by locating a node with the same component name, behaviour name and type. Another property of reversion nodes is that when a reversion executes, every thread which was a descendent of the target node is terminated. The purpose of this is to avoid having sub-threads continue to execute while their parent thread has re-started.

Figure 13 (c) shows a thread kill node. The purpose of thread kill nodes is to terminate another thread. Figure 13 (d) shows a synchronisation node. Control flow remains blocked at this node until all of the other nodes involved in the synchronisation have been reached. In this thesis, these nodes will be referred to as the *synchronising partners*. The synchronising partners are identified by finding other nodes with the synchronisation flag. When all of the synchronising partners are ready to execute, the node may execute. If the synchronisation node is a conditional node, its condition is only evaluated after all of its synchronising partners have been reached. Note that a synchronisation flag can be used in conjunction with one of the other types of flags, in order to cause a reversion, reference or thread kill node to synchronise.

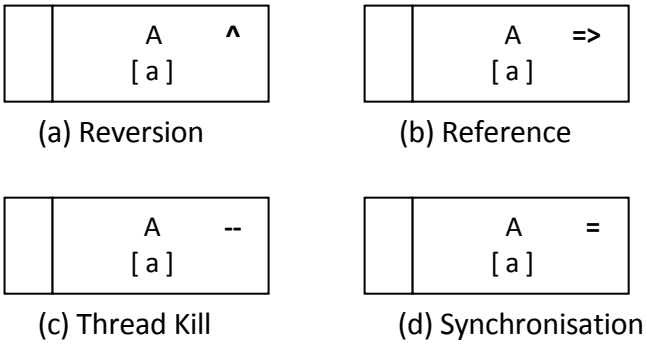


Figure 13. Behavior Tree Flags

Behavior Trees are able to handle set operations. The syntax for this is shown in Figure 14. In the nodes in the figure, S and T are set attributes of the component C. The same set operations can also be performed on sets that are components themselves. The nodes correspond to the following:

(a) *Addition to a set:* The element x is added to the set S.
(b) *Subtraction from a set:* The element x is removed from the set S.
(c) *Set intersection:* The set S is updated to the intersection of sets S and T.
(d) *Set union:* The set S is updated to the union of sets S and T.
(e) *Set difference:* The set S is updated to the difference between sets S and T.

(f) *Set membership:* The condition is true if the element x belongs to the set S. This node can also be a BTguard type.
(g) *Set cardinality:* The condition is true if the set S contains greater than k elements. The expression can contain either <, > or =. This node can also be a BTguard type.

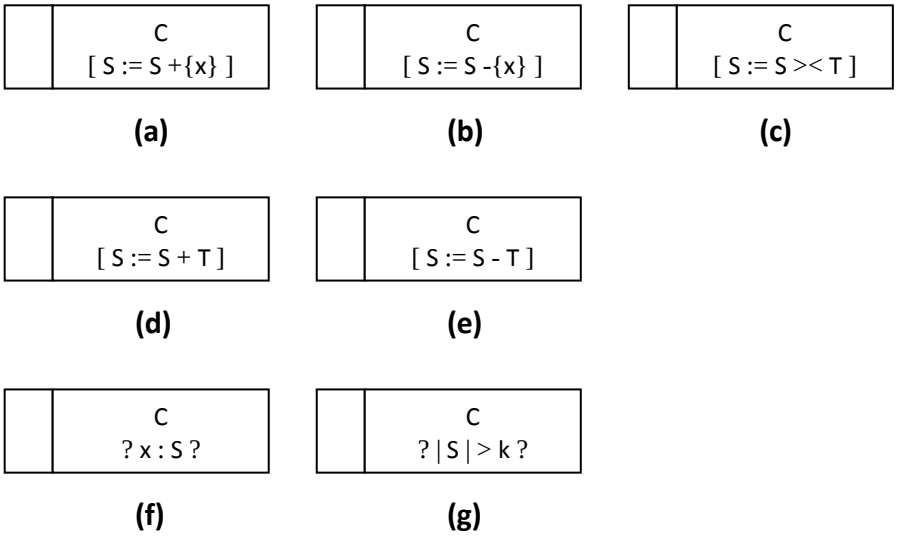


Figure 14. Set Operation Nodes

In this thesis, the following auxiliary functions will be used, to refer to the various elements of the Behavior Tree notation. The function *comp*(*n*) returns the component name of node *n* and *behav*(*n*) return the behaviour name of node *n*. If the node defines or uses an attribute, *attr*(*n*) returns the attribute and *attrExp*(*n*) returns the expression involving the attribute. Note that a unique name for the attribute is given comprising the component and attribute names, to avoid confusion if other components have attributes of the same name. For example, if the node is C ?A = b?, *attr*(*n*) would return C_A and *attrExp*(*n*) would return "C_A = b". The type of a node is given by the function *type*(*n*) and the flag by the function *flag*(*n*). Two or more nodes are designated as *matching* if they have the same component name, behavior and type. If *matching*(*p*,*q*) then *comp*(*p*) = *comp*(*q*), *behav*(*p*) = *behav*(*q*)

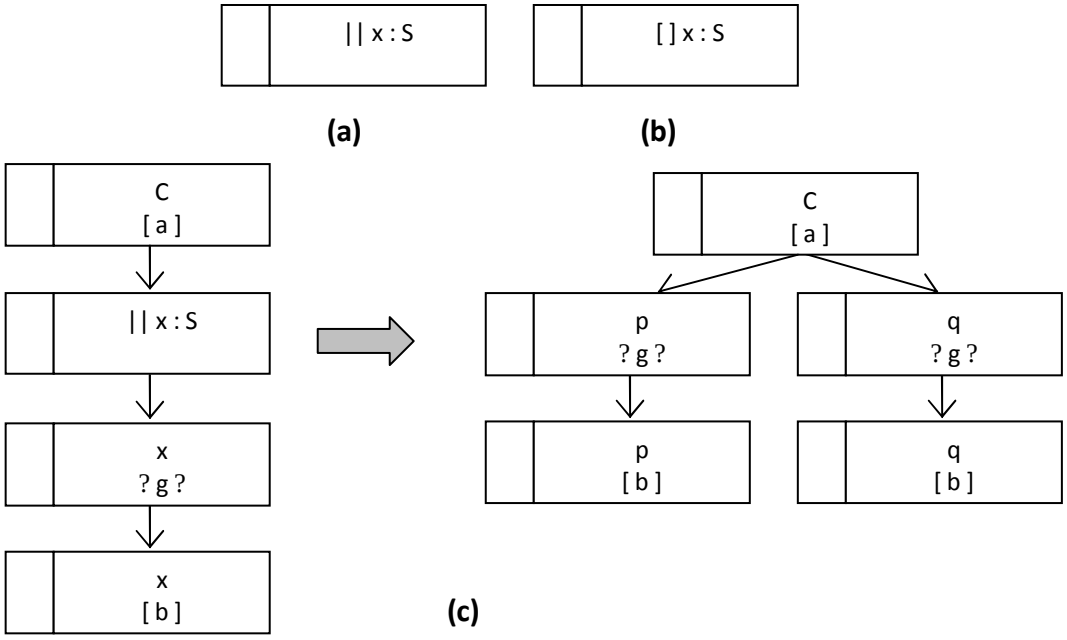


Figure 15. For-all and For-one nodes

and *type*(*p*) = *type*(*q*). The function *target*(*n*) returns the target node of *n*, if *n* is a reversion, reference or thread kill; it is undefined for all other nodes.

For a set operation node *n*, the functions *comp*(*n*), *behav*(*n*), *attr*(*n*) and *attrExp*(*n*) operate in the same manner as for non-set nodes, treating the set name as either a component or attribute name and the expression involving the set as either a behaviour name or attribute expression. Additionally, Behavior Trees have *for-all* and *for-one* nodes, as depicted in Figure 15 (a) and (b), respectively. The nodes specify that the sub-trees below should be evaluated over all or one of the elements x in the set S. This is accomplished by expanding the sub-tree below to have one branch for each element in the set. In each branch, every occurrence of x is replaced by one of the elements in the set. If it is a *for-all* node, the branches are joined by a concurrent branching point, while if it is a *for-one* node, they are joined by an alternative branching point. An example of this is depicted in Figure 15 (c). Assume the set S contains two elements: p and q. The sub-tree below the *for-all* node is replaced by two branches, one for the element p and the other for the element q.

If *conc*(*p*,*q*) then nodes *p* and *q* are in concurrent threads. If *alt*(*p*,*q*) then nodes *p* and *q* are in alternate branches of the same thread. In this thesis, a node with multiple children, such as A[a] in the figures, will be referred to as a *branching node*. If there is an edge linking node *m* to node *n*, then *m* is the *parent* of *n*, given by *parent*(*n*). The node *n* is referred to as a *child* of *m*. Note that in Behavior Trees, a node may have zero or more children, given by the function *children*(*n*), but only one parent. The function *childNum*(*n*) returns the number of children of *n* and the function *child*(*n*, *i*) returns the i^{th} child of *n*. Every node has a parent except the *root* node. The set *ances*(*n*) gives the *ancestors* of *n*, where *ances*(*n*) = *parent*(*n*) ∪ *ances*(*parent*(*n*)). The set *desc*(*n*) gives the *descendents* of *n*, which are the nodes that have *n* as an ancestor. The descendents of a node *n* form a *sub-tree* with *n* as its root. A node is not an ancestor nor a descendent of itself.

In this thesis, as a convention each function may additionally be given a sub-script denoting the Behavior Tree it refers to. For example, $comp_b(n)$ returns the component name of node *n* in Behavior Tree *b*.

2.3.2 Requirements Translation and Integration

The first step in the Behavior Engineering process is to create individual RBT's for each requirement. The intent of Behavior Engineering is to provide a mechanism for creating a formal model "*out* of its requirements instead of *from* its requirements" (Dromey, 2003, 2005), which is accomplished by proposing a rigorous approach for requirements translation. Each sentence of the textual requirement is translated into nodes by identifying the components and behaviour described. The nodes are given tags corresponding to the label of the requirement, to maintain traceability to the requirements document.

Next, each of the RBT's are merged together into an IBT. The root node of each RBT must match a node in the IBT. This represents the point at which the pre-condition of the RBT is established, so that is the location where the RBT should be inserted. If no matching node can be found, it indicates that some information is missing from the requirements, since the required pre-condition never occurs. This must then be rectified by consultation with the clients or by making assumptions. If an assumption is made, it is noted in the tag of the corresponding nodes using a "+" symbol to indicate an implied requirement or a "-" to indicate a missing requirement. Additionally, while normal nodes are coloured green, nodes indicating implied behaviour are coloured yellow and nodes indicating missing behaviour are coloured red. For the purposes of this thesis, the tags and colours are irrelevant, as the slicing approaches that follow can be applied to any nodes regardless of their colour or tag information. Accordingly, in some of the diagrams that follow, the nodes may not have any colour and the tags may be left blank. The following example illustrates the process of integrating RBT's.

Example.
Consider the RBT shown in Figure 16 on the right, corresponding to a requirement numbered R4. It states that when the oven is cooking, if the door is opened the powertube will be turned off. The current IBT is given on the left of the figure. It has been created by merging requirements R1 to R3. The root node of the RBT is Oven [cooking]. This is the pre-condition which must be established by the

IBT. There is a matching Oven [cooking] node in the IBT, so the RBT is joined to the IBT at that position. The final IBT is given in Figure 17.

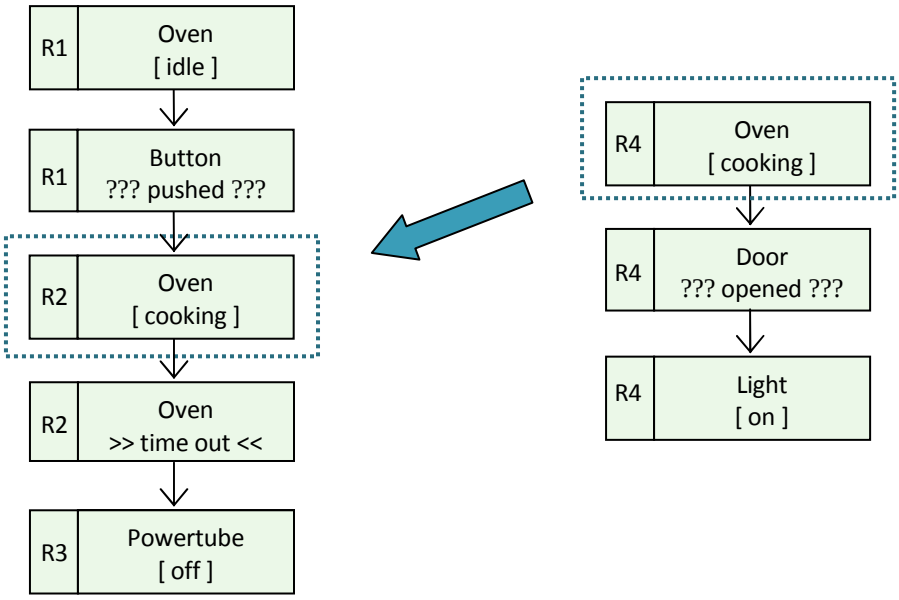


Figure 16. Identifying Matching Pre-conditions

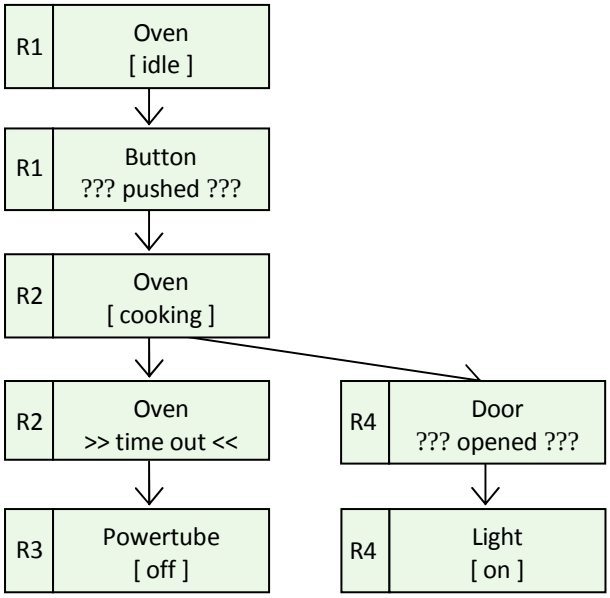


Figure 17. Final IBT

2.3.3 Model Checking Behavior Trees

Behavior Trees can be translated into the input languages of various model checkers to allow them to be verified. At present, automatic translators exist for translating Behavior Trees into the input languages of two model checkers (Wen, et al., 2007): the Symbolic Analysis Laboratory (SAL) and UPPAAL model checkers. The SAL suite (de Moura, et al., 2004) is a set of model checking tools, including a symbolic model checker for LTL and a bounded model checker. In previous work, the

symbolic model checker was used for verifying properties on Behavior Tree models (Grunske, et al., 2011). The UPPAAL model checker (Larsen, et al., 1997) allows timed behaviour to be verified.

The process of translating Behavior Trees into the input languages of SAL and UPPAAL consists of two stages: the parsing stage and the translation stage. The parsing stage identifies a sequence of syntax rules which can be used to construct the given Behavior Tree, in the process determining whether or not the Behavior Tree is well-formed. If there are no syntax errors, the translation stage begins. The SAL or UPPAAL code is produced by executing translation rules that correspond to the sequence of syntax rules. This systematic approach allows the translators to be easily extended or modified, simply by changing the necessary translation rules. In the final code, the Behavior Tree is represented as a form of transition system, in which each node or block of atomic nodes corresponds to a transition. The code makes use of *program counters*, which are integer variables designed to keep track of the current location in the Behavior Tree. If there is a block of atomic state realisations at the top of the tree, these are translated as the initialisation section of the code; otherwise the root node becomes the initialisation. Alternatively, the initialisation can be provided by the user as a separate text file. For further details of the translation process, refer to Grunske et al. (2008).

3 SLICING BEHAVIOR TREES

Creating a slice of a Behavior Tree model follows the same process as for program slicing. The Behavior Tree is first converted into a *control flow graph* that shows the control flow of the model. Using the information in the control flow graph, the dependencies between the nodes are identified and represented in a *dependence graph*. This dependence graph can then be used to identify the relevant nodes with respect to a given slicing criterion, extracted from the temporal logic theorem. The relevant nodes form a *slice set*. The slice set is then merged back into a syntactically correct Behavior Tree, which is the slice. Figure 18 depicts the overall process of Behavior Tree slicing. Sections 3.1 and 3.2 define control flow graphs for Behavior Trees and the underlying transition system, respectively. Section 3.3 defines the various dependency types. Section 3.4 describes the process of merging the slice set into a tree and the slicing algorithm is given in Section 3.5. A proof of correctness is given in Section 3.6, which guarantees that the slice will preserve the validity of the property of interest, so it can be used in place of the larger original Behavior Tree for model-checking.

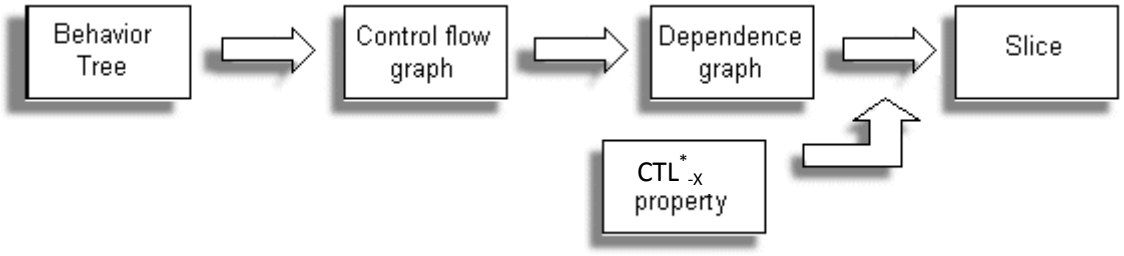


Figure 18. Overview of the Behavior Tree Slicing Process.

Type of Slicing

Model checking verifies all possible paths of the system. Dynamic slicing assumes that only one path is of interest, so static slicing is the most suitable. A backwards slicing approach is required, since the objective is to find all the nodes that influence the criterion. This is in accordance with other approaches for using slicing for model checking purposes, such as Hatcliff et al. (2000) and Brückner (2007).

The final slice will be used in place of the original model for verification. Therefore, it must be executable, i.e. it should have a valid Behavior Tree structure. There should not be any disconnected nodes in the slice and it should conform to the rules of Behavior Tree structures, for example that all reversions should point to an ancestor. Since the backwards traversal of the dependence graph might not produce an executable Behavior Tree, an extra post-processing step is needed to make the necessary modifications to the slice.

3.1 Creating a BT Control Flow Graph

The first step in slicing Behavior Trees is to create a *BT control flow graph*. Behavior Trees model the control flow of systems, but they cannot be used as a control flow graph directly, due to the information which is implicit in the tree. Specifically, for selections, guards, synchronisations and input event nodes, the path where the condition is unsatisfied is not explicitly represented in the Behavior Tree. In a BT control flow graph, all such paths would be represented as a *false* branch from the node. For selections, when the condition is unsatisfied, the thread terminates, so an end node must be introduced and the false branch must link to it. For guards, synchronisations and input event nodes, the control flow waits until the condition becomes satisfied, so the false branch must revert back to the node itself.

Due to these differences, instead of using a Behavior Tree directly as a control flow graph, it must be transformed into a new structure, known as a *BT control flow graph*. A BT control flow graph is a directed graph $G = <N,E, start, end>$, where N is a set of nodes, each representing a node in the Behavior Tree, E is a set of edges representing the flow of control, such that $E = N \times N$, *start* is the start node and *end* is a set containing the end nodes. Since Behavior Trees may have multiple exit points or may model non-terminating behaviour, the use of a unique end node is impractical. As Ranganath et al. (2007) pointed out, a single end node is not always possible, particular for systems with infinite loops.

Similar terminology as for the CFG's of programs is assumed. Specifically, an edge $e \in E$, where $e = (m_1, m_2)$, indicates that m_2 is one of the nodes which can execute immediately after m_1. The node m_2 is known as an *immediate successor* of m_1. The edge from m_1 to m_2 is denoted by $edge(m_1, m_2)$. A *trace*[**] in a BT control flow graph consists of a sequence of nodes $< m_0, m_1, ..., m_k >$, where for every m_i, such that $0 \leq i < k$, $(m_i, m_{i+1}) \in E$. The function $trace(m_i, m_j)$ is used to denote a trace from node m_i to m_j. A *maximal trace* from node m is a trace that starts at node m and either ends at a leaf node or contains an infinite loop. The set of maximal traces from a node m is given by the function *maxTraces*(m). For every node m in the control flow graph, there exists a trace from *root* to m. Every edge in the control flow graph additionally has a label associated with it, to describe whether the edge corresponds to the *true* or *false* choice of a node. The function *label*(e) returns the label associated with the edge e.

Each node in a Behavior Tree is represented by a node in the corresponding control flow graph. Control flow graphs additionally have *end* nodes which do not correspond to Behavior Tree nodes. Apart from end nodes, the nodes in control flow graphs retain all the information of the corresponding Behavior Tree nodes, such as their component names and types. In the rest of this thesis, the term *node* will be used to refer to control flow graph nodes unless otherwise specified.

The following steps are used to construct a control flow graph from a Behavior Tree:

1) Create a node in the control flow graph to represent the root node of the Behavior Tree.

2) For each node n in the Behavior Tree which has a corresponding node m in the control flow graph, locate each of the children of n in the Behavior Tree. For each child, place a node c into the control flow graph, with an edge from m to c. In this manner, a control flow graph node will be created for every Behavior Tree node, with edges representing the arrows in the Behavior Tree.

3) For a single sequential node n in the Behavior Tree, locate its corresponding node in the control flow graph m. Then, label all of the outgoing edges of m as *true*. Insert an additional outgoing edge from m to a new *end* node. Label this edge *false*. This represents the semantics of selection nodes. If the condition of the selection is satisfied, the control flow may proceed to

[**] The term *trace* is used instead of *path* as in program CFG's to correspond with the trace of the underlying transition system, as described in the next section.

all subsequent nodes; otherwise the control flow for this thread terminates. See Figure 19 as an example.

Note that for a group of selection nodes connected by an alternative branching point, the same method is used for handling them but the condition represented by the *false* branches are different. For sequential selection nodes, the *false* branch represents the case where the selection's condition does not hold. However, for a group of alternative branching selection nodes, the *false* branches of each node represent the case where *all* of the selection nodes' conditions do not hold. For example, if the nodes were A?a? and B?b?, then the *false* branches of each node would represent NOT(A=a) AND NOT(B=b).

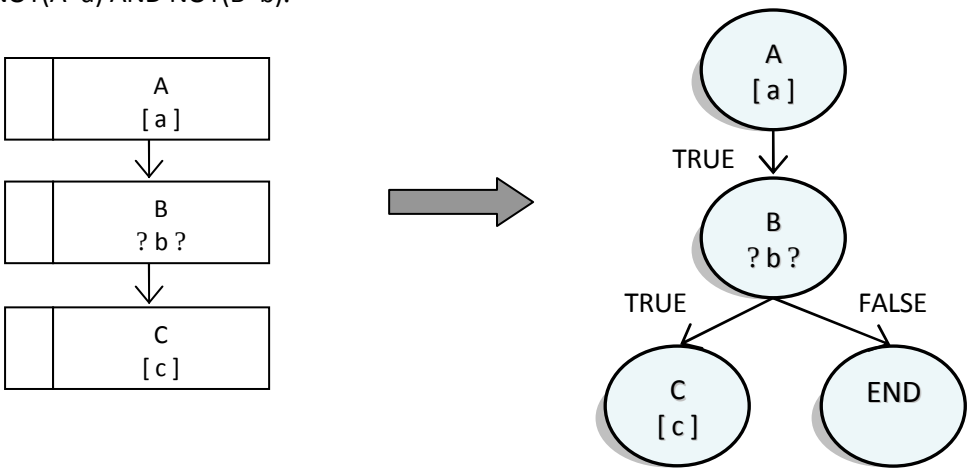


Figure 19. Representing a Selection Node

4) For each guard, synchronisation node or input event node (both external and internal event types) in the Behavior Tree, locate its corresponding node in the control flow graph *m*. Label all of the outgoing edges of *m* as *true*. Insert an additional outgoing edge from *m* back to itself, labelled *false*. This represents the "wait-until" semantics of guards, synchronisation nodes and input events. See the following diagram as an example. If a synchronisation node is also a conditional node, it will have two *false* edges in the control flow graph: one representing the *false* case of the condition and one for when the synchronising partners have not yet been reached. The following diagram illustrates this.

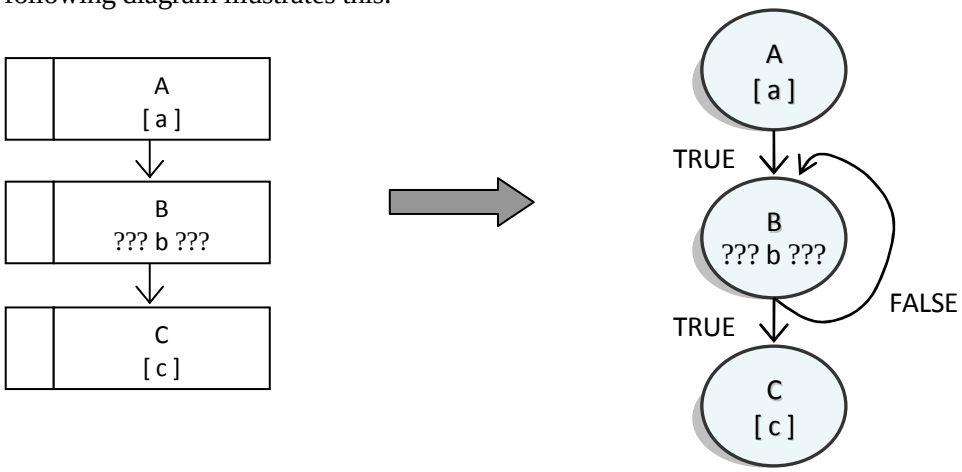


Figure 20. Representing a Guard Node

The following table summarises how each type of Behavior Tree node is represented in the BT control flow graph.

Behavior Tree node type	Representation in the BT control flow graph
State realisation or output event.	Represented by a single node; see Step 2 above.
Selection	Represented by a node with *true* & *false* outgoing edges, where *false* leads to an END node; see Step 3 above.
Guard, input event or synchronisation	Represented by a node with *true* & *false* outgoing edges, where *false* loops back to the node; see Step 4 above.

Table 1. Representation of Nodes in the BT Control Flow Graph

There are some significant differences between a BT control flow graph and the CFG's normally used for programs. These differences are unavoidable due to the differences in semantics between Behavior Trees and programs. Specifically:

- The nodes in a BT control flow graph may have more than two successors. This is due to the presence of concurrent and alternative branching.
- A BT control flow graph does not have a unique end node. Furthermore, each end node denotes the termination of a thread, not the entire system.
- Edges in a BT control flow graph retain information about their type, such as whether the edge is atomic or sequential.
- Reversion and reference nodes alter the control flow, although in a CFG only the edges denote flow of control.

3.1.1 Concurrent Branching

The threads of a BT control flow graph are taken to each start at the root node and continue until a leaf node is reached. See Figure 21 for an example of this. Note that due to alternative branching and conditional nodes, there may be more than one leaf node per thread. Each thread is given a unique identifier. Each node may belong to more than one thread. The identifiers of the threads which a node *m* belongs to are given by the function *threads*(*m*).

The concurrency models used by Krinke (1998) and Nanda and Ramesh (2000) are unsuitable for Behavior Trees. In Nanda and Ramesh (2000), they explore the use of two different concurrency models, one which they describe as having a complete interleaving semantics and one which depicts the concurrency semantics in Java. Concurrency in Behavior Trees also follows a complete interleaving semantics, as each thread operates fully in parallel with the others, unless synchronisation nodes are explicitly used. Despite this, the model that Nanda and Ramesh claim has a complete interleaving semantics is still not adequate for describing the threads in Behavior Trees. Their concurrency model has implicit synchronisation points at the ends of each thread. This arises due to the *co-begin* and *co-end* statements in their model; the former representing the point at which two or more threads begin and the latter the point at which they end and merge back to the parent thread. The *co-end* statements act as a synchronisation point between the threads, so each of the threads must finish their behaviour completely before control reverts back to the parent thread. This is not the case for Behavior Trees. There is no equivalent *co-end* location, since the threads may finish at any time, regardless of what stage the other threads have reached. For this reason, when an end node in a BT control flow graph is reached, it signals the termination of that particular thread only, not the entire system. The other threads can continue to execute. This is different to the CFG's of programs, in which the end node represents the termination of the whole program.

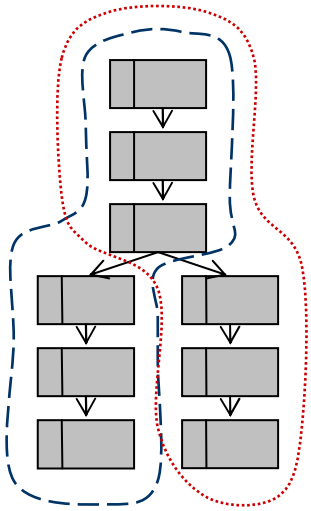


Figure 21. Threads in Behavior Trees

3.1.2 Alternative Branching

When programs contain an *if...elseif...else* construct, this can be modelled in a control flow graph using only two successors per conditional node, as shown in Figure 22.

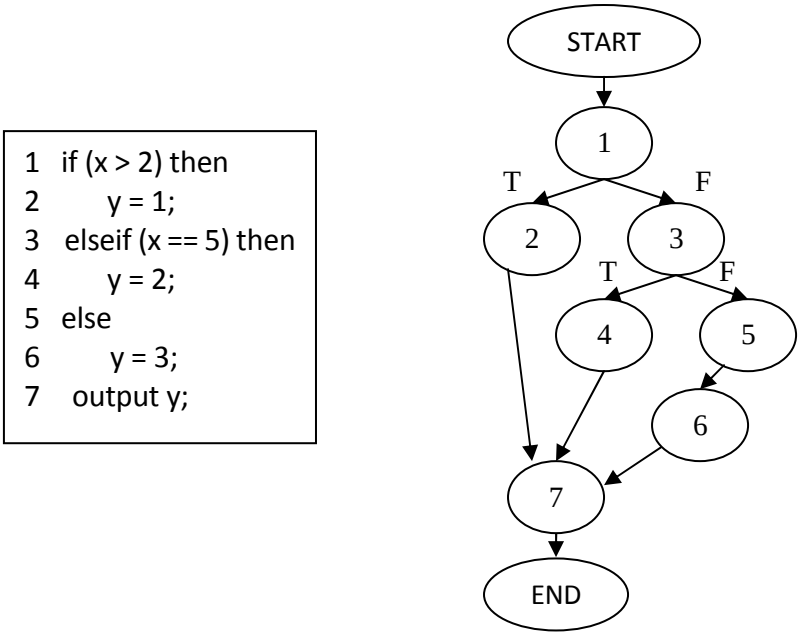


Figure 22. If-else Branching in Programs

However, alternative branching cannot be modelled in this way. The difference is in the order of evaluation of the conditions. For an *if* construct, each condition is only evaluated if the previous one failed. In an alternative branching group, the order in which the conditions will be evaluated is not specified. For this reason, alternative branching can result in a node having multiple successors.

3.2 BT Control Flow Graphs as Transition Systems

It is useful to interpret the execution of a BT control flow graph as a doubly-labelled transition system, in order to reason about the behaviour of the system in terms of the underlying states and

transitions. The interpretation of BT control flow graphs as transition systems as described here corresponds with the existing translation into SAL (Grunske, et al., 2008).

Recall from Section 2.1.1 that a doubly-labelled transition system is a tuple $T = (\mathcal{S}, AP, \mathcal{I}, \mathcal{L}, \longrightarrow)$. In a BT control flow graph, the nodes represent the transitions between states rather than the states. The states of the system are not explicitly represented. Nonetheless, for every BT control flow graph $G = \langle N,E, start, end\rangle$, a set of states $\mathcal{S}$ can be constructed, where each state represents the current evaluation of the system. Let $\mathcal{V}$ represent the set of uniquely-labelled variables in G, where each variable $v \in \mathcal{V}$ belongs to one of the following subsets: *Components*, *Attributes*, *Messages* or *SynchLabels*. The ranges of these types are as follows:

- If $v \in Components$, the range is $\{b \mid \exists\, m \in N$, where $comp(m) = v$ and $behav(m) = b\}$,
- if $v \in Attributes$, the range is $\{b \mid \exists\, m \in N$, where $attr(m) = v$ and $attrExp(m)$ involves a behaviour $b\}$, or
- if $v \in Messages$ or $v \in SynchLabels$, the range is $\{true, false\}$.

When BT control flow graphs are interpreted as doubly-labelled transition systems, each state is defined in terms of the atomic propositions which hold in that state, given by the labelling function $\mathcal{L}$. The atomic propositions are given as the current evaluation of variables in $\mathcal{V}$, denoted by pairs of variables and their values using the following notation: for each $s \in \mathcal{S}$, if a variable $v \in \mathcal{V}$ has the value *val*, then $(v, val) \in \mathcal{L}(s)$. The set of atomic propositions which hold in a given state is dependent on the nodes which have executed so far. Note that each node or block of atomic nodes may be able to execute in many different states. That is, for each node n, there exists a set of pairs of states (s,s') such that when the system is in state s, it is possible to execute n and result in state s'. This is due to concurrent branching. When a Behavior Tree is executing, several threads may be executing at once, each represented by a separate branch in the tree. If node n is about to execute, the only certainty is the current location of the thread which contains n; all other threads may have reached any location. Therefore, the overall system could be in one of several possible states. In a similar manner, the other various constructs of the Behavior Tree language, such as reference nodes and thread kills, could cause the system to be in any one of many different states when it is ready to execute a particular node.

If a node changes the value of a variable in the system, the state s before the node executes has a different labelling to the state s' after the node has executed, i.e. $\mathcal{L}(s) \neq \mathcal{L}(s')$. Otherwise, the states have the same labelling, i.e. $\mathcal{L}(s) = \mathcal{L}(s')$. The nodes which can change the value of a variable are state realisations, internal output and internal input nodes. If a state realisation node executes, it modifies a component or attribute. In the next state, that component or attribute has a new value but all other variables have the same value. Internal output nodes cause the message variable to change to *true* in the next state. Similarly, internal input nodes cause the message variable to change to *false*, to indicate that the message has been consumed. For all other nodes, such as selections, the purpose of the node is only to direct the control flow, not to change the state of the system. Therefore, the next state after the node has executed is identical to the previous state. This is described in Definition 10 below, where $\oplus$ is the function override operator. A function $updates(n)$ returns a set of variable and value pairs, which are the variables modified by n and their new values. Note that it is a singleton set.

DEFINITION 10. UPDATING STATES

If a node n executes in a state s, leading to a new state s', $\mathcal{L}(s) = \mathcal{L}(s') \oplus updates(n)$:

- if n is a state-realisation, where ($comp(n) = C$ and $behav(n) = b$) or ($attr(n) = C\_A$ and $attrExp(n) =$ "$C\_A := b$"), then $updates(n) = \{(C, b)\}$ or $updates(n) = \{(C\_A, b)\}$, respectively.
- if n is of type *internalOutput*, where m is the message being sent, $updates(n) = \{(m, true)\}$,
- if n is of type *internalInput*, where m is the message being sent, $updates(n) = \{(m, false)\}$ and
- for all other types, $updates(n) = \{\ \}$, i.e. $\mathcal{L}(s) = \mathcal{L}(s')$.

■

Following the conventions of model checking, the set $\mathcal{I}$ of initial states contains all possible states unless it is restricted. If the Behavior Tree begins with an atomic block of state realisations, $\mathcal{I}$ is restricted to the states in which those components and attributes have the given values. Otherwise, the initial states are only restricted by the root node. The set $\mathcal{I}$ can be left unrestricted by using a blank node as the root of the tree. The function *init*(*T*) returns the initialisation nodes for the given transition system *T*, i.e. either the root node or the atomic block of state realisations at the top of the tree.

Selection nodes and other conditional nodes dictate the direction of control flow based on whether their guard holds in the current state. The guard of a conditional node is an expression involving a variable of the system. If the expression holds in the current state, the guard holds. In this case, one of the *true* branches in the control flow graph is taken next; otherwise the *false* branch is taken. Definition 11 gives details of the possible evaluations of a guard in a particular state, given by a function *guard*(*n*, *s*): $(N \times \mathcal{S}) \rightarrow Bool$. The function returns *true* by default if the node has no guard.

DEFINITION 11. GUARDS OF CONDITIONAL NODES

For a node *n*, in a state $s \in \mathcal{S}$,

- if *n* is of type *BTguard* or *selection*, where *comp*(*n*) = *C* and *behav*(*n*) = *g*, then *guard*(*n*, *s*) = *true* iff $(C, g) \in \mathcal{L}(s)$.
- if *n* is of type *BTguard* or *selection*, where *attr*(*n*) = *C_A*, then *guard*(*n*, *s*) = *true* iff (*attrExp*(*n*), *true*) $\in \mathcal{L}(s)$.
- if *n* is of type *internalInput* or *externalInput* and *m* is the unique name of the message, then *guard*(*n*, *s*) = *true* iff (*m*, *true*) $\in \mathcal{L}(s)$.
- for all other types, *guard*(*n*, *s*) = *true*.

■

Similarly, synchronisation nodes have an associated guard, known as the *synchGuard*, to indicate whether or not all of the synchronising partners have executed. The reason it is considered separately to other types of guards is that a node can be both a synchronisation node and a BTguard, selection or input message node. In such cases, in a particular state, the node's guard may evaluate to a different value than its synchGuard. A function *synchGuard*(*n*, *s*): $(N \times \mathcal{S}) \rightarrow Bool$ is defined below, which returns *true* if all of a node's synchronising partners have executed and *false* otherwise. If the node is not a synchronisation node, the function returns *true*.

DEFINITION 12. SYNCHRONISATION NODES

For a node *n*, in a state $s \in \mathcal{S}$,

- if *n* is a synchronisation node, then *synchGuard*(*n*, *s*) = *true* iff all of *n*'s synchronising partners have executed; *false* otherwise.
- if *n* is not a synchronisation node, then *synchGuard*(*n*, *s*) = *true*.

■

As seen so far, the current state is dependent on the sequence of nodes that have executed. Obviously, not every sequence is allowable for a given BT control flow graph. From each state, only a small set of nodes are permitted to execute. These are the nodes that have been reached so far in each thread. In a given state *s* and a thread *t*, the nodes which are ready to execute next are given by a function $ready_t(s)$: $N \rightarrow 2^N$. Definition 13 defines this. In general, when a node *n* executes, the nodes which can execute next are its immediate children. However, this differs if *n* has a guard or synchGuard which evaluates to *false* at *s*. For the synchronising case, *n* remains as the next node to execute, since it is still waiting for its synchronising partners to be reached. Recall that if a synchronisation node is also a conditional node, its condition must only be evaluated when all of its synchronising partners are ready to execute. Therefore, the node's guard is only considered when its synchGuard evaluates to *true*.

In a given state, if a node's synchGuard is *true*, but *n* has a guard that evaluates to *false*, the children reached via *n*'s *false* branch become the next to execute. In the case of selections, the next node to

execute will be an end node, whereas for BTguards and input events, it will be *n* itself. If the guard is *true*, i.e. *n* is free to execute, then its immediate children via the *true* edge are chosen next, unless *n* is a reversion or reference node. Additionally, if *n* is a thread kill node, the threads of its target node are terminated, so no more nodes will execute in those threads. The function $ready_t(s)$ returns an empty set for those threads.

If *n* is a reversion or reference node, the next nodes to execute are the immediate children of its target. This allows the control flow to jump to the new location. Recall that threads in BT control flow graphs start at the root and extend down to the leaf nodes. Therefore, the target node belongs to the threads of all of its descendents. After the reversion, all of these threads will be ready to execute the target node's children. As an example, consider the BT control flow graph in Figure 23. Assume that thread 2 has just executed node 7. Assume thread 1 then executes the reversion to node 1. After the reversion, the next node to execute in both threads is node 2.

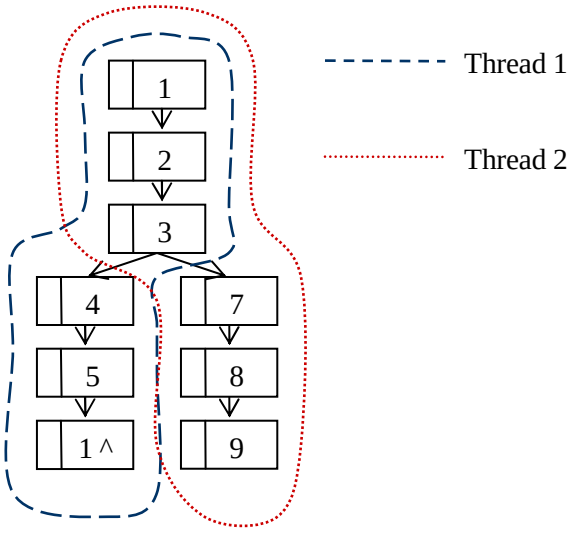


Figure 23. Execution of a Reversion

Atomic blocks are handled in a similar fashion. The only difference is that all nodes in the atomic block must execute before control passes to the children of the block. Therefore, if there are any synchronisation or guard nodes whose guards evaluate to *false* at *s*, the entire block cannot execute.

DEFINITION 13. CONSTRUCTION OF THE READY FUNCTION.
The *ready* function is constructed as follows:

For all states $s_i \in \mathcal{I}$, for all threads *t*, $ready_t(s_i) = \{root\}$.

For all other states $s' \in \mathcal{S}$, let *n* be the node executed in state *s* to reach state *s'*, where $s \in \mathcal{S}$. Then:

(i) if $synchGuard(n, s) = false$, then $ready_t(s') = ready_t(s)$,

(ii) if (i) does not hold and $guard(n, s') = false$, then
$ready_t(s') = \{m \mid parent(m) = n$ and $label(edge(n, m)) = false\}$,

(iii) if (i) and (ii) do not hold and *n* is a *thread kill* node, then for all threads *t* such that
$t \in threads(target(n))$, then $ready_t(s') = \{\}$,

(iv) if (i) and (ii) do not hold and *n* is a *reversion* or *reference* node, then for all threads *t* such that $t \in threads(target(n))$, $ready_t(s') = \{m \mid parent(m) = target(n)$ and $label(edge(target(n), m)) = true\}$ and

(v) if (i), (ii) and (iv) do not hold, then for all threads t such that $t \in threads(n)$, $ready_t(s') = \{m \mid parent(m) = n \text{ and } label(edge(n, m)) = true\}$.

If, instead of a single node n, a block of atomic nodes $b = \{n_1, n_2, ..., n_k\}$ were executed in state s to reach s', where n_1 is the top node of the block and n_k is the last node, then each of the statements (i) to (iv) must be considered in that order. For each statement, if any $n_i \in b$ satisfies the condition of the statement, then that statement must be applied to n_i. If no node in the atomic block satisfies any of the conditions in statements (i), (ii) and (iv), then statement (v) must be applied. However, statement (v) must be modified to:

If (i), (ii) and (iv) do not hold, then for all threads t such that $t \in threads(n_i)$, $ready_t(s') = \{m \mid parent(m) = n_k \text{ and } label(edge(n_k, m)) = true\}$.

■

Example.

Consider the control flow graph shown in Figure 24. There are three threads, labelled 1, 2 and 3. Thread 1 consists of the nodes A[a], B[b] and G[g]. Thread 2 consists of A[a], C[c] and D???d???. Thread 3 consists of A[a], C[c], E[e] and the A[a] reversion.

At an initial state $s_0 \in \mathcal{I}$, the ready sets for all three threads contain only the root node, A[a].

Initial state: $ready_1(s_0) = \{A[a]\}$, $ready_2(s_0) = \{A[a]\}$, $ready_3(s_0) = \{A[a]\}$.

After A[a] executes, at state s_1, there is a choice between executing B[b] and C[c] next. The ready set for thread 1 will contain B[b] and the ready sets for both other threads will contain C[c].

After A[a] executes: $ready_1(s_1) = \{B[b]\}$, $ready_2(s_1) = \{C[c]\}$, $ready_3(s_1) = \{C[c]\}$.

Assume that C[c] is chosen, reaching state s_2. Then the ready set for thread 1 will remain unchanged, while the ready set for thread 2 will be updated to D??d??? and the ready set for thread 3 will be updated to E[e].

After C[c] executes: $ready_1(s_2) = \{B[b]\}$, $ready_2(s_2) = \{D???d???\}$, $ready_3(s_2) = \{E[e]\}$.

Assume D???d??? is chosen next, reaching state s_3, and assume that the condition does not hold. Then the ready set for thread 2 will be updated to contain D???d??? again, as it is a child of itself, and the other ready sets will remain unchanged.

After D???d??? executes: $ready_1(s_3) = \{B[b]\}$, $ready_2(s_3) = \{D???d???\}$, $ready_3(s_3) = \{E[e]\}$.

Next, assume the node E[e] is chosen, reaching state s_4. The ready set for thread 3 will be updated to the reversion node.

After E[e] executes: $ready_1(s_4) = \{B[b]\}$, $ready_2(s_4) = \{D???d???\}$, $ready_3(s_4) = \{A[a]^\wedge\}$.

Then assume the reversion node A[a]^ executes, reaching state s_5. The ready sets for all three threads are updated. The ready set for thread 1 is updated to contain B[b], because the thread was terminated and then re-started. The ready sets for threads 2 and 3 are both updated to contain C[c], as these two threads were both terminated and C[c] is ready to execute next.

After A[a]^ executes: $ready_1(s_5) = \{B[b]\}$, $ready_2(s_5) = \{C[c]\}$, $ready_3(s_5) = \{C[c]\}$.

■

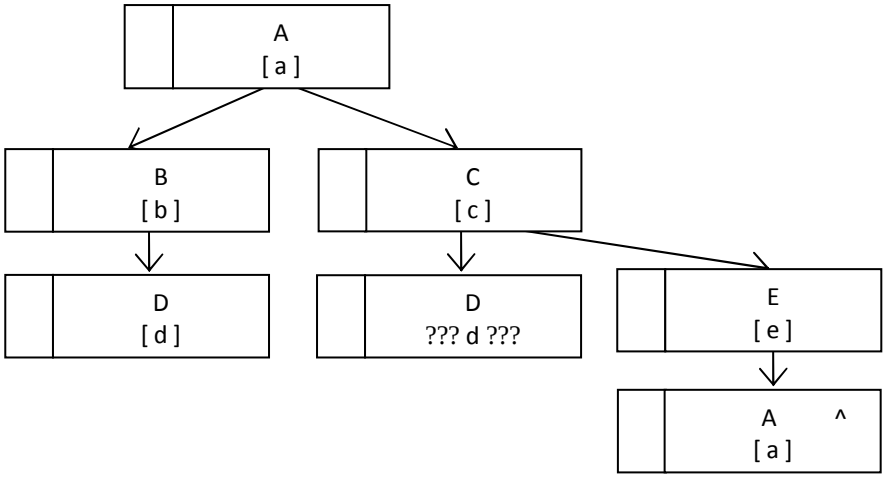


Figure 24. Example to illustrate ready sets.

Using these definitions, a BT control flow graph can be represented as a doubly-labelled transition system $T = (\mathcal{S}, AP, \mathcal{I}, \mathcal{L}, \mathcal{N}, \longrightarrow)$, where $\mathcal{S}$ is a set of states, AP is a set of atomic propositions, $\mathcal{I}$ is a set of initial states, $\mathcal{L}$ is a labelling on states, $\mathcal{N}$ is a set of actions representing nodes and $\longrightarrow : \mathcal{S} \times \mathcal{N} \times \mathcal{S}$ are the transitions. As a short-hand, the notation $s \xrightarrow{n} s'$ corresponds to $(s, n, s') \in \longrightarrow$, which corresponds to the execution of the node n.

A *run* $\rho = < s_0, n_1, s_1, n_2, s_2 ... >$ in a doubly-labelled transition system T is a sequence of alternating states and nodes, starting and ending at states. A *path* $\pi = <s_0, s_1, ..., s_k>$ is a sequence of states derived from a run by removing all the nodes from the sequence. Similarly, an *execution trace* $\sigma = < n_0, n_1, ..., n_{k-1}>$ is a sequence of nodes derived from a run by removing all the states from the sequence. The function *run*(π) takes a path as an argument and returns the corresponding run. The function *runs*(T) returns the set of all runs in the doubly-labelled transition system T, while *paths*(T) returns the set of all paths and *traces*(T) returns the set of all execution traces. The function *preTraces*(s) returns the set of traces $\sigma = < n_0, n_1, ..., n_{k-1}>$ that correspond to a run $\rho = < s_0, n_1, s_1, n_2, s_2 ... n_k, s>$, i.e. the traces which cause the system to reach state s. The notation $\rho\langle\!\langle s_i]$ denotes the prefix of the run ρ ending at (and including) s_i, while $\rho[s_i\rangle\!\rangle$ returns the suffix of ρ starting at (and including) s_i. The same notation will be used for paths and execution traces.

3.3 Dependencies

After the control flow graph has been constructed, the next step is to create a *dependence graph* from the control flow graph. A dependence graph is a directed graph $G = <N, E>$, where N is a set of nodes and $E = N \times N$ is a set of edges. For an edge $(n_i, n_j) \in E$, the notation $n_i \rightarrowtail n_j$ is also used, indicating a dependency from n_i to n_j. That is, node n_j *depends* on n_i. Unlike control flow graphs, edges in a dependence graph can link two nodes from different threads. The dependence graph is created by identifying the various dependencies between nodes. A path π in a dependence graph is a sequence of nodes such that for every pair of consecutive nodes in the sequence $\{n_i, n_{i+1}\} \in \pi$, $n_i \rightarrowtail n_{i+1}$, where d is a label identifying the type of dependency. The types of dependencies are *control*, *data*, *interference*, *message*, *synchronisation* and *termination* dependencies. The definitions for each type of dependency are given in Sections 3.3.1 to 3.3.6. These definitions utilise the notions of the Definition Set (*DEF(n)*) and Reference Set (*REF*(*n*)). Informally, *DEF*(*n*) contains all the components and attributes that are defined or modified at node *n* and *REF*(*n*) contains all the components and attributes that are referenced at *n*. In the following definitions, assume that C is a component, s is a

behaviour name, n is a Behavior Tree node, a and b are attributes of C, S and T are sets and x is an element of S.

DEFINITION 14. DEFINITION SET
Let *DEF*(*n*) represent the set of components and attributes defined at node *n*. Specifically, if the node is of the form:

i) (state-realisation) C [s], then C ∈ *DEF*(*n*),
ii) (state realisation of attributes) C [a := s], then C_a ∈ *DEF*(*n*),
iii) (adding/ removing an element from a set) C [S := S + x] or C [S := S - x], then S ∈ *DEF*(*n*),
iv) (union/ subtraction/ intersection of sets) C [S := S + T] or C [S := S - T] or C [S := S >< T], then S ∈ *DEF*(*n*).

■

DEFINITION 15. REFERENCE SET
Let *REF*(*n*) represent the set of components and attributes referenced at node n. Specifically, if the node is of the form:

i) (selection/guard) C ?s? or C ???s???, then C ∈ *REF*(*n*),
ii) (selection/guard over attributes) C ?a = exp? or C ???a = exp???, where exp is an expression or a behavior, then C_a ∈ *REF*(*n*),
iii) (state realisation of attribute) C [a := f(b)], where f(b) is an expression over b, then b ∈ *REF*(*n*),
iv) (selection over set predicates) C ?x : S? or C ?S = {}? or C ?S ⋈ m?, where ⋈ ∈ {=, >, <, ≤, ≥}, then S ∈ *REF*(*n*).

■

3.3.1 Control Dependence

Control dependence occurs when one Behavior Tree node controls whether or not another node will be executed. The definition for control dependence is as follows.

DEFINITION 16. CONTROL DEPENDENCE
For two nodes *p* and *q* in a control flow graph, node *q* is control-dependent on node *p*, denoted as ($p \xrightarrow{cd} q$), iff node *p* has at least two successors *m* and *n*, such that $p \neq m$, where:

- *label*(*edge*(*p*,*n*)) = *false* and
- $\exists \pi \in maxTraces(m)$ such that $q \in \pi$ and $\forall r \in \pi$, where $r \neq m$ and $r \neq q$, for all edges *e* from *r*, *label*(*e*) = *true*.

■

This definition captures the usual meaning of control dependence. A node *q* is control-dependent on a node *p* if there are two possible outcomes after executing *p*: in one scenario *q* is reached, and in the other scenario *q* is not reached. The definition requires that there are at least two successors, *m* and *n*. The requirement that *m* cannot be the same node as *p* ensures that no node can have a control dependency to itself. The first criterion is that *n* must be reached via a *false* edge. Since *false* edges only reach end nodes or a loop, this implies that there is a trace from *p* on which *q* is never reached. Note that *n* may be *p* itself, as a *false* edge may loop back to *p*. The second criterion is that there is a maximal path from *m* on which *q* occurs and none of the other nodes on the path have a *false* edge. This ensures that none of those nodes can induce a control dependency to *q* as well.

The traditional definition of control dependence is unsuitable because BT control flow graphs do not necessarily have a single end node. The new control dependence definitions of Ranganath et al.

(2007) are designed for non-terminating systems. However, those definitions are also unsuitable for BT control flow graphs due to concurrent and alternative branching.[††] Ranganath et al.'s *non-termination sensitive control dependence* (see Section 2.2.2 on page 18), requires there to be a path from one of the successors of *p* on which *q* never occurs or *p* always precedes any occurrence of *q*. Using the same criterion for BT control flow graphs induces control dependencies from branching nodes to *all* of their descendents. To see this, let *p* be a node with more than one child due to branching and *q* be one of its descendents. Since there is more than one path from *p*, it will always be possible to find a path on which *q* never occurs, by following one of the other branches. Although the descendents are indeed controlled by the decision of which branch is taken, they are not actually dependent on the branching node *p* itself. Even if *p* was removed, the system would have the same behaviour, since the closest ancestor of *p* in the slice would become the new branching node.

Example.
Consider the BT control flow graph in Figure 25. Even though the execution of the node C[c] is dependent on the choice made after the branching node B[b], the node B[b] itself is not the controlling

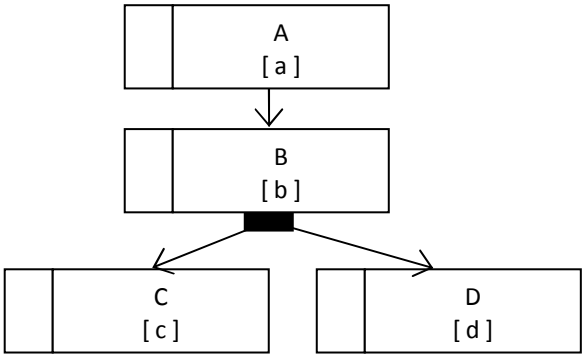


Figure 25. BT Control Flow Graph with Branching

element. If B[b] was removed and the branching node became A[a], the behaviour of C[c] will remain the same.

■

Ranganath et al.'s definition (2007) requires that *for all* paths from one of *p*'s successors, *q* always occurs or always precedes any occurrence of *p*. This requirement is too strong for BT control flow graphs. There may be a path to *q* which contains alternative or concurrent branches. Following these branches would lead to paths that do not reach *q*, even though *p* may still be controlling whether or not *q* executes. For this reason, the second requirement of Definition 16 only requires that *there exists* a path leading to *q*.

Example.
Consider the BT control flow graph in Figure 26. The node B?b? has two successors, C[c] and the end node. The end node satisfies the first criterion of control dependency, as it is reached via a *false* edge. From C[c], there is a path on which D[d] occurs and another on which it does not occur, i.e. the E[e] branch. Therefore, if the requirement was that *all* paths from C[c] must reach D[d], the requirement would not be satisfied, so there would not be a control dependency from D[d] to B?b?. Despite this, the selection node does indeed control whether or not D[d] can execute. This can be identified by requiring only that *there exists* a path from C[c] that reaches D[d]. Similarly, B?b? controls E[e].

■

[††] The dependencies that arise from alternative branching are captured in *termination dependence*, given in Section 3.3.6.

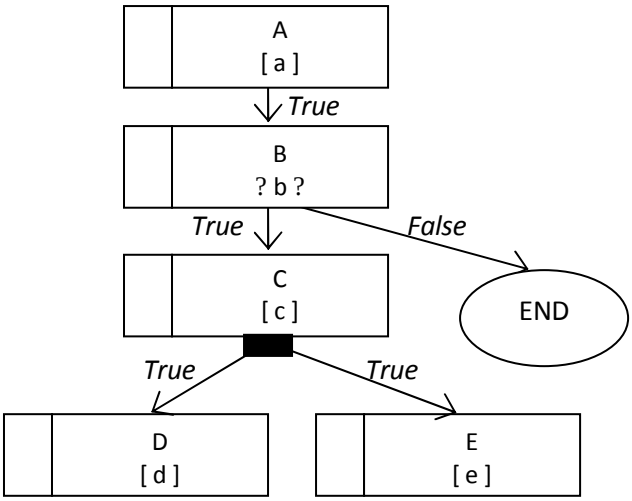


Figure 26. A Controlling Node with Branching Descendents

Non-termination

The term *maximal* requires the trace to either end at a leaf node or contain an infinite loop. Most leaf nodes in Behavior Trees are either reversions or reference nodes, which cause the control flow to revert to another location. However, this change in control flow is not represented as an edge in the BT control flow graph. Therefore, the maximal paths will end at the leaf nodes. The implication of this is that a node *m* can never control an ancestor, *n*, even if the ancestor can be reached via a reversion or reference node. In this case, *m* is not actually controlling whether or not its ancestor executes, but whether it executes *for a second time*. The first time *n* is reached, it will execute regardless of the controlling node *m* below. The node *m* is actually controlling the reversion or reference node, which in turn dictates whether or not *n* will be reached on a subsequent iteration. This does not result in any difference in the final slice. As will be seen in Section 3.4.3, if node *n* is in the slice, the reversion or reference node below will be included as well. Due to its control dependency to *m*, this will in turn result in the inclusion of *m*.

This can be thought of as similar to Nanda and Ramesh's (2000, 2006) approach of differentiating between normal data dependence and *loop-carried data dependence*, which is data dependence arising from a previous iteration of a loop, although their approach does not address control dependencies induced by loops. One advantage of considering control dependence in this way is that it is computationally easier to explore paths up to the leaf nodes only, without following the paths created by reversions or reference nodes. Another advantage will be seen in Chapter 4, for identifying paths which are *infeasible*.

Example.
Consider the Behavior Tree in Figure 27. The guard D???d??? controls the reversion but not its ancestors. If the guard is not satisfied, it would only prevent its ancestors from executing on future iterations, not the first time.

■

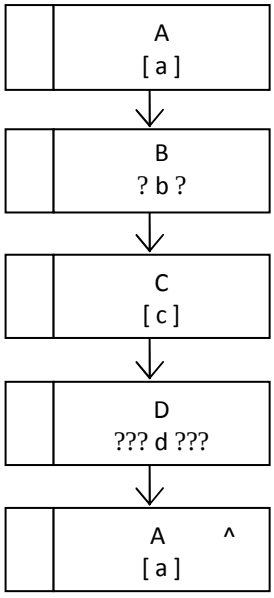


Figure 27. Control Dependency to a Reversion.

The goal of *non-termination sensitive control dependence* (Ranganath, et al., 2007) is to identify nodes which control possibly infinite loops. In Behavior Trees, the loops created by reversion nodes are often not controlled by any conditional nodes. Thus, control dependency is not sufficient for identifying all the reversions and references which are necessary. Section 3.4.3 will present a method for identifying relevant reversions and reference nodes without requiring control dependence.

Types of Controlling Nodes

Using Definition 16, the only nodes that can induce control dependencies are those with an outgoing *false* edge. These are either: selections, guards, synchronisations or input events, referred to collectively as conditional nodes. This produces a useful result: that every descendent of a conditional node is either directly or transitively control dependent on it. The reasons for this are simple: all nodes are control dependent on their nearest conditional ancestor, which will in turn be control dependent on their nearest conditional ancestor and so on. The following lemma demonstrates this. This result will be utilised for the proof of correctness for slicing, presented in Section 3.6 on page 84.

LEMMA 1. ESTABLISHING CONTROL DEPENDENCE USING GUARDS

For any node n_x such that $conditional(n_x)$, $\forall n_y \in desc(n_x)$, $n_x \xrightarrow{cd^+} n_y$.

Proof.

(By induction over the number of conditional nodes between n_x and n_y).

Base Case: n_x is the closest ancestor of n_y which is conditional,

i.e. $\forall n_i \mid n_i \in desc(n_x) \wedge n_i \in ances(n_x)$, NOT($conditional(n_i)$). .....(1)

$conditional(n_x)$

$\Rightarrow$ n_x has two successors n_i and n_j, where $label(edge(n_x,n_j)) = false$ .....(2)

$n_y \in desc(n_x)$,

$\Rightarrow \exists \pi \in maxTraces(n_x)$ such that $n_y \in \pi$.

From (1), $\forall n_k \in \pi$, where $n_k \neq n_x$, for all edges e from n_k, $label(e) = true$. .....(3)

From (2) and (3), and by the definition of control dependence, $n_x \overset{cd}{\rightarrowtail} n_y$.

Induction Step: n_x is not the closest ancestor of n_y which is conditional. Let n_z be the closest ancestor of n_y that is a conditional node. Since $n_z \in desc(n_x)$, assume that $n_x \overset{cd^+}{\rightarrowtail} n_z$.

conditional(n_z)
$\Rightarrow$ n_z has two successors n_i and n_j, where *label*(*edge*(n_z,n_j)) = *false*. .....(4)
$n_y \in desc(n_z)$,
$\Rightarrow \exists \pi \in maxTraces(n_z)$ such that $n_y \in \pi$.
Since n_z is the closest ancestor of n_y such that *conditional*(n_z),
$\forall n_k \in \pi$, where $n_k \neq n_x$, for all edges e from n_k, *label*(e) = *true*. .....(5)
From (4) and (5), and by the definition of control dependence, $n_z \overset{cd}{\rightarrowtail} n_y$.
Therefore, $n_x \overset{cd^+}{\rightarrowtail} n_y$.

□

3.3.2 Data Dependence

Data dependence is defined in the same way as for programs. A node is data dependent on another if it refers to the state of a variable (component or attribute) that the other node defines or updates. For example, a selection node Button ?pushed? would be data-dependent on a state realisation node Button [pushed] or even a node Button [released]. If there is a component or attribute *c* that is in the set *REF*(*q*), then the node *q* is data-dependent on any node *p* for which *c* is in the set *DEF*(*p*), as long as *c* is not re-defined by another node on the path between *p* and *q*. This dependence refers only to two nodes in a single thread. Data dependence between nodes in parallel threads is referred to as *interference dependency* and is covered in the next section.

DEFINITION 17. DATA DEPENDENCE.
For two nodes *p* and *q* in a control flow graph, node *q* is data-dependent on node *p*, ($p \overset{dd}{\rightarrowtail} q$), iff:

- $\exists c \in DEF(p)$ such that $c \in REF(q)$,
- $\exists \pi = trace(p, q)$, where $\forall k \in \pi$, $c \notin DEF(k)$ and
- $\neg(conc(p, q))$.

■

3.3.3 Interference Dependence

Interference dependence is the same as data dependence except that the two nodes involved are in parallel threads. For example, in Figure 24 on page 51, the node D???d??? is interference-dependent on the node D[d]. It is differentiated from data dependence because interference dependence is intransitive, unlike the other dependency types. Due to this, if a slice is created simply by following transitions in the dependency graph, the resulting slice may be imprecise, containing unnecessary nodes. This concept is covered in further detail in Chapter 4. The definition for interference dependence contains the same requirements as for data dependence, except that the nodes must be in parallel threads and therefore are not required to be connected by a path in the control flow graph.

DEFINITION 18. INTERFERENCE DEPENDENCE.
For two nodes *p* and *q* in a control flow graph, node *q* is interference-dependent on node *p*,
($p \overset{id}{\rightarrowtail} q$), iff:

- $\exists\, c \in DEF(p)$ such that $c \in REF(q)$ and
- $conc(p, q)$. ■

3.3.4 Message Dependence

Message dependence is very similar to data dependence, except that it arises from internal input and output nodes. Each internal input node is message-dependent on all internal output nodes that send the message it is waiting to receive. For example, an internal input node controller >lowAir< would be message-dependent on an internal output node sensor <lowAir>. There may be multiple senders and multiple receivers. As with interference dependence, message dependence is intransitive because it can occur between parallel threads. External input and output message nodes do not induce message dependencies, since they represent interactions with the environment, not with other nodes in the control flow graph.

DEFINITION 19. MESSAGE DEPENDENCE.

For two nodes p and q in a control flow graph, node q is message-dependent on node p, ($p \xrightarrow{md} q$) iff:

- $type(p) = intOutput$ and $behav(p) = m$ and
- $type(q) = intInput$ and $behav(q) = m$. ■

Message dependence is similar to Labbé et al.'s notion of *communication dependence* (2007) for communicating automata specifications, which describes communication occurring between two automata via channels. The *interference control dependence* of Luangsodsai and Fox (2010), used for slicing statecharts, also performs a similar purpose to message dependence. Interference control dependence occurs when an event in a statechart is triggered by a parallel action. This can be seen as similar to an output message triggering an input message in a parallel thread.

3.3.5 Synchronisation Dependence

Synchronisation dependence refers to the dependence between a group of synchronising nodes. For example, if three nodes, all labelled A[a] but in different threads, are synchronising with each other, each will be synchronisation-dependent on each of the others. Synchronisation dependence is thus symmetric. Note that since synchronising nodes have two successors in the control flow graph, a synchronising node induces a control dependence on its descendents. Additionally, the synchronising node is itself dependent on its synchronising partners. The result of this is that the descendents of a synchronisation node are transitively dependent on all of the synchronising partners.

Although synchronisation dependence occurs between parallel threads, it does not suffer from the intransitivity problem. Intransitivity of message and interference dependence occurs when a node m is message or interference-dependent on a node n in another thread, which is in turn message or interference-dependent on a node p in the first thread, where p cannot execute before m. In such a case, m cannot be dependent on p and it is not necessary to include p in the slice. In contrast, if a node m is synchronisation-dependent on a node n in a second thread, n cannot be synchronisation-dependent on another node in the first thread, since there cannot be two synchronising partners from the same thread.

DEFINITION 20. SYNCHRONISATION DEPENDENCE.

For two nodes p and q in a control flow graph, node q is synchronisation-dependent on node p, ($p \xrightarrow{sd} q$) iff:

- $flag(p) = synch$ and $flag(q) = synch$ and
- $matching(p,q)$. ■

3.3.6 Termination Dependence

All of the dependencies discussed so far are ones that *enable* the dependent node to execute. For instance, a node n_2 is control-dependent on a node n_1, then n_2 needs n_1 in order to execute. If a node n_2 is data-dependent on a node n_1, then n_1 may potentially create the conditions under which n_2 can execute. However, there are some types of nodes that always *prevent* other nodes from executing. Thread kill nodes are the most obvious of these. A thread kill node terminates the thread that it is referring to. Thus, any node in that thread can suddenly be terminated during its operation.

When a reversion node executes, all threads that were started after the top reversion point are terminated immediately (see Section 2.3). Every node in all of these sub-threads are therefore dependent on the reversion node. For example, in the Behavior Trees shown in Figure 24, node B[b] is termination-dependent on the reversion node A[a], since B[b] belongs to a thread that will be terminated if the reversion executes.

The final dependency type in this category is produced as a result of alternate branching points. If one of the root nodes of the branches executes, all other branches are terminated. Thus, every node in each branch is dependent on the root nodes of the other branches. For example, in the Behavior Tree shown on the left of Figure 28, both nodes C[c] and P[p] are termination-dependent on D[d]. The node D[d] is termination-dependent on C[c].

DEFINITION 21. TERMINATION DEPENDENCE.
For two nodes p and q in a control flow graph, node q is termination dependent on node p, ($p \xrightarrow{td} q$) iff:

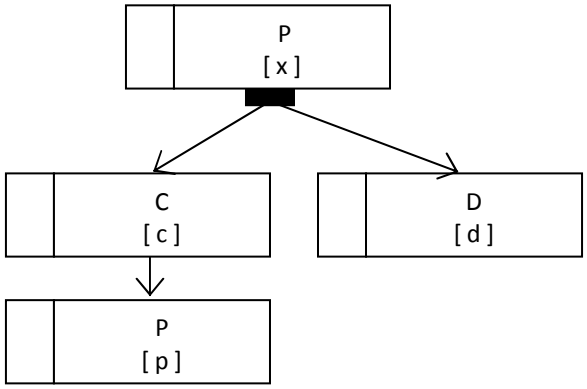
- ($type(p) = threadKill$) and $q \in desc(target(p))$ or
- ($type(p) = reversion$) and $q \in desc(target(p)) \wedge p \notin desc(q)$ or
- ($alt(p,q)$) and $q \in desc(parent(p))$.

■

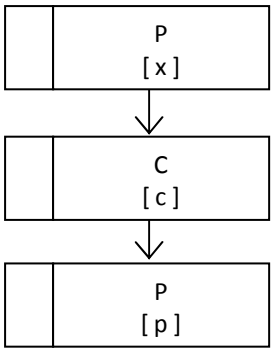
In other words, a node q is thread-termination dependent on a node p if and only if p is either:
i) a thread kill node which terminates q's thread, or
ii) a reversion node which terminates q's thread, unless it is a descendent of q, or
iii) a root node of a branch in an alternative branching set, where q belongs to another branch in the set.

Example.

The following example illustrates the importance of termination dependence. Assume that the theorem to be verified is G(F(P = p ∧ C = c)). In other words, it is always the case that eventually the component P has a value p and the component C has a value c. Consider the Behavior Tree in Figure 28. The Behavior Tree on the left is the original tree. If termination dependencies were not used for computing the slice, the node D[d] would be removed, producing the slice shown on the right. However, the two trees behave very differently. The theorem holds on the slice, while it does not hold on the original tree. In the slice, the node P[p] is always eventually reached, but in the original tree, if the D[d] branch is chosen, P[p] will never execute since its branch is terminated. This example illustrates the need for a dependency type that describes terminating behaviour.



Original Behavior Tree



Slice without termination dependence.

Figure 28. Example of slicing without termination dependence.

3.4 Creating the Slice

After the dependence graph has been created, the final step is to produce the slice based on a given slicing criterion. In traditional program slicing, the slicing criterion is a set of variables and a statement of the program. The goal is to determine the values of the given variables at that point in the program. When slicing for verification, however, the goal is to determine the validity of a temporal logic property. This means that there may be more than one node to be used as the starting point for traversing the dependence graph. The slicing criterion is extracted from the temporal logic theorem which is to be verified. The criterion is a set consisting of every node which modifies the state of one of the variables mentioned in the theorem. For Behavior Trees, this amounts to every state realisation or set operation which updates a variable in the theorem. In the following, for a temporal logic theorem φ, the theorem φ is said to *contain* a variable *v* if *v* is mentioned in the formula. The slicing criterion is defined with respect to a given CTL^*_{-X} formula φ, as given by the definition below. The nodes in the slicing criterion will be referred to henceforth as *criterion nodes*.

DEFINITION 22. SLICING CRITERION FOR BEHAVIOR TREE SLICING

For a transition system $B = (\mathcal{S}, AP, \mathcal{I}, \mathcal{L}, \mathcal{N}, \longrightarrow)$ of a BT control flow graph,
the slicing criterion C_φ with respect to a formula $\varphi \in CTL^*_{-X}$, is defined as:
$C_\varphi = \{n \mid \exists\, v \in DEF(n)$, where φ contains $v\}$.

■

The slice is created in several stages. In the first phase, a simultaneous backward static slice is generated using the criterion nodes. The nodes in the criterion set form the starting points for the backwards traversals of the dependency graph. In other words, the set of nodes that the criterion nodes depend on, either directly or transitively, are located. Using each of the criterion nodes as the starting points, the dependency graph is traversed in reverse, collecting every node that is encountered via dependency edges. The algorithm checks whether a node has previously been encountered before adding it to the slice, in order to prevent infinite cycles caused by cyclic or symmetric dependencies. The set of nodes encountered by the traversals of the dependency graph is referred to as the *slice set*, given by Definition 23 below.

DEFINITION 23. SLICE SET

For a transition system $B = (\mathcal{S}, AP, \mathcal{I}, \mathcal{L}, \mathcal{N}, \longrightarrow)$ of a BT control flow graph,

the slice set $nodes\_slice_{\varphi}(B)$ is defined as:
$nodes\_slice_{\varphi}(B) = \{n_x \mid n_x \overset{d}{\rightarrowtail} n_c$, for some $n_c \in C_{\varphi}\}$, where $d \in \{cd, dd, id, md, sd, td\}$.

■

The second phase involves identifying which reversion and reference nodes to add back into the slice. These reversion and reference nodes are then used as the starting points for another reverse exploration of the dependence graph, in order to locate any further dependencies.

Finally, the nodes collected so far are re-formed into a syntactically correct Behavior Tree, forming the slice. This stage involves including additional place-holder nodes.

3.4.1 Observable vs. Stuttering Nodes

The criterion nodes, i.e. the nodes which directly modify a variable in the temporal logic formula, are known as *observable*. All other nodes are referred to as *stuttering* nodes. The function $obs_{\varphi}(n)$ returns true if and only if the node n is observable. The stuttering nodes include the nodes which are reached during the backwards traversal, so while all observable nodes *must* be included in the slice, stuttering nodes may or may not be included in the slice.

For a doubly-labelled transition system $T = (\mathcal{S}, AP, \mathcal{I}, \mathcal{L}, \mathcal{N}, \longrightarrow)$ representing a BT control flow graph, the labelling $\mathcal{L}$ maps each state to the set of atomic propositions (AP) which hold in that state. A slice must exhibit the same behaviour as the original model in terms of only a subset of AP. This subset, denoted AP_{φ}, consists of all the atomic propositions (v, val) such that v is a variable contained in the formula φ. To enable a control flow graph to be viewed in terms of this subset of atomic propositions only, the labelling on states can be restricted as well. The notation $\mathcal{L}_{\varphi}(s)$ is used to denote the label of the state s restricted to the atomic propositions in AP_{φ}. That is, $\mathcal{L}_{\varphi}(s) = \{(v, val) \mid (v, val) \in AP_{\varphi}\}$.

The notation $s \dashrightarrow s'$ denotes the execution of a stuttering node in state s, leading to the state s'. The transitive closure of this is denoted as $s \dashrightarrow^{*} s'$. If $s \dashrightarrow s'$ in the context of φ, then $\mathcal{L}_{\varphi}(s) = \mathcal{L}_{\varphi}(s')$. The reverse holds as well: if $\mathcal{L}_{\varphi}(s) = \mathcal{L}_{\varphi}(s')$ and s' was reached from s by a single step, then it must have been a stuttering step. The notation $s \dashrightarrow^{j} s'$ denotes that j stuttering steps are taken after s to reach state s'.

An execution trace in a transition system $\sigma = < n_0, n_1, ..., n_k >$ can be viewed in terms of observable nodes only, referred to as an *observable execution* trace, described in Definition 24 below. The observable execution trace is obtained using a notion of projection, similar to other approaches such as by Hatcliff et al. (2000).

DEFINITION 24. OBSERVABLE EXECUTION TRACE

An *observable execution trace* σ_{φ}, of an execution trace σ, is given by $\sigma_{\varphi} = proj_{\varphi}(\sigma)$, where:

- $proj_{\varphi}(<>) = <>$,
- $proj_{\varphi}(< n_0, n_1, ..., n_k >) = < n_0 > ^\frown proj_{\varphi}(< n_1, ..., n_k >)$, if $n_0 \in C_{\varphi}$ and
- $proj_{\varphi}(< n_0, n_1, ..., n_k >) = proj_{\varphi}(< n_1, ..., n_k >)$, if $n_0 \notin C_{\varphi}$.

■

3.4.2 Blank Nodes

As will be seen in the following sections, sometimes a node may need to be included in the slice for the sole purpose of maintaining the correct tree structure, for example when a group of child nodes require a common parent. In these cases, the details of the node are unnecessary, so a *blank* place-holder node may be used instead. These are nodes which have no data, i.e. they have no associated component or behaviour, and they do not cause any action to be executed. The advantage of using blank nodes instead of simply preserving the original node is that blank nodes have no dependencies. The original node may have a long chain of dependencies, all of which are unnecessary in the slice.

This situation only occurs for state realisations which are dependent on an attribute. If the original node was a conditional or synchronisation node, it would have induced a control dependency on its descendents and would therefore be in the slice already. If it had been an output message node (either external or internal), or a state realisation with no attributes, it would not have any dependencies other than a control dependency to an ancestor. Since the node's descendents are in the slice, they would also have the same control dependency, so that ancestor would be in the slice as well. Therefore, the only type of node which can introduce additional dependencies is state realisations with attributes. Since the state realisation was not originally included in the slice, it is irrelevant to the slicing criterion, and therefore the attribute's value is also irrelevant. By replacing the state realisation with a blank node, the unnecessary dependencies will be ignored. Blank nodes are positioned in the tree at a specific location, given by the *parent* and *children* attributes in the function *blank*:

blank(*parent*, *children*),

where *parent* is the node that will become the parent of the blank node, and *children* is the set of child nodes which will now have the blank node as their parent. Blank nodes have no corresponding *updates* or *guard*. If a node *n* is a blank node then *isBlank*(*n*) returns *true*; *false* otherwise.

3.4.3 Reversion and Reference Nodes

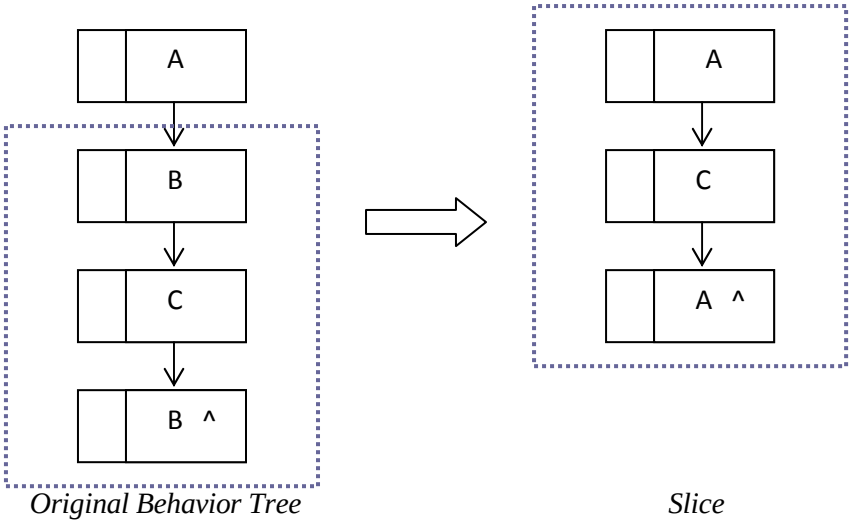
Most reversions and reference nodes would not be included using the slicing method described so far. Nonetheless, they are essential, as they are necessary for describing repeated behaviour and non-terminating systems. In the following discussion, reversions and reference nodes will be collectively referred to as *jump nodes*, for ease of reading.

Leaving out a jump node could result in a slice that performs different behaviour to the original model. For instance, a component might change state and then change back to an earlier state after the control flow follows back a reversion. Leaving out the reversion in this case would incorrectly model the component only changing state once. For this reason, the jump nodes must be added to the slice as well. A jump node may introduce additional dependencies to the slice. Therefore an additional traversal of the dependency graph starting at each jump node is necessary.

Target Nodes

The next consideration is the target of a jump node. The target of a jump node might not already be in the slice, so when a jump node is added to the slice, its target may have to be added as well. Since the goal is to reduce the size of the slice as much as possible, the target should not be included into the slice unless it is absolutely necessary. An alternative is to assign a new target to the jump node. The location of the target node affects which nodes will be repeated after the reversion executes, i.e which nodes are involved in the loop from the reversion to the target. The new target must be as close as possible to the original target, in order to ensure that the same section of the tree will be repeated. Using the closest ancestor can lead to an incorrect slice. The ancestor was not originally involved in the loop, so its behaviour was never repeated. By choosing it to be the new target, it will now be repeated whenever the jump node executes. This situation is illustrated in Figure 29. The Behavior Tree on the left is the original model in which nodes B, C and the jump node are repeated in the loop. The picture on the right is the slice created by changing the target of the reversion to be node A, the closest ancestor of the original target. The slice now contains different execution traces than the original model. In the slice, the node A can be repeated infinitely often, whereas it could only execute once in the original model. Thus, the closest descendent must be used instead. This will produce a correct slice, since the descendent was originally involved in the loop.

Nevertheless, there is still a case that presents some difficulties. There may be more than one closest descendent in the slice, as shown in Figure 30, where both B and D are the closest descendents to the target node A. In this case, using one of the descendents is not sufficient. In the original model both branches were required to re-start when the reversion executed, due to the semantics of reversions. However, in the slice, only one branch repeats. In this situation, the original target node must be included into the slice.

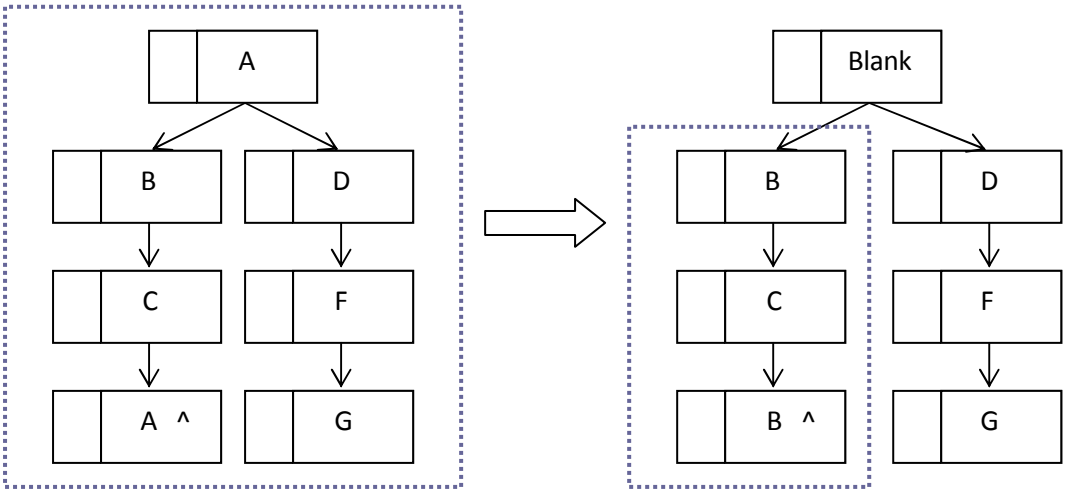


Original Behavior Tree *Slice*

(Dotted lines indicate regions which will be repeated due to the reversions).

Figure 29. Change in Loop Caused by Using Closest Ancestor

Definition 25 explains how the new target is calculated for a reversion or reference node in a slice. If the original target is in the slice, this is returned. Otherwise, if the original target is not in the slice and there is only one nearest descendent, the descendent is used as the new target. Finally, if the original target is not in the slice and there is more than one nearest descendent, the original target is added back to the slice. Recall from Section 2.3.1 that a function with a sub-script denotes the Behavior Tree (or BT control flow graph) it operates on, so in the definition, $target_B$ operates on the BT control flow graph B and $target_S$ operates on the slice S.



Original Behavior Tree *Slice*

(Dotted lines indicate regions which will be repeated due to the reversion to A and the new reversion to B).

Figure 20. Multiple Closest Descendents

DEFINITION 25. TARGET OF A REVERSION/REFERENCE NODE.

For a BT control flow graph B with a corresponding transition system $T_1 = (\mathcal{S}_1, AP_1, \mathcal{I}_1, \mathcal{L}_1, \mathcal{N}_1, \longrightarrow_1)$ and a slice S derived from B with a corresponding transition system $T_2 = (\mathcal{S}_2, AP_2, \mathcal{I}_2, \mathcal{L}_2, \mathcal{N}_2 \longrightarrow_2)$,

the function $target_S(n_x)$ returns n_y, where $descSet = nearestDesc(target_B(n_x))$ and

$$\text{where } n_y = \begin{cases} target_B(n_x), & \text{if } target_B(n_x) \in \mathcal{N}_2, \text{ or} \\ \text{the element of } descSet, & \text{if } target_B(n_x) \notin \mathcal{N}_2 \text{ and } |descSet| = 1, \text{ or} \\ target_B(n_x)^{*} & \text{otherwise.} \end{cases}$$

* In this case, the node $target_B(n_x)$ must be added to the slice as the parent of the nodes in $nearestDesc(target_B(n_x))$.

■

Alternatively, in the last case a blank node could be inserted instead, but there is no advantage in doing this. Although a blank node would not bring any extra dependencies to the slice, the original target will not either. The jump node has the same component name, behavior name and type as its target, so it has the same data, interference and message dependencies as its target. Thus, these dependencies would have already been added to the slice when the dependencies of the jump node were explored. If the target node has a control dependency, its descendents would also share the same dependency transitively. By a similar reasoning, if the target node has a synchronisation dependency, it is a conditional node, so the jump node would be transitively control dependent on it. Therefore, the synchronising nodes would already be in the slice. Similarly, if the target node has termination dependencies, these would equally apply to its descendents, since they are in the same thread. Again, these dependencies would already be in the slice. In conclusion, there is no advantage in using a blank node instead of the original target node. Currently, the original target is preferable because at present there is no mechanism in Behavior Trees for nodes to revert to or reference a blank node.

Reducing the Number of Jump Nodes

Including all the jump nodes into the slice is the simplest approach. This approach is correct but imprecise because not all of the loops are necessary. Reducing the number of loops could greatly reduce the time taken for model checking the slice.

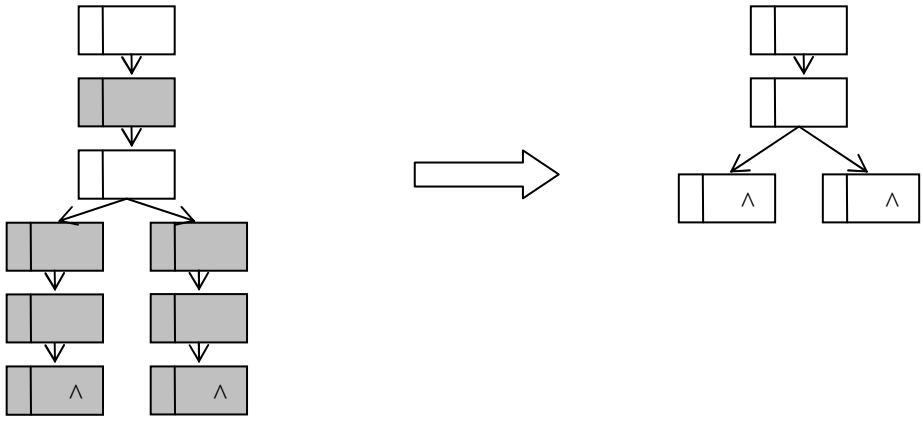
After the slice has been created, a node may end up with multiple jump nodes as descendents with no intervening behaviour, such as shown in Figure 31. The Behavior Tree on the left is the original tree, with the grey coloured nodes indicating the nodes which are not in the slice set. Assume that the target of both reversion is the same. The Behavior Tree on the right is its slice, after all reversions have been added back in. The reversions have now become the immediate children of the second node, whereas earlier there were several other nodes which executed in between. In the slice, both branches now lead directly to identical reversions, so it does not matter which one is chosen. The key observation is that if two or more of a node's children are jump nodes with the same target, both will result in identical execution traces. Due to this, only one of the jump nodes needs to remain in the slice.

The same principle can be extended to cases where even though the intervening behaviour is also in the slice, two or more jump nodes can be reached via the same sequence of nodes, thus producing the same execution traces. If two jump nodes cannot be reached via the same sequence of nodes, both must remain in the slice. The following example illustrates this.

Example

Consider the slice Behavior Tree in Figure 32. When `C[c]` is reached, there are two possible jump nodes which may execute. Both reach the same target location. However, they do not result in identical execution traces. The reversion on the left is only executed after the node `B[d]` has executed. Thus, that reversion can produce traces where the component `B` changes between the state `b` and the state `d` alternately, while the reversion on the right cannot produce such traces. In this case, both reversions are necessary in order to ensure that all the traces of the original model are preserved in the slice.

■



Original Behavior Tree (grey nodes indicate nodes not needed in the slice).

Slice after adding all reversions.

Figure 31. Unnecessary Jump Nodes

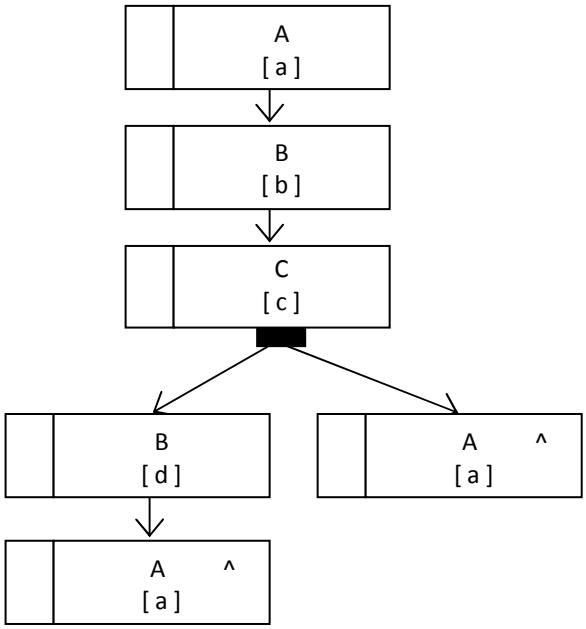


Figure 32. Reversions Producing Different Execution Traces

In order to decide which jump nodes should remain in the slice, it is necessary to identify the ones which result in identical execution traces. Let *n* and *m* be two jump nodes. If both produce identical execution traces, they must satisfy the following requirements:

(i) both *n* and *m* can only execute at the same steps in the same observable execution traces, i.e. for every observable execution trace σ such that $\sigma \frown \langle n \rangle$ is an execution trace, $\sigma \frown \langle m \rangle$ is also an execution trace and vice versa,

and

(ii) after *n* executes, for every execution trace after *n*, the observable remainder of the trace is identical to what would have occurred if *m* had executed instead and vice versa, i.e. if $\langle n \rangle \frown \sigma$ is an execution trace, where σ is an observable execution trace, then $\langle m \rangle \frown \sigma$ is also an execution trace and vice versa.

The second requirement is satisfied if both jump nodes have the same target, since they both lead to the same subsequent behaviour. For the first requirement, it must be determined whether both jump nodes can only execute in the same traces. This occurs if both jump nodes can be reached by the same

sequence of nodes in the slice. Therefore, they must have the same closest ancestor in the slice. However, sharing the closest ancestor is not sufficient to guarantee the first requirement. One jump node may have additional control dependencies that the other does not have, thus restricting the traces in which it can execute.

Fortunately it is not necessary to consider all control dependencies from each jump node. What matters is only whether or not there is a chain of dependencies from the jump node to a node in the slice set. Suppose that two jump nodes *n* and *m* have no transitive dependencies to a node in the slice set. Then, the nodes that *n* and *m* depend on can execute at any time, regardless of the current states of nodes in the slice set. Therefore, it becomes a non-deterministic choice as to whether or not *n* and *m* will execute in a particular trace. Both nodes are effectively equivalent to each other, since both may or may not execute in every trace. The only exception is if one of the jump nodes can *never* execute, due to some dependency that is never satisfied. Such a situation can be identified by searching the dependency paths starting at the jump node, to identify any nodes needed by the jump node that cannot execute before it. An alternate solution is to only remove a jump node *n* if there is another jump node *m* whose dependencies are a strict subset of *n*'s dependencies. This ensures that whenever *m* can execute, *n* can too, as *n* is more restricted.

Comparing the entire chains of dependencies originating at each jump node is too computationally expensive. A simpler solution is to compare the start of the dependency paths. If both paths start with the same node, the rest of the dependency paths will be identical. If one jump node has a dependency that restricts the traces in which it can execute, the associated dependency path must begin with either a control, data or interference dependency. According to the second requirement, both jump nodes have the same target node. In that case, both must have matching component names, behaviour names and types, so they are already known to share the same data and interference dependencies. Therefore, the only paths which need to be compared are those that begin with a control dependency. If both have matching control dependencies, they therefore have matching dependency paths.

In fact, it is not necessary for both nodes to have control dependencies to the *same* nodes; only to have control dependencies to matching nodes. Since both controlling nodes have the same component and behaviour names, both will lead to the same dependency paths. Figure 33 gives an example of two jump nodes that both have control dependencies to matching G?g? nodes. The only differences would be any further control dependencies from the controlling nodes themselves to an ancestor. However, in such a case, the jump node would be transitively control dependent on the ancestor as well. Therefore, to check whether two jump nodes have the same dependencies, it is sufficient to check that all of their transitive control dependencies match.

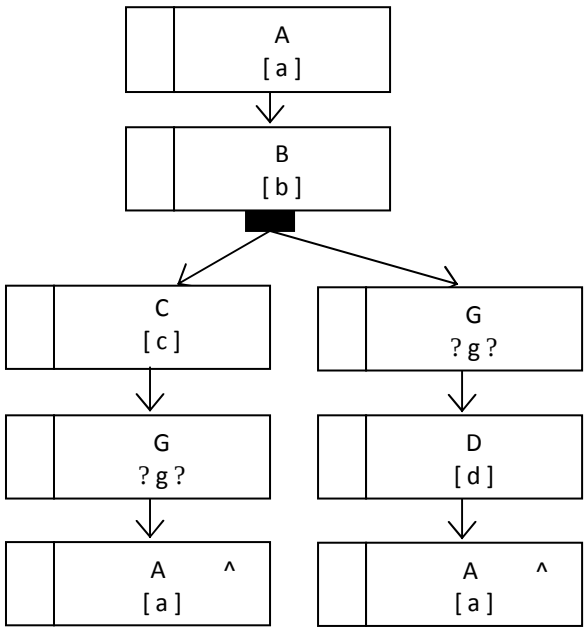


Figure 33. Example of two reversions with the same dependencies

Further reductions can be made by observing that no nodes in the system can influence an external input node, so the component and message names of the external input node are irrelevant. Therefore, if one jump node has a control dependency to an external input node, it is enough for the other jump node to have a control dependency to *any* external input node, not necessarily a matching one.

This process of checking for matching dependencies is utilised for a function *sameGuards*, given in Definition 26. The function takes three nodes as arguments: n_x, n_y and n_l. The node n_l is the closest ancestor in the slice of the two jump nodes, n_x and n_y. First, the set of nodes which n_x is transitively control dependent on are found, excluding nodes higher than n_l. Out of these, if any of them are transitively dependent on a node in the slice set, then n_y is required to also have a control dependency to a matching node. Additionally, if n_x is transitively control dependent on an external input node, n_y must be too. The function *sameGuards* returns true if these conditions hold.

DEFINITION 26. CHECKING FOR SAME GUARDS.

Let S_φ be the slice set. Let $required(n_x, n_l) = \{n_i \mid n_i \overset{cd^+}{\rightarrowtail} n_x \text{ and } n_i \in desc_o(n_l)\}$.

Then, the function $sameGuards(n_x, n_y, n_l)$ returns true iff $\forall\, n_i \in required(n_x, n_l)$ such that $\exists\, n_z \in S_\varphi$ where $n_z \overset{d}{\rightarrowtail} m_0 \overset{d}{\rightarrowtail} m_1 \overset{d}{\rightarrowtail} \ldots \overset{d}{\rightarrowtail} m_k \overset{d}{\rightarrowtail} n_i$ and $\forall\, 0 \le j \le k, m_j \notin required(n_x, n_l)$,

$\exists\, n_p \in required(n_y, n_l)$ such that either:

- $n_i = n_p$ or
- $matching(n_i, n_p)$ and

$\forall\, n_i \in required(n_x, n_l)$ such that $type(n_i) = external\ input$, $\exists\, n_p \in required(n_y, n_l)$ such that $type(n_j) = external\ input$.

■

If two jump nodes have the same closest ancestor in the slice, the same target, are either both reversions or both reference nodes and *sameGuards* returns true, then only one of them is necessary in the slice. This is given by Definition 27. The function $equiv_\varphi(n_x, n_y)$ returns *true* if the two nodes are equivalent. The requirement that both must be reversions or both reference nodes is necessary due to the difference in semantics between the two types: a reversion will terminate all the sub-threads below the target, while a reference node will not.

In the definition, the closest ancestor is given by examining each leaf node of the slice (before any jump nodes have been inserted) and comparing all jump nodes that are descendents of the leaf node in the original model. The leaf node is the closest ancestor in the slice for each of those jump nodes. Using this method makes it easy to determine whether a particular jump node can produce traces which are different to other jump nodes. Note that in the following definition, $canExec(n)$ denotes that the node n is able to execute in at least one trace, i.e. it is not the case that it can *never* execute.

DEFINITION 27. EQUIVALENT JUMP NODES.

For a transition system $B = (\mathcal{S}_1, AP_1, \mathcal{J}_1, \mathcal{L}_1, \mathcal{N}_1, \longrightarrow_1)$ of a BT control flow graph and a transition system $S = (\mathcal{S}_2, AP_2, \mathcal{J}_2, \mathcal{L}_2, \mathcal{N}_2 \longrightarrow_2)$, where $S = slice_\varphi(B)$ for some formula φ,

$\forall\, n_l \in S$ such that $leaf(n_l)$,

$\forall\, n_x, n_y \in desc_B(n_l)$, such that $flag(n_x) = flag(n_y) \in \{rev, ref\}$ and $canExec(n_x)$ and $canExec(n_y)$,

if $target_s(n_x) = target_S(n_y)$ and $sameGuards(n_x, n_y, n_l)$, then $equiv_\varphi(n_x, n_y)$.

Otherwise if $target_B(n_x) \in desc_B(n_l)$ and $target_B(n_y) \in desc_B(n_l)$ and $sameGuards(n_x, n_y, n_l)$, then $equiv_\varphi(n_x, n_y)$. ■

Example

Returning to the earlier example in Figure 32, the reversion on the left would not be compared with the reversion on the right, because their closest ancestors in the slice are different. The node `B[d]` is the last slice node before the reversion on the left, whereas `C[c]` is the last slice node before the other

reversion. Thus, both reversions would be kept in the slice, since they can produce different traces of behaviour.

■

A common result of slicing is for a node to have several jump nodes as descendents that each cause divergence (traces consisting entirely of stuttering steps). Since the purpose of slicing Behavior Trees is for verification, such divergent behaviour must be preserved in the slice. The jump nodes in such cases must therefore remain in the slice. Again, the same principle can be applied. Locate two or more jump nodes that result in identical observable execution traces. As well as being identical in terms of observable nodes, these traces must both have divergent behaviour occurring at the same steps in each trace. Divergence occurs when a loop operates entirely in a stuttering portion of the tree. Therefore, the target m of a jump node n must also be a descendent of n's closest ancestor in the slice. This is given by the last statement in Definition 27. If the targets of both jump nodes are descendents of their closest ancestor in the slice, n_l, and both have the same dependencies, then only one of the jump nodes is necessary.

Example

Consider the Behavior Tree in Figure 34. There are four reversion nodes below the node labelled N, each numbered 1 to 4. The dotted arrows indicate where each reversion's target is. The white coloured nodes are in the slice set. Reversions 1 to 3 all have the same closest ancestor in the slice, node N. On the other hand, the closest ancestor in the slice of Reversion 4 is node M, so it cannot be compared with Reversions 1 to 3. Reversion 4 should only be compared with any other reversions below node M. In this case, there is only one reversion, so Reversion 4 must remain in the slice. Out of the other three, all have targets that are descendents of N, so assuming that they have matching control dependencies, only one of them needs to be in the slice.

■

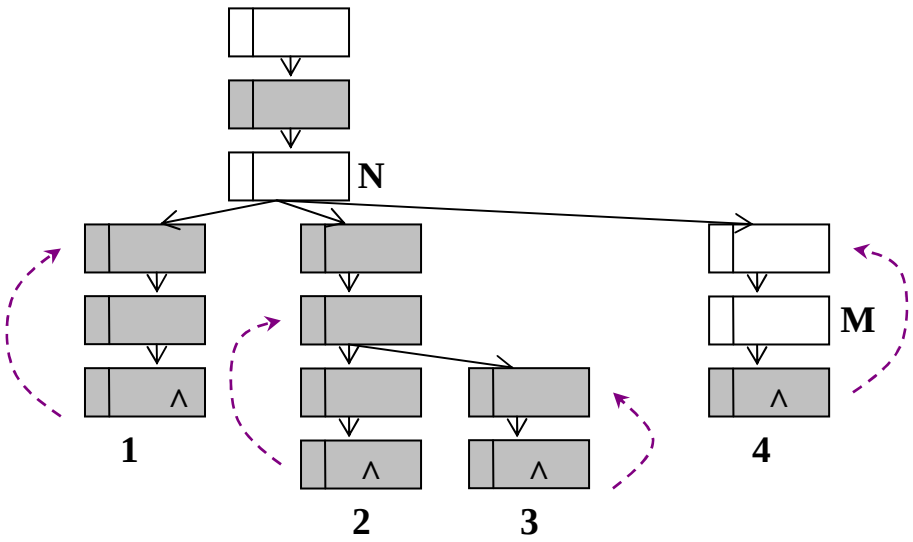


Figure 34. Divergence Caused By Reversions

The following lemma demonstrates that if there is a reversion or reference node n_y that is equivalent to another reversion or reference node n_r, then n_y is able to execute in all the same traces as n_r and produce the same subsequent traces. This result confirms that reversion or reference nodes that are equivalent to another node are not required to be included into the slice.

LEMMA 2. PRESENCE OF JUMP NODES

For a doubly-labelled transition system $B = (\mathcal{S}_1, AP_1, \mathcal{J}_1, \mathcal{L}_1, \mathcal{N}_1, \longrightarrow_1)$ of a BT control flow graph and a transition system $S = (\mathcal{S}_2, AP_2, \mathcal{J}_2, \mathcal{L}_2, \mathcal{N}_2 \longrightarrow_2)$, where $S = slice_\varphi(B)$ for some formula φ,
$\forall n_r \in \mathcal{N}_1$ such that $type(n_r) = rev$ or ref, if $\exists n_y \in \mathcal{N}_2$ such that $equiv_\varphi(n_r, n_y)$, then
$\forall \sigma_1 \in traces(B)$, such that $\sigma_1 = \sigma_2 \frown \langle n_x \rangle \frown \sigma_3$, where $\sigma_2 \in traces(B)$ and $\sigma_3 \in traces(B)$,
$\exists \sigma_4 \in traces(B)$ such that $\sigma_4 = \sigma_5 \frown \langle n_y \rangle \frown \sigma_6$, where $\sigma_5 \in traces(B)$ and $\sigma_6 \in traces(B)$ and $proj_\varphi(\sigma_2) = proj_\varphi(\sigma_5)$ and $proj_\varphi(\sigma_3) = proj_\varphi(\sigma_6)$.

Proof.

Let S_φ be the slice set. By Definition 27, $\forall n_r \in \mathcal{N}_1$ such that $type(n_r) = \{rev, ref\}$, either $n_r \in \mathcal{N}_2$ or $\exists n_y \in \mathcal{N}_2$ such that:

- the closest ancestor of n_r in S_φ, n_a, is the same as the closest ancestor of n_y in S_φ,
- $target_S(n_r) = target_S(n_y)$ or ($target_B(n_r) \in desc_B(n_a)$ and $target_B(n_y) \in desc_B(n_a)$) and
- $sameGuards(n_r, n_y, n_a)$.

Let $\sigma_1 \in traces(B)$, where $\sigma_1 = \sigma_2 \frown \langle n_r \rangle \frown \sigma_3$.
If $target_S(n_r) = target_S(n_y)$ then $\exists \sigma_6 \in traces(B)$, such that $\langle n_y \rangle \frown \sigma_6 \in traces(B)$
and $proj_\varphi(\sigma_3) = proj_\varphi(\sigma_6)$.
Otherwise, if $target_B(n_r) \in desc_B(n_a)$ and $target_B(n_y) \in desc_B(n_a)$, then
σ_3 consists only of stuttering steps.
$\Rightarrow \exists \sigma_6 \in traces(B)$ such that $\langle n_y \rangle \frown \sigma_6 \in traces(B)$
$target_B(n_y) \in desc_B(n_a)$
$\Rightarrow \sigma_6$ consists only of stuttering steps.
$\Rightarrow proj_\varphi(\sigma_3) = proj_\varphi(\sigma_6)$.

Since n_a is the closest ancestor of both n_r and n_y such that $n_a \in S_\varphi$, the last observable node in σ_2 is n_a.
$\Rightarrow \exists \sigma_5 \in traces(B)$ such that the last observable node in σ_5 is n_a
$\Rightarrow proj_\varphi(\sigma_2) = proj_\varphi(\sigma_5)$.
$sameGuards(n_r, n_y, n_a)$,
$\Rightarrow$ both n_r and n_y have the same dependencies,
$\Rightarrow \sigma_5 \frown \langle n_y \rangle \in traces(B)$, since $\sigma_2 \frown \langle n_r \rangle \in traces(B)$,

□

3.4.4 Re-forming the nodes into a tree

The final set of slice nodes is often a disjoint set of sub-trees, which must be re-formed into a syntactically correct Behavior Tree. This situation results from deleting irrelevant nodes. Some program slicing algorithms instead replace irrelevant program statements with *skip* statements (statements that do not perform any action). This is a convenient method for maintaining the program structure. However, theses slicing approaches were designed for the purpose of debugging or understanding, not for verification. If the irrelevant Behavior Tree nodes were replaced with *skip* nodes, the final model used for model checking would still have the same number of program counter variables as the original. The skip nodes would have to be translated into transitions that do nothing except update the program counters. This would therefore cause unnecessary extra states to be examined by the model checker. Since the goal of slicing Behavior Trees is to reduce the model size as much as possible, it is more desirable to completely remove the irrelevant nodes from the tree.

A node and its subtree becomes disjoint from the main tree if its parent node has been removed from the tree. In some cases, this can be easily resolved by joining the node to its nearest ancestor in the slice. However, this can become complicated if branching nodes are involved, especially if the root nodes of the branches have also been removed, or if a branch originally split off into further branches. For this reason, blank place-holder nodes are sometimes necessary. These can be thought of as *skip* nodes as above; however, instead of using them to replace all deleted nodes, they are only used in certain cases in order to maintain the original branching structure of the tree.

The algorithm for re-forming the slice set into a proper Behavior Tree operates on the structure of the tree only; none of the individual characteristics of the nodes, such as their type or name, are necessary. The algorithm is thus best described using a function $T(n)$ which returns the slice sub-tree rooted at a given node. $T(root(B))$ describes the slice sub-tree which begins with the root node of the original Behavior Tree B. Since the slice begins with the same root node, the sub-tree returned is the entire slice tree. The notation $T(n) = (n, T_1, T_2, ..., T_m)$ describes the sub-tree with the root node n and for each sub-tree T_i, where $1 \geq i \geq m$, $parent(root(T_i)) = n$. The process for re-forming the slice set into a tree is given by Definition 28 below.

DEFINITION 28. REFORMING TREE STRUCTURE.
For a doubly-labelled transition system $B = (\mathcal{S}, AP, \mathcal{I}, \mathcal{L}, \mathcal{N}, \longrightarrow)$ of a BT control flow graph and a slice set $S_\varphi = nodes\_slice_\varphi(B)$,

$$T(n) = \begin{cases} (n, T(child_B(n, 0)), T(child_B(n, 1)), \ldots, T(child_B(n, x)), & \text{if } n \in S_\varphi, \\ (blank, T(child_B(n, 0)), T(child_B(n, 1)), \ldots, T(child_B(n, x)), & \text{if } n \notin S_\varphi \text{ and } x > 1, \\ T(child_B(n, 0)), & \text{if } n \notin S_\varphi \text{ and } x = 1, \\ (), & \text{otherwise.} \end{cases}$$

where x is the number of children of n in B, and *blank* denotes a blank node.

■

The process operates by following the structure of the original Behavior Tree in a depth-first manner. Each node is either placed into the new tree, replaced by a blank node or not included into the new tree. In the latter case, its nearest descendent in the new tree becomes joined to its nearest ancestor. The details of this approach are as follows. If a node n is in the slice, the sub-tree rooted at n, $T(n)$, consists of n joined to each of the sub-trees of its children from the original Behavior Tree. The sub-trees of the children are in turn given by the function T, so n will not necessarily end up being joined to its original children. If n is not in the slice, $T(n)$ depends on the number of children n has in the original Behavior Tree. If there is more than one child, n must be replaced with a blank node in the slice, in order to preserve the branching structure. In this case, $T(n)$ consists of a blank node joined to the sub-trees of n's children, again each given by the function T. If n has only one child, no node needs to be added to the slice at n's location. $T(n)$ is the same as the sub-tree of n's child. Finally, if n has no children, $T(n)$ is empty.

Using this process, the slice nodes will be all joined back into a tree, with blank nodes placed wherever there are branches without a parent. However, after the tree has been formed, there may be unnecessary blank nodes present. A blank node is created whenever a node is not in the slice but has several children in the slice. This is necessary in cases where the blank node's parent has other siblings. Suppose that the other siblings execute as an alternate choice, whereas the blank node's children execute concurrently, as shown in Figure 35. In this case, an alternate choice must first be made between each of the other siblings and the blank node. Then, if the blank node is chosen, its children can then execute. If the blank node was not used, it would result in mixed alternative and concurrent branching on the same level. Thus, blank nodes are necessary in such cases. However, using the above algorithm, a blank node might be inserted even when its parent has no other siblings. In such cases, the blank node is unnecessary, as its children can simply be connected directly to the blank node's parent. During the initial pass, it is not possible to identify the situation where a node originally had siblings, but does not have any in the slice. This will only be apparent after the tree has been fully

formed. Therefore, a second pass is necessary, in which such unnecessary blank nodes are removed. For each blank node, if it has no siblings, then it is removed and each of its children are joined to its parent. If a blank node has only one child or no children, the blank node is unnecessary, so is removed. This is described in Definition 29.

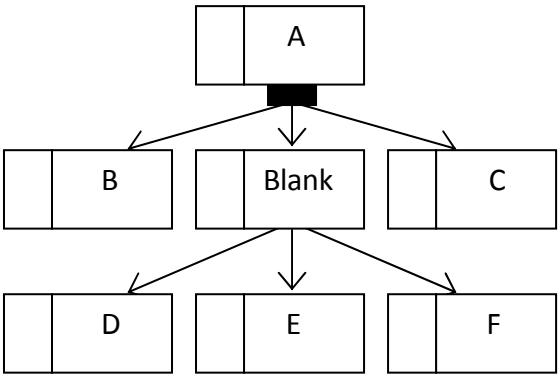


Figure 35. Branching Involving a Blank Node.

DEFINITION 29. REMOVING EXTRA BLANK NODES.

For a transition system $B = (\mathcal{S}, AP, \mathcal{I}, \mathcal{L}, \mathcal{N}, \longrightarrow)$ of a BT control flow graph and a slice $S = T(root(B))$,

$$T_2(n) = \begin{cases} T_2(child_S(n, 0)), & \text{if } c = 1 \text{ or } k < 2 \\ (n, T_2(child_B(n, 0)), T_2(child_B(n, 1)), \ldots, T_2(child_B(n, x)), & \text{otherwise.} \end{cases}$$

where $c = childNum_S(parent_S(n_x))$ and $k = childNum_S(n)$.

■

Finally, a third pass is needed to ensure that all the edges are of the correct type. Since some nodes which previously had several siblings might now have only one, their edge type must be changed to sequential instead of alternate or concurrent. Conversely, some sequential nodes might have moved up to become part of a group of siblings, so they must now be changed to concurrent or alternate, to match their siblings. This is described in Definition 30 below. For each node in the slice, if it is the only child of its parent, then its edge type is set to sequential. Otherwise, its edge type is set to match the edge type of the parent's original children. Note that since all of the children must have matching edge types, it is sufficient to query only the edge type of one of the children.

DEFINITION 30. CHANGING EDGE TYPES.

For a transition system $B = (\mathcal{S}, AP, \mathcal{I}, \mathcal{L}, \mathcal{N}, \longrightarrow)$ of a BT control flow graph and a slice $S = T_2(root(B))$,

$\forall n_x \in S$ and $n_p = parent_S(n_x)$, if $childNum_S(parent_S(n_x)) = 0$, set $edgeType_S(n_p, n_x) = seq$; otherwise set $edgeType_S(n_p, n_x) = edgeType_B(n_p, child_B(n_p, 0))$.

■

Example.

Figure 36 shows an example of a slice that has been re-formed into a tree. The Behavior Tree on the left is the original tree, with grey boxes representing the nodes that will not be in the slice. The Behavior Tree on the right is the final slice. Note that after the first phase, node A would have a blank node its only child and nodes B and C will be children of the blank node. In the second phase, the blank node is deemed unnecessary and removed. ■

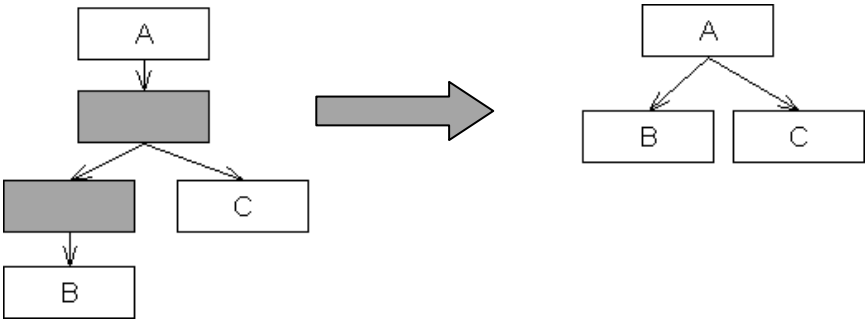


Figure 36. Re-forming a slice into a Behavior Tree.

Example.

In Figure 37, two blank place-holder nodes are necessary because A has another child E. There is one thread in which E executes, one with D and one with the B and C alternate choice. ■

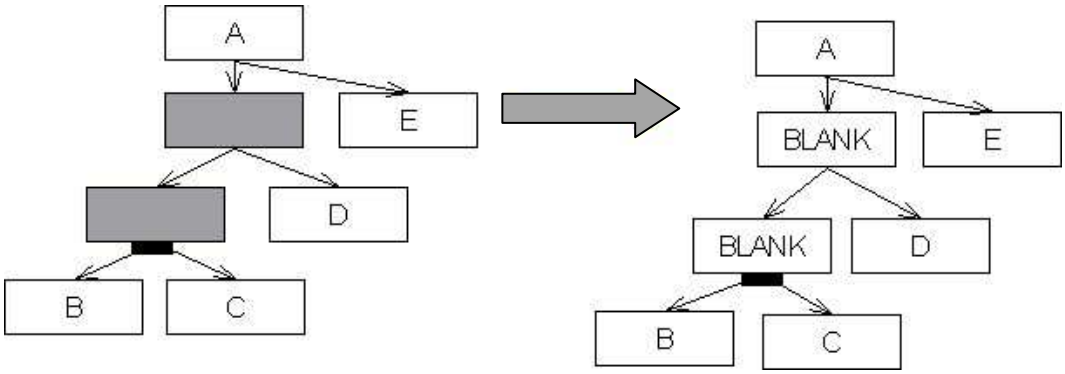


Figure 37. Re-forming a slice into a Behavior Tree using two place-holder nodes.

The *for-one* and *for-all* constructs are used to model that some behaviour applies to one or all items in a set. The sub-tree below a *for-all* or *for-one* node usually contains at least one node referring to the items in the set, for instance to set all the items to a particular state. After slicing, however, all of these nodes may have been removed, leaving only nodes that do not refer to the items in the set (i.e. the parameter of the *for-all* / *for-one* expression). In previous work, (Yatapanage, et al., 2010), the *for-all* / *for-one* nodes were removed in these cases. However, it would not always be correct to do so, since the user may have designed the model with the expectation that the descendents of the *for-all* / *for-one* node would be repeated. One possible solution is to automatically identify locations where the *for-all* / *for-one* node *may* be unnecessary and then to ask the user to decide whether or not the node should be retained in the slice. As a default, all *for-all* / *for-one* nodes are added to the slice.

Example.

In Figure 38, for each entry ticket purchased, a global counter is updated. If the *for-all* / *for-one* node was to be removed, the counter would incorrectly increase only once. This type of situation is impossible to determine automatically, as it requires contextual knowledge of the system.

■

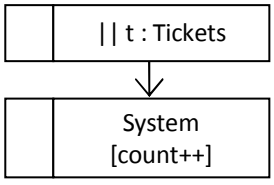


Figure 38. Example with a *for-all* node.

By following these steps, the slice set is transformed into a Behavior Tree. The transition system corresponding to the slice tree is given by the function *nodes_slice*, as given in the following definition.

DEFINITION 31. SLICE.
For a BT control flow graph G and its corresponding transition system B, the function $slice_{\varphi}(B)$ returns the transition system of the slice created from the slice set $nodes\_slice_{\varphi}(B)$, by using the construction algorithms T_1 and T_2. ■

Each of the functions that operate on BT control flow graphs and their nodes, such as *component*(*n*), can be applied to the slice and its nodes as well. In order to make it easier to reason about a Behavior Tree and its corresponding slice, the thread identifiers for the slice are the same as for the original model. Therefore, if a thread *t* is in the slice, there will be a thread *t* in the original model as well, but not necessarily vice versa.

3.5 Slicing Algorithm

The algorithm for computing a slice of a BT is computationally inexpensive. The creation of the dependence graph needs only to be performed once per BT and can be re-used for any temporal logic formula. The slicing algorithm normally used for programs is a two-phase algorithm designed for inter-procedural programs (Reps, et al., 1994). However, since Behavior Trees do not have procedure calls, a simpler one-phase slicing algorithm is sufficient. In the following, the modules for identifying the dependencies are described, followed by the calculation of the slice.

From the definition of control dependency, the only nodes that can create control dependencies are ones with two successors in the BT control flow graph. These are only guards, selections, synchronisations, internal input events or external input events. The algorithm for searching for control dependencies is given below in pseudocode. The function takes two parameters: a node *n* and a node *lastGuard*, which is the closest controlling ancestor. It recursively explores each node in the BT (lines 5-6). A mapping, *controlDepMap*, is maintained from each node to the set of nodes it is control dependent on. The *controlDepMap* variable is updated to store the control dependency from *n* to *lastGuard* (lines 1-2). In line 3, if the current node *n* is a selection, guard, synchronisation or input event, it induces a control dependence on each of its descendents, upto the next controlling node. Therefore, line 4 sets *n* to be *lastGuard*, which is given as the second parameter when the function is recursively called for the children. The time complexity for this method is O(*n*), where *n* is the number of nodes in the Behavior Tree, since it explores each node exactly once.

```
     findControlD(Node n, Node lastGuard)
1      if (lastGuard != null) then
2        controlDepMap.store(n, lastGuard);
       end if
3      if (type(n) == (selection OR BTguard OR input OR synch) then
4        lastGuard:=n;
       end if
5      for each child c of n do
6        findControlD(c, lastGuard);
       next c
```

Data dependencies are calculated in two steps. First, all the guards, selections and state-realisations in the BT are identified in a single traversal of the tree. In the psuedocode below, *guards* is a mapping from behavior names to the set of guard and selection nodes with that behavior. *SRs* is a mapping from behavior names to the set of state-realisation nodes with that behavior. The time complexity for this step is O(*n*), where *n* is the number of nodes in the Behavior Tree, since it explores each node exactly once.

```
     identifyGuardsAndSRs(Node n)
1    create map guards, SRs;
2      if (type(n) == state realisation) then
3          currentSet = SRs.lookup(behav(n));
4          SRs.setAt(behav(n), currentSet ∪ {n});
5      else if (type(n) == guard or selection) then
6          currentSet2 = guards.lookup(behav(n));
7          guards.store(behav(n), currentSet2 ∪ {n});
       end if
8      for each child c of n do
9        identifyGuardsAndSRs(c);
       next c
```

In the next step, each guard or selection node is identified as having a data dependency to each state-realisation node that has the same behaviour name. The variable *dataDepMap* is a mapping from guard and selection nodes to the set of state-realisations they are dependent on. For each pair (*behaviour*, *nodeList*) in the mapping *SRs*, the set of guard and selection nodes with the same behaviour name are located (lines 2-3). For each node *n* in the guard node list, it is first checked that *n* can be reached from each node *m* in *nodeList* (lines 4-7). For nodes in parallel threads, the check always returns true. For nodes in the same thread, the check returns true if there is a path from *m* to *n* in the BT control flow graph. This check can be performed in O(1) time, by utilising an ordering on node IDs that allows one to immediately determine whether a node is a descendent of another. The nodes that satisfy the check are added to a new list (lines 8-9). Finally, *dataDepMap* is updated to contain the dependency from *n* to the new list (line 10). For each guard (or selection) node, every state realisation node in the corresponding list is explored once. Therefore, the worst-case time complexity is $O(n^2)$, where *n* is the number of nodes in the Behavior Tree. However, this case is impossible since each guard is only dependent on the state realisations with the same behaviour name. In most cases, there will only be a few state realisations and guards for each behaviour name, so the complexity will normally be less than $O(n)$.

```
     findDataD(Node n)
1    identifyGuardsAndSRs(n);
2    for each (behaviour, nodeList) ∈ SRs do
3        guardNodes = guards.lookup(behaviour);
4        for each n ∈ guardNodes do
5            create new list nodeList2;
6            for each m ∈ nodeList do
7                bool isReachable = checkIfReachable(m, n);
8                if isReachable then
9                    nodeList2.add(m);
                 end if
             next m
10           dataDepMap.store(n, nodeList2);
         next n
     next
```

The algorithm for calculating message dependencies operates in the same way as for data dependencies. The message nodes are first identified. In the next stage, the input nodes are matched to the output nodes in the same way that the guards were matched to the state-realisations for the data dependency function.

Synchronisation dependencies are very simple. Each synchronising node is dependent on each of the others. The algorithm simply records these dependencies by exploring each synchronising group

once. Within each group, the nodes are accessed once for each of their synchronising partners. As for the previous case, the algorithm has a worst case of $O(n^2)$ time complexity, in the case where each node in the tree is a synchronisation node. However, for most Behavior Trees, there are only one or two synchronising groups, each involving only two or three nodes each, so the practical time complexity would be far less.

The pseudocode below shows the function which calculates termination dependencies. It begins by creating a mapping *terminatingRoots*, from nodes to the node that can terminate them. The mapping will only contain the root nodes of branches that can be terminated. The algorithm traverses each node *n* of the tree. If *n* is a thread kill, its target is mapped to *n* in *terminatingRoots* (lines 3-4). Next, if *n* is a reversion, each child of the target that is not an ancestor of *n* is mapped to *n* in *terminatingRoots* (lines 5-8). Finally, if *n* is an alternative branching node, each of its siblings is mapped to it (lines 9-12). When the entire tree has been traversed, each pair of nodes (*t*, *r*) in *terminatingRoots* is traversed (line 13). For each node *t*, each of its descendents are termination-dependent on *r*. This is recorded in another mapping *termDepMap*, which maps nodes to the nodes they are termination-dependent on (lines 14-19). This function also has a worst-case time complexity of $O(n^2)$, but again this represents an unrealistic case. In most cases, there are only a few nodes that cause termination dependencies in the tree.

```
     findTermDep(Node n)
1    create map terminatingRoots;
2    for each node n do
3       if (flag(n) == threadKill) then
4          terminatingRoots.add(n.target, n);
        end if
5       if (flag(n) == reversion) then
6          for each child c of n.target do
7             if (! n in c.desc()) then
8                terminatingRoots.add(c, n);
             end if
          next c
        end if
9       if (edgeType(edge(n.parent, n)) == alt) then
10         for each child c of n.parent do
11            if (c != n) then
12               terminatingRoots.add(c, n);
             end if
          next c
        end if
     next n

13   for each (t, r) in terminatingRoots do
14      for each d ∈ t.desc() do
15         list termNodes = termDepMap.lookup(d);
16         if (termNodes == null) then
17            create new list termNodes;
           end if
18         termNodes.add(r);
19         termDepMap.store(d, termNodes);
        next d
     next
```

After all of the dependencies have been calculated, the stored information can be re-used for all slices from the Behavior Tree. The previous functions need only be executed once per Behavior Tree. For each new criterion set, the function given below calculates the slice set. The function makes use of four sets of nodes, *visited*, *tempSet*, *sliceSet*, and *currentSet*. The first two are initialised to empty sets, while *sliceSet* and *currentSet* are both initialised to contain the nodes in the criterion (lines 1-2). The main loop of the algorithm operates until *currentSet* no longer contains any nodes (line 3). For each node *n* in *currentSet*, it is not explored unless it is not in the *visited* set. This prevents infinite cycles from occurring due to cyclic dependencies, such as the termination dependencies between alternative branching siblings. If *n* has not been previously explored, it is added to the *visited* set (lines 4-6), to prevent it from being explored again in the future. Next, the nodes which *n* is control-dependent on are located (line 7). Each of these are added to *tempSet* (lines 8-10). This is repeated for each of the types of dependencies. Finally, *tempSet* contains all the nodes that *n* is dependent on. These are all added to *sliceSet* (line 13). When all the nodes in *currentSet* have been explored, the set is emptied and replaced with the nodes in *tempSet*, which are the nodes that were discovered by exploring the dependencies (lines 14-15). The set *tempSet* is also emptied. The while loop then continues by exploring each of the new nodes in *currentSet*. This continues until no more new dependencies can be reached. The complexity of this algorithm is O(*n*), since no node is explored more than once, due to the *visited* set.

```
     calculateSlice( )
1    initialise visited, tempSet to empty;
2    initialise sliceSet, currentSet to nodes in criterion;
3    while (currentSet.size > 0) do
4       for each n ∈ currentSet do
5          if (! n ∈ visited) then
6             visited.add(n);
7             depNodes = controlDepMap.lookup(n);
8             if (depNodes != null) then
9                for each m ∈ depNodes do
10                  tempSet.add(m);
                 next m
              end if
11            depNodes = dataDepMap.lookup(n);
12            ... (repeat 9 to 11 for each DepMap).
13            sliceSet.addNodes(tempSet);
           end if
        next n
14      currentSet = empty;
15      currentSet.addNodes(tempSet);
16      tempSet = empty;
     loop
```

3.6 Proof of Correctness

The purpose of this slicing approach is to enable larger models to be verified than would normally be possible. An essential requirement is therefore that the slice preserves the same set of properties as the original model. In other words, a CTL^*_{-X} property will be satisfied on the slice if and only if it is satisfied on the original model. The user must be reassured that this requirement will be fulfilled for any slice, regardless of the property or the original model. If this can be guaranteed, the slice can be used to replace the original model. For this reason, a proof of correctness of the approach is necessary. This section presents such a proof, based on the notion of bisimulation, which is a well-established technique for computing the equivalence of two structures. For further details on bisimulation, refer to Section 2.2.5 on page 25. Recall from Theorem 1 in Section 2.2.5 that if two transition systems are related by a branching bisimulation with explicit divergence, a CTL^*_{-X} property holds on one transition system if and only if it holds on the other. Since CTL_{-X} and LTL_{-X} are subsets of CTL^*_{-X}, this also guarantees the preservation of CTL_{-X} and LTL_{-X} formulas. In this section, it will be demonstrated that a branching bisimulation with explicit divergence can be constructed between a Behavior Tree model and its slice. This will thereby show that a slice preserves the same CTL^*_{-X} formulas as the original Behavior Tree.

In order to construct such a relation, it is necessary to show that every initial state in the original model can be matched by an initial state in the slice, as shown by Lemma 3 below. This is not a one-to-one mapping, since several of the initial states in the original model may be matched to the same initial state in the slice.

LEMMA 3. EQUIVALENCE OF INITIAL STATES

For a transition system $B = (\mathcal{S}_1, AP_1, \mathcal{I}_1, \mathcal{L}_1, \mathcal{N}_1, \longrightarrow_1)$ of a BT control flow graph and a transition system $S = (\mathcal{S}_2, AP_2, \mathcal{I}_2, \mathcal{L}_2, \mathcal{N}_2 \longrightarrow_2)$, where $S = slice_\varphi(B)$, $\forall s_i \in \mathcal{I}_1$, $\exists\, t_i \in \mathcal{I}_2$, such that $\mathcal{L}_\varphi(s_i) = \mathcal{L}_\varphi(t_i)$.

Proof.

$\forall n_1 \in init(B)$ such that $obs_\varphi(n_1)$, $n_1 \in C_\varphi$, by Definition 22.

$\Rightarrow n_1 \in init(S)$, by Definition 23.

$\Rightarrow \forall s_i \in \mathcal{I}_1$, $\exists\, t_i \in \mathcal{I}_2$, such that $\mathcal{L}_\varphi(s_i) = \mathcal{L}_\varphi(t_i)$.

□

The following definition describes the construction of a relation, $\mathcal{R}$, between the original model and a slice. The initial states are related by finding a matching initial state in the slice for each initial state in the original model. Subsequent states are then related as follows. Let *s* and *t* be two states in the relation such that a node *n* can execute in state *s*, leading to the state *s'*. If the node is in the slice and can be executed in state *t* as well, then both subsequent states are added to the relation. Note that in this case *n* may be either stuttering or observable. If *n* is not present in the slice, then *s'* is said to relate to *t*.

DEFINITION 32. RELATION $\mathcal{R}$

Let $B = (\mathcal{S}_1, AP_1, \mathcal{I}_1, \mathcal{L}_1, \mathcal{N}_1, \longrightarrow_1)$ be a transition system corresponding to a BT control flow graph and $S = (\mathcal{S}_2, AP_2, \mathcal{I}_2, \mathcal{L}_2, \mathcal{N}_2 \longrightarrow_2)$ be a transition system such that $S = slice_\varphi(B)$ for some formula φ.

A relation $\mathcal{R}_\varphi = \mathcal{N}_1 \times \mathcal{N}_2$ can be constructed as follows:
In the following, let *s*, *s'*, s_0, s_1, ... range over $\mathcal{S}_1$ and *t*, *t'*, t_0, t_1, ... range over $\mathcal{S}_2$.

$\forall s_i \in \mathcal{I}_1$, find $t_i \in \mathcal{I}_2$, such that $\mathcal{L}_\varphi(s_i) = \mathcal{L}_\varphi(t_i)$. (A state t_i must exist according to Lemma 3).

Let each $(s_i, t_i) \in \mathcal{R}_\varphi$.

Then, $\mathcal{R}_\varphi$ can be constructed inductively as follows: $\forall s \in \mathcal{S}_1$ and $t \in \mathcal{S}_2$, if $s \mathcal{R}_\varphi t$, then:

1) if $\exists s'$ such that $s \xrightarrow{n} s'$ and $n \in \mathcal{N}_2$ and if $\exists t'$ such that $t \xrightarrow{n} t'$, let $(s', t') \in \mathcal{R}_\varphi$ and
2) if $\exists s'$ such that $s \xrightarrow{n} s'$ and $n \notin \mathcal{N}_2$, let $(s', t) \in \mathcal{R}_\varphi$.

■

It now remains to be shown that the relation $\mathcal{R}$ is a branching bisimulation with explicit divergence. In order to shown this, an auxiliary result is necessary: if a state is related to another by the relation $\mathcal{R}$, then the states have identical labellings with respect to the variables in the criterion. This result arises from the definition of $\mathcal{R}$, as shown by the following lemma.

LEMMA 4. RELATIONSHIP BETWEEN STATES AND $\mathcal{R}$.

For a transition system $B = (\mathcal{S}_1, AP_1, \mathcal{I}_1, \mathcal{L}_1, \mathcal{N}_1, \longrightarrow_1)$ of a BT control flow graph and a transition system $S = (\mathcal{S}_2, AP_2, \mathcal{I}_2, \mathcal{L}_2, \mathcal{N}_2 \longrightarrow_2)$, where $S = slice_\varphi(B)$, $s \mathcal{R}_\varphi t \Rightarrow \mathcal{L}_\varphi(s) = \mathcal{L}_\varphi(t)$.

Proof.
(By induction).

Base Case: $s \in \mathcal{I}_1$ and $t \in \mathcal{I}_2$ and $s \mathcal{R}_\varphi t$.

$\Rightarrow \mathcal{L}_\varphi(s) = \mathcal{L}_\varphi(t)$, by Definition 32.

Induction Step:

Assumption: For some $s \in \mathcal{S}_1$ and $t \in \mathcal{S}_2$, $s \mathcal{R}_\varphi t \Rightarrow \mathcal{L}_\varphi(s) = \mathcal{L}_\varphi(t)$.

Required to show: $\forall s, s' \in \mathcal{S}_1$ such that $s \longrightarrow s'$ and $\forall t, t' \in \mathcal{S}_2$ such that $t \longrightarrow t'$, and $s \mathcal{R}_\varphi t$,

(1) $s' \mathcal{R}_\varphi t' \Rightarrow \mathcal{L}_\varphi(s') = \mathcal{L}_\varphi(t')$ and

(2) $s' \mathcal{R}_\varphi t \Rightarrow \mathcal{L}_\varphi(s') = \mathcal{L}_\varphi(t)$.

Case (1):

$s' \mathcal{R}_\varphi t'$

$\Rightarrow s \xrightarrow{n} s'$ and $t \xrightarrow{n} t'$, for some node n, by Definition 32, point (1).

$\Rightarrow \mathcal{L}_\varphi(s') = \mathcal{L}_\varphi(s) \oplus updates(n)$ and

$\mathcal{L}_\varphi(t') = \mathcal{L}_\varphi(t) \oplus updates(n)$.

$\mathcal{L}_\varphi(s) = \mathcal{L}_\varphi(t)$, by assumption.

$\Rightarrow \mathcal{L}_\varphi(s') = \mathcal{L}_\varphi(t')$.

Case (2):

$s' \mathcal{R}_\varphi t$

$\Rightarrow s \xrightarrow{n} s'$ and $n \notin \mathcal{N}_2$, by Definition 32, point (2).

$\Rightarrow \neg\, obs_\varphi(n)$, by Definition 23.

$\Rightarrow \mathcal{L}_\varphi(s) = \mathcal{L}_\varphi(s')$.

$\mathcal{L}_\varphi(s) = \mathcal{L}_\varphi(t)$, by assumption.

$\Rightarrow \mathcal{L}_\varphi(s') = \mathcal{L}_\varphi(t)$.

□

One of the requirements for branching bisimulation with explicit divergence is for each system to be able to match any observable steps made by the other, possibly preceded by stuttering steps. It is therefore necessary to show that every observable step made in the original model can be matched by the slice. This is demonstrated by the following lemma, Lemma 5. Since the slice does not have any nodes that are not in the original model, the stuttering requirement is not necessary. The proof is shown by induction over the dependency paths starting at a node n_x, which is the step made by the original system. The base case is where n_x has no dependencies. The induction step assumes that Lemma 5 holds for some node n_y such that n_x depends on n_y, and thus proves that Lemma 5 holds for n_x as well. Both cases are shown by contradiction. It is assumed that n_x can execute in the original model but not in the slice. Figure 39 illustrates the various cases which arise from this.

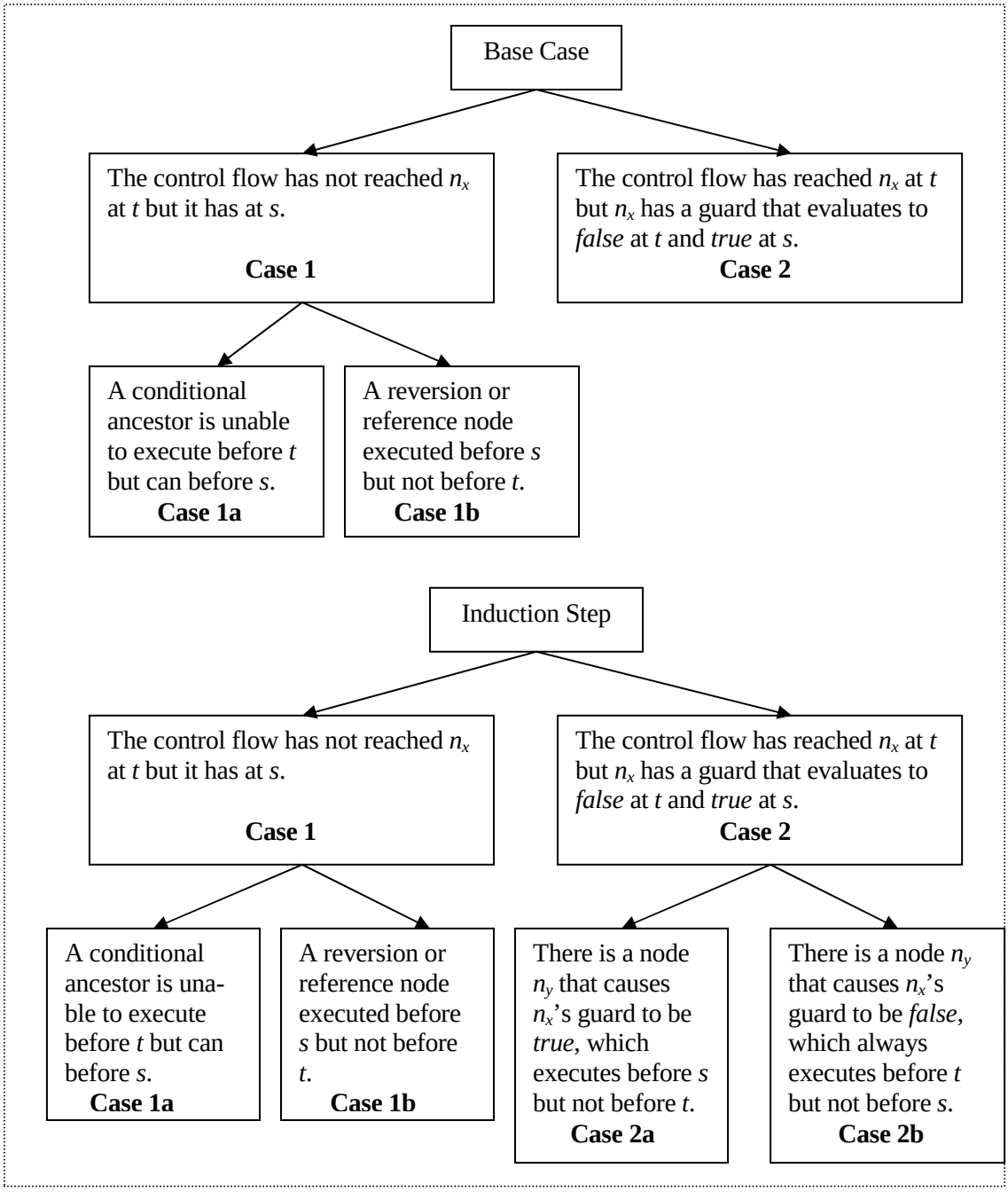


Figure 39. Cases where n_x executes at s but not at t.

LEMMA 5. ORIGINAL STEP MATCHES SLICE STEP

For a transition system $B = (\mathcal{S}_1, AP_1, \mathcal{I}_1, \mathcal{L}_1, \mathcal{N}_1, \longrightarrow_1)$ of a BT control flow graph and a transition system $S = (\mathcal{S}_2, AP_2, \mathcal{I}_2, \mathcal{L}_2, \mathcal{N}_2 \longrightarrow_2)$, where $S = slice_\varphi(B)$,
$\forall s \in \mathcal{S}_1, t \in \mathcal{S}_2$ such that $s \mathcal{R} t$, if $s \xrightarrow{n_x} s'$ and $n_x \in \mathcal{N}_2$, then $\exists t', t'' \in \mathcal{S}_2$ such that $t \xrightarrow{n_x} t'$, $s \mathcal{R}_\varphi t''$ and $s' \mathcal{R}_\varphi t'$.

Proof.
By induction.
In the following, let $s, s', s_0, s_1, \ldots$ range over $\mathcal{S}_1$ and $t, t', t_0, t_1, \ldots$ range over $\mathcal{S}_2$.

Base Case: There is no n_y such that $n_y \overset{d}{\rightarrowtail} n_x$.

By contradiction, assume that $s \mathcal{R}_\varphi t$ and $\exists s' \mid s \xrightarrow{n_x} s'$ and $n_x \in \mathcal{N}_2$,
but $\nexists t' \mid t \xrightarrow{n_x} t'$.

Case (1): $\exists m \in threads_S(n_x)$ such that $n_x \in ready_m(s)$ but $n_x \notin ready_m(t)$.

Case (1a): $\exists n_a \in ances_s(n_x)$ such that $conditional(n_a)$ and
$\forall \sigma_1 \in preTraces(s), n_a \in \sigma_1$, but $\forall \sigma_2 \in preTraces(t), n_a \notin \sigma_2$.

$s \mathcal{R}_\varphi t \Rightarrow \mathcal{L}_\varphi(s) = \mathcal{L}_\varphi(t)$, by Lemma 4.
$\Rightarrow \neg\, obs_\varphi(n_a)$.
$conditional(n_a)$ and $n_a \in ances_s(n_x)$
$\Rightarrow n_a \overset{cd^+}{\rightarrowtail} n_x$, by Lemma 1,
which contradicts the base case assumption.

Case (1b): $\exists n_r \in \mathcal{N}_1$ such that $n_r \notin \mathcal{N}_2$ and $type(n_r) \in \{rev, ref\}$ and $target(n_r) \in ances_o(n_x)$.
$\Rightarrow \exists n_y \in \mathcal{N}_2$ such that $equiv_\varphi(n_r, n_y)$, by Lemma 2.
$\Rightarrow \forall \sigma_1 \in preTraces(t), n_r \in \sigma_1$,
$\Rightarrow n_x \in ready_m(t)$,
which contradicts the assumption of Case (1).

Case (2): $\exists g \in guards(n_x)$ such that $g \notin \mathcal{L}_\varphi(t)$ but $g \in \mathcal{L}_\varphi(s)$.
$\Rightarrow guard(n_x) \neq \{\}$,
$\Rightarrow conditional(n_x)$,
$\Rightarrow \exists n_y$ such that $n_y \overset{d}{\rightarrowtail} n_x$,
which contradicts the base case assumption.

Induction Assumption: Requirement 1a holds for some node n_y, where $n_y \overset{n_x}{\rightarrowtail} n_x$,
where $d \in \{cd, dd, id, md, sd, td\}$.
Induction Step: Required to show that Requirement 1a holds for n_x.

By contradiction, assume that $s \mathcal{R}_\varphi t$ and $\exists s' \mid s \xrightarrow{n_x} s'$ and $n_x \in \mathcal{N}_2$,
but $\nexists t' \mid t \xrightarrow{n_x} t'$.

Case (1): $\exists\, m \in threads_S(n_x)$ such that $n_x \in ready_m(s)$ but $n_x \notin ready_m(t)$.

Case (1a): $\exists\, n_a \in ances_s(n_x)$ such that $conditional(n_a)$ and

$\forall \sigma_1 \in preTraces(s), n_a \in \sigma_1,$ but $\forall \sigma_2 \in preTraces(t), n_a \notin \sigma_2$.

$s \mathcal{R}_\varphi\ t \Rightarrow \mathcal{L}_\varphi(s) = \mathcal{L}_\varphi(t)$, by Lemma 4.

$\Rightarrow \neg\, obs_\varphi(n_a)$.

$conditional(n_a)$ and $n_a \in ances(n_x)$

$\Rightarrow n_a \overset{cd^+}{\rightarrowtail} n_x$, by Lemma 1,

$\Rightarrow n_a \in \mathcal{N}_2$, by Definition 23.

$\Rightarrow$ Lemma 5 holds for n_a, by the induction assumption.

Therefore, since $\forall \sigma_1 \in preTraces(s), n_a \in \sigma_1$,

$\Rightarrow \forall \sigma_2 \in preTraces(t), n_a \notin \sigma_2$,

which contradicts the assumption of Case 1a.

Case (1b): $\exists\, n_r \in \mathcal{N}_1$ such that $n_r \notin \mathcal{N}_2$ and $type(n_r) \in \{rev, ref\}$ and $target(n_r) \in ances_o(n_x)$.

$\Rightarrow \exists\, n_y \in \mathcal{N}_2$ such that $equiv_\varphi(n_r, n_y)$, by Lemma 2.

$\Rightarrow \forall \sigma_1 \in preTraces(t), n_r \in \sigma_1$,

$\Rightarrow n_x \in ready_m(t)$,

which contradicts the assumption of Case (1).

Case (2): $\exists\, g \in guards(n_x)$ such that $g \notin \mathcal{L}_\varphi(t)$ but $g \in \mathcal{L}_\varphi(s)$.

Let $g = (var, value_1)$.

Case (2a): $\exists\, n_y \mid n_y \in \mathcal{N}_1$, where $(var, value_1) \in updates(n_y)$ and

$\forall \sigma_1 \in preTraces(s), n_y \in \sigma_1,$ but $\forall \sigma_2 \in preTraces(t), n_y \notin \sigma_2$.

If $var \in components$, then:

$var \in DEF(n_y)$, by Definition 14.

$var \in REF(n_x)$, by Definition 15.

$\Rightarrow n_y \overset{dd\ /\ id}{\rightarrowtail} n_x$, by Definition 17 and Definition 18.

$\Rightarrow n_y \in \mathcal{N}_2$, by Definition 23.

$\Rightarrow$ Lemma 5 holds for n_y, by the induction assumption.

Therefore, since $\forall \sigma_1 \in preTraces(s), n_y \in \sigma_1$,

$\Rightarrow \forall \sigma_2 \in preTraces(t), n_y \notin \sigma_2$,

which contradicts the assumption of Case 2a.

If $var \in messages$ then:

$type(n_y) = intOut$ and $behavior(n_y) = var$.

$\Rightarrow n_y \overset{md}{\rightarrowtail} n_x$, by Definition 19.

$\Rightarrow n_y \in \mathcal{N}_2$, by Definition 23.

$\Rightarrow$ Lemma 5 holds for n_y, by the induction assumption.
Therefore, since $\forall \sigma_1 \in preTraces(s)$, $n_y \in \sigma_1$,
$\Rightarrow \forall \sigma_2 \in preTraces(t)$, $n_y \notin \sigma_2$,
which contradicts the assumption of Case 2a.

If $var \in synchLabels$, then:
$flag(n_x) = synch$ and $flag(n_y) = synch$ and $matching(n_x, n_y)$
$\Rightarrow n_x \overset{sd}{\rightarrowtail} n_y$, by Definition 20.
$\Rightarrow n_y \in \mathcal{N}_2$, by Definition 23.
$\Rightarrow$ Lemma 5 holds for n_y, by the induction assumption.
Therefore, since $\forall \sigma_1 \in preTraces(s)$, $n_y \in \sigma_1$,
$\Rightarrow \forall \sigma_2 \in preTraces(t)$, $n_y \notin \sigma_2$,
which contradicts the assumption of Case 2a.

Case (2b): $\exists n_y \mid n_y \in \mathcal{N}_2$, where $(var, value_2) \in updates(n_y)$ for some $value_1 \neq value_2$ and $\forall \sigma_2 \in preTraces(t)$, $n_y \in \sigma_2$, but $\forall \sigma_1 \in preTraces(s)$, $n_y \notin \sigma_1$.
$\Rightarrow \exists t_j, t_j'$ such that $t_j \xrightarrow{n_y} t_j'$
$\Rightarrow \exists s_i$ such that $s_i \mathcal{R}_\varphi t_j$ but $\nexists s_i'$ such that $s_i \xrightarrow{n_y} s_i'$
$\Rightarrow (s_i, t_j') \notin \mathcal{R}_\varphi$, by Definition 32,
$\Rightarrow (s, t) \notin \mathcal{R}_\varphi$, by Definition 32,
which contradicts the assumption.

□

LEMMA 6. SLICE STEP MATCHES ORIGINAL STEP

For a transition system $B = (\mathcal{S}_1, AP_1, \mathcal{I}_1, \mathcal{L}_1, \mathcal{N}_1, \longrightarrow_1)$ of a BT control flow graph and a transition system $S = (\mathcal{S}_2, AP_2, \mathcal{I}_2, \mathcal{L}_2, \mathcal{N}_2 \longrightarrow_2)$, where $S = slice_\varphi(B)$,
$\forall s \in \mathcal{S}_1, t \in \mathcal{S}_2$ such that $s \mathcal{R}_\varphi t$, if $t \xrightarrow{n_x} t'$ and $n_x \in \mathcal{N}_2$, then $\exists s', s'' \in \mathcal{S}_2$ such that $s \dashrightarrow^* s'' \xrightarrow{n_x} s'$, $s'' \mathcal{R}_\varphi t$ and $s' \mathcal{R}_\varphi t'$.

Proof.
By induction.
In the following, let s, s', s_0, s_1, ... range over $\mathcal{S}_1$ and t, t', t_0, t_1, ... range over $\mathcal{S}_2$.

Case (1): $\exists n_a \in \mathcal{N}_1$ such that $n_a \notin \mathcal{N}_2$, where $n_a \in ances_o(n_x)$ and $conditional(n_a)$
$\Rightarrow n_a \overset{cd^+}{\rightarrowtail} n_x$, by Definition 16,
$\Rightarrow n_y \in \mathcal{N}_2$, by Definition 23,
which contradicts the assumption.

Case (2): $\exists n_a \in \mathcal{N}_1$ such that $n_a \notin \mathcal{N}_2$,
where $type(n_a) = synch$ and $type(n_x) = synch$ and $matching(n_a, n_x)$.
$\Rightarrow n_a \overset{sd}{\rightarrowtail} n_x$, by Definition 16,

$\Rightarrow n_y \in \mathcal{N}_2$, by Definition 23,
which contradicts the assumption.

□

THEOREM 2. RELATIONSHIP BETWEEN SLICE AND ORIGINAL
For a transition system B corresponding to a BT control flow graph, for all transition systems S such that $slice_\varphi(B) = S$ for some formula φ, S is divergent-sensitive branching bisimilar to B,

i.e. $B \triangleq S$.

Proof.
Let $B = (\mathcal{S}_1, AP_1, \mathcal{I}_1, \mathcal{L}_1, \mathcal{N}_1, \longrightarrow_1)$ and $S = (\mathcal{S}_2, AP_2, \mathcal{I}_2, \mathcal{L}_2, \mathcal{N}_2 \longrightarrow_2)$. In the following, let s, s', s_0, s_1, ... range over $\mathcal{S}_1$ and t, t', t_0, t_1, ... range over $\mathcal{S}_2$.

Let $\mathcal{R}_\varphi$ be a relation constructed according to Definition 32. It remains to be shown that $\mathcal{R}$ is a divergent-sensitive branching bisimulation.
For $\mathcal{R}_\varphi$ to be a divergent-sensitive branching bisimulation, $\mathcal{R}_\varphi$ must satisfy all of the following properties. $\forall s, t$ such that $s \mathcal{R}_\varphi t$:

(1a) if $s \xrightarrow{n} s'$, for some $n \in \mathcal{N}_1$, then either $s \dashrightarrow s'$ and $s' \mathcal{R}_\varphi t$ or
$\exists t', t''$ such that $t \dashrightarrow^* t'' \xrightarrow{n} t'$, $s \mathcal{R}_\varphi t''$ and $s' \mathcal{R}_\varphi t'$.

(1b) if $t \xrightarrow{n} t'$, for some $n \in \mathcal{N}_2$, then either $t \dashrightarrow t'$ and $s \mathcal{R}_\varphi t'$ or
$\exists s', s''$ such that $s \dashrightarrow^* s'' \xrightarrow{n} s'$, $s'' \mathcal{R}_\varphi t$ and $s' \mathcal{R}_\varphi t'$.

(2a) if there exists an infinite path fragment $s \dashrightarrow s_0 \dashrightarrow s_1 \dashrightarrow ...$,
there exists an infinite path fragment $t \dashrightarrow t_0 \dashrightarrow t_1 \dashrightarrow ....$, such that $t_0 \mathcal{R}_\varphi s_k$,
for some $k \geq 0$.

(2b) if there exists an infinite path fragment $t \dashrightarrow t_0 \dashrightarrow t_1 \dashrightarrow ...$,
there exists an infinite path fragment $s \dashrightarrow s_0 \dashrightarrow s_1 \dashrightarrow ....$, such that $s_0 \mathcal{R}_\varphi t_k$,
for some $k \geq 0$.

Requirement 1a: if $s \xrightarrow{n_x} s'$, for some $n \in \mathcal{N}_1$, then either $s \dashrightarrow s'$ and $s' \mathcal{R}_\varphi t$ or
$\exists t', t''$ such that $t \dashrightarrow^* t'' \xrightarrow{n_x} t'$, $s \mathcal{R}_\varphi t''$ and $s' \mathcal{R}_\varphi t'$.

If $n_x \in \mathcal{N}_2$,
$\Rightarrow \exists t', t'' \in \mathcal{S}_2$ such that $t \xrightarrow{n_x} t'$, $s \mathcal{R} t''$ and $s' \mathcal{R}_\varphi t'$, by Lemma 5,
as required.

Otherwise, if $n_x \notin \mathcal{N}_2$,

$\Rightarrow \neg\, obs_\varphi(n_x)$ and $s' \mathcal{R}_\varphi t$, by Definition 32,

$\Rightarrow s \dashrightarrow s'$ and $s' \mathcal{R}_\varphi t$
as required.

Requirement 1b: if $t \xrightarrow{n_x} t'$, then either $t \dashrightarrow t'$ and $s \mathcal{R}_\varphi t'$ or

$\exists s', s''$ such that $s \dashrightarrow^* s'' \xrightarrow{n_x} s'$, $t \mathcal{R}_\varphi s''$ and $s' \mathcal{R}_\varphi t'$.

$t \xrightarrow{n_x} t'$

$\Rightarrow n_x \in \mathcal{N}_2$

$\Rightarrow \exists s', s'' \in \mathcal{S}_1$ such that $s \dashrightarrow^* s'' \xrightarrow{n_x} s'$, $s'' \mathcal{R}_\varphi t$ and $s' \mathcal{R}_\varphi t'$, by Lemma 5,
as required.

Requirement 2a:

$\forall s, t$ such that $s \mathcal{R}_\varphi t$, if there exists an infinite path fragment $s \dashrightarrow s_0 \dashrightarrow s_1 \dashrightarrow ...$,

there exists an infinite path fragment $t \dashrightarrow t_0 \dashrightarrow t_1 \dashrightarrow ....$, such that $t_0 \mathcal{R}_\varphi s_k$, for some $k \geq 0$.

Let $\rho_1 = < s, a_0, s_0, a_1, s_1 ... >$, where $\forall i \geq 0, \neg obs_\varphi(a_i)$
Let $r = \{n_x \mid \exists i \geq 0$ where $n_x = a_i$ and $type(n_r) \in \{rev, ref\}\}$
$r \neq \{\ \}$, since the path is infinite.
$\forall n_x \in r$, either $n_r \in \mathcal{N}_2$ or $\exists n_y \in \mathcal{N}_2$ such that $equiv_\varphi(n_r, n_y)$, by Lemma 2.
$\Rightarrow \forall n_x \in r$, if $\exists s_m, s_{m+1} \in \rho_1$ such that $s_m \xrightarrow{n_x} s_{m+1}$,
then $\exists t_j, t_{j+1} \in \mathcal{N}_2$ such that $t_j \xrightarrow{n_x} t_{j+1}$ or $t_j \xrightarrow{n_y} t_{j+1}$
$\Rightarrow \exists \rho_2 = < t, b_0, t_0, b_1, t_1 ... >$, where $\forall i \geq 0, \neg obs_\varphi(b_i)$ and for some $k \geq 0$, $b_0 = a_k$
$\Rightarrow t_0 \mathcal{R}_\varphi s_k$, for some $k \geq 0$.

Requirement 2b:

$\forall s, t$ such that $s \mathcal{R}_\varphi t$, if there exists an infinite path fragment $t \dashrightarrow t_0 \dashrightarrow t_1 \dashrightarrow ...$,

there exists an infinite path fragment $s \dashrightarrow s_0 \dashrightarrow s_1 \dashrightarrow ....$, such that $s_0 \mathcal{R}_\varphi t_k$, for some $k \geq 0$.

Proof by contradiction. Assume for every path from s, there is an observable node.
Let $\rho_2 = < t, b_0, t_0, b_1, t_1 ... >$
$\mathcal{N}_2 \subseteq \mathcal{N}_1$
$\Rightarrow \forall i \geq 0, b_i \in \mathcal{N}_1$
$\Rightarrow \forall \rho_1 \in runs(s)$, such that $\rho_1 = < s, a_0, s_0, a_1, s_1 ... >$ where for some $m \geq 0$, $\forall i$ where $0 \leq i < m$,
$\exists x$ such that $a_i = b_x$,
$s_m \xrightarrow{n} s_{m+1}$ and $obs_\varphi(n)$
$\Rightarrow n \in \mathcal{N}_2$, by Definition 23,
$\Rightarrow \exists j \geq 0$ such that $\rho_2 = < t, b_0, t_0, b_1, t_1 ..., t_j, n, t_{j+1} >$,
which is not an infinite stuttering path,
which contradicts the assumption.

□

The proof confirms that the dependencies presented in this chapter have been defined correctly and that no additional nodes are necessary in order to preserve CTL^*_{-X} properties.

This chapter presented the concepts of slicing of Behavior Tree models. The definitions for the different dependency types were given, as well as algorithms for computing the dependencies and re-forming the slice into a syntactically correct Behavior Tree. The slice was shown to preserve the same CTL^*_{-X} properties as the original, using divergence-sensitive branching bisimulation. The following chapter introduces a method for further reducing the size of the slice.

4 INFEASIBLE PATHS

As discussed in Section 2.2.3, interference dependence can lead to imprecise slices due to its intransitivity. Krinke (1998) developed the notion of threaded witnesses to solve this problem. Whenever a node is reached via an interference edge, it is only added to the slice if it forms a valid threaded witness with the other nodes collected so far along that dependency path. However, the slices produced may still be imprecise, as will be shown in section 4.2. In Krinke's method, when a node is to be added to the slice due to an interference dependency, it is first checked whether the node's dependency path forms a threaded witness. If it does not, the node is not included in the slice. Although this is correct, it does not prevent other nodes whose only dependency is to the discarded node, from being added to the slice. Since those other nodes could not execute without the discarded node, they could be removed from the slice as well, producing a smaller slice. Section 4.2 presents a new procedure, which considers the entire dependency path instead of single nodes, thereby removing more unnecessary nodes from the slice. If a sequence of nodes does not form a threaded witness, then none of the nodes are included in the slice.

4.1 Threaded Witnesses for Behavior Trees

Dependencies which cross thread boundaries, such as interference and message dependency, are intransitive. This can lead to imprecise slices because the normal slicing algorithm assumes that all dependencies are transitive. Although the resulting slice will be correct, it is imprecise, which means there may be unnecessary nodes. Since the slice is still correct, as shown by the proof in Section 3.6, the unnecessary nodes will not cause the slice to produce a different verification result than the original. However, by removing these unnecessary nodes, the resulting slice may be smaller. Krinke (1998) proposed the notion of threaded witnesses to identify such nodes. In this section, the concept of threaded witnesses will be adapted for Behavior Trees.

4.1.1 Threaded Witnesses

The following definitions have been adapted for Behavior Trees from the original definitions given by Krinke (2003). As described in Section 2.2.3, Krinke and Nanda and Ramesh (2000, 2006) both defined versions of slicing suitable for inter-procedural concurrent programs. Since Behavior Trees do not have procedure calls, the following definitions have been adapted from Krinke's approach for slicing intra-procedural concurrent programs, which uses threaded witnesses. The idea behind the threaded witness approach is to identify nodes which cannot execute before a criterion node and so cannot influence the criterion node. If a node n_c is in the criterion and is transitively dependent on a node n_x, the node n_x is normally included into the slice. However, if n_x is known to be unable to execute before n_c, since there is no feasible path between the two, there is no need to include it. This situation occurs if either the nodes are in alternative branches or n_x is a descendent of n_c and there is no way of reaching n_c via a reversion or reference node.

The notion of *threaded witness* is defined in Definition 34. A threaded witness is a sequence of nodes in which for every node n_x, for each of its predecessors n_y in the sequence, either n_x is *reachable* from n_y or they execute in parallel. A node n_x is reachable from another, n_y, if it is possible to execute n_y and then n_x afterwards. The definition of *reachable* is given in Definition 33. It is not sufficient for there to merely be a trace from n_x to n_y in the control flow graph, since traces in control flow graphs terminate at reversions and reference nodes even though the control flow is simply diverted by these nodes. Therefore, the definition includes the possibility that n_y can be reached from n_x via one or more reversions or reference nodes. In the following, *last*(σ) returns the last node and *first*(σ) returns the first node of a sequence σ.

DEFINITION 33. REACHABLE NODES

For two nodes n_x, n_y in a BT control flow graph *B*, *reachable*(n_x, n_y) iff

$\exists \sigma \in traces(B)$ such that $\sigma = < m_0, m_1, ..., m_k >$, where $m_0 = n_x$ and $m_k = n_y$ and for some $j \geq 0$, σ can be divided into *j* sub-sequences $\sigma_1, \sigma_2, ..., \sigma_j$ such that for every σ_i, where $1 \leq i < j$, $type(last(\sigma_i)) \in \{rev, ref\}$ and $first(\sigma_{i+1}) = target_B(last(\sigma_i))$.

■

DEFINITION 34. THREADED WITNESS

Let *G* be a dependency graph derived from a BT control flow graph *B*. Let $\pi \in paths(G)$ such that $\pi = < n_0, n_1, \ldots, n_k >$. Then, π is a *threaded witness*, denoted *tw*(π) iff:

(adapted from (Krinke, 1998))

$\forall 0 < i \leq k$, $\forall 0 \leq j < i$, either:

- $thread(n_i) \neq thread(n_j)$, or
- *reachable*(n_j, n_i) in *B*.

■

The slice is constructed by collecting all the nodes encountered via a backwards search of the dependency graph, starting at the criterion nodes, as before. However, this time, if a path in the dependency graph to a node is not a threaded witness, the node is not added to the slice. This is described in Definition 35. Note that termination dependency must be ignored at this stage, as many termination dependencies do not result in threaded witnesses but are still legitimate dependencies. For example, the termination dependence between nodes in alternative branches would result in dependency paths that are not threaded witnesses since there is no feasible path between the nodes. For this reason, termination dependencies are collected at a later stage of the slicing process, after the phase where threaded witnesses are used to remove infeasible dependencies.

DEFINITION 35. SLICING USING THREADED WITNESSES

A slice set containing only nodes on threaded witnesses, $nodes\_TW_\varphi(B)$, of a BT control flow graph *B* using a criterion C_φ and a dependency graph *G*, is defined as:

$$nodes\_TW_\varphi = \{n_i \mid \exists n_c \in C_\varphi \text{ and } \exists \pi \in paths(G) \text{ where } \pi = < n_i, ..., n_c > \text{ and } tw(\pi)\}.$$

■

Example.

Consider the Behavior Tree in Figure 40. Assume the slicing criterion is {C}, so the node C[c] is a criterion node. It is control dependent on B?b?, which is in turn data dependent on B[b] in the parallel thread. This is in turn control dependent on D?d?, which is data dependent on D[d]. The chain of dependencies can be seen in Figure 41. The normal slicing algorithm would include all of these nodes in the slice. However, D[d] and C[c] are in alternative branches and there are no reversions, so there is

no path from D[d] to C[c]. The dependency chain does not form a threaded witness. Therefore, D[d] cannot influence C[c] and should be left out of the slice.

■

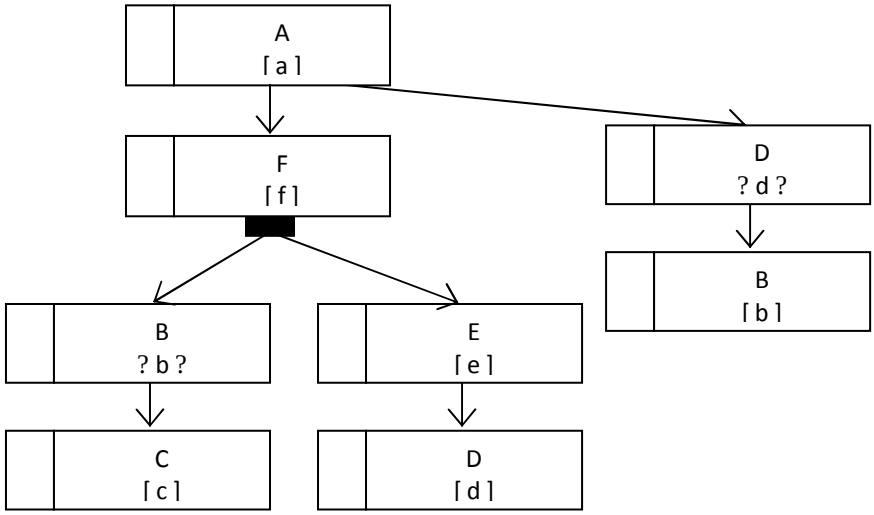


Figure 40. Example Behavior Tree.

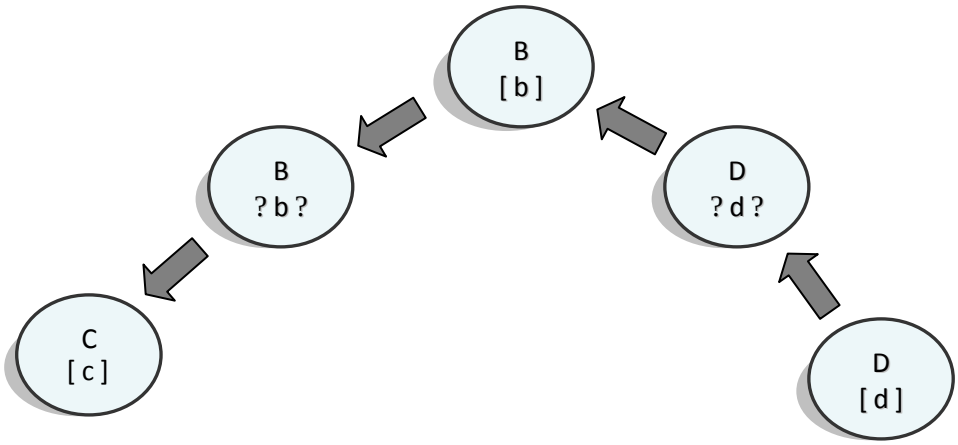


Figure 41. Dependence Graph for the Behavior Tree in Figure 40.

4.1.2 Nested Threads

As explained previously in Section 2.2.3, Nanda and Ramesh (2000, 2006) proposed an improvement to Krinke's algorithm in order to handle nested threads more precisely. This problem does not arise for Behavior Tree slicing, because of the difference in how threads are defined. As demonstrated by the example in Figure 8 on page 21, in Nanda and Ramesh's Control Flow Graphs, a thread is considered to start when the control flow branches into the new path and ends when the control flow converges back to the parent thread. On the other hand, threads in Behavior Trees are considered to start from the root and stretch all the way to a leaf node, as demonstrated by Figure 42, which shows the boundaries of two threads. By this definition, a node may belong to more than one thread. In the figure, the top three nodes are common to both threads. This definition prevents the nested threads

problem from occurring because both the parent and child threads are considered as one single thread. The style of Control Flow Graphs used by Nanda and Ramesh cannot be used for Behavior Trees since threads in Behavior Trees do not always have a distinct end point.

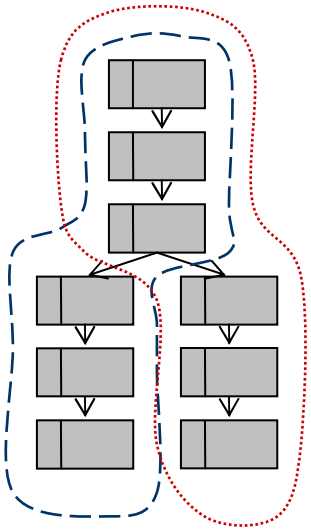


Figure 42. Threads in Behavior Trees

As explained in Section 2.2.3, a second improvement was suggested by Nanda and Ramesh (2006), for creating more precise slices in the presence of nested loops. The problem arises when a node *n* is found to be transitively dependent on another node *m* even though the variable update performed at *m* will always be overridden by another node *p*, an ancestor of *n*. Again, this problem does not apply to Behavior Trees. Threads in Behavior Trees operate fully concurrently and the order in which the nodes in parallel threads interleave is not fixed. Therefore, this issue does not arise because the possibility that node *m* may execute *after* node *p* cannot be ruled out.

4.2 More Precise Slices

The threaded witness approach described in the previous section can still lead to imprecise slices. In this section another approach is presented that extends the concept of threaded witnesses to remove further nodes from the slice. The main difference is that this approach considers the entire maximal path before deciding whether or not to include a node, instead of only the path up to the node.

Suppose that a criterion node is transitively dependent on another, *n*. Now, suppose that the path from the criterion node to *n* does not form a threaded witness. According to the previous approach, *n* will not be included in the slice. However, there may be other nodes on the dependency path which also cannot influence the criterion node, since they are solely dependent on *n*. If it is known that *n* cannot execute before the criterion node, these other nodes cannot execute either. The threaded witness approach cannot identify such nodes, because the path up to these nodes *does* form a threaded witness.

This situation occurs because each node is considered separately to determine if it should be included in the slice. There is no way of identifying the case where the path up to a given node may form a valid threaded witness despite the fact that the rest of the path does not. In this section, a different approach is presented that considers entire dependency paths instead of single nodes at a time.

Example.
Consider again the Behavior Tree in Figure 40 and its corresponding dependency graph in Figure 41. The node `D[d]` was left out of the slice since the path up to it does not form a threaded witness. The paths up to each of the other nodes form threaded witnesses, so these nodes were all added to the slice. However, all the other nodes on the path also cannot influence `C[c]`, due to their dependency to `D[d]`.

■

4.2.1 Infeasible Paths

The set of maximal paths in the dependency graph ending at a given node is given by the function *depPaths*, as described in Definition 36. In other words, *depPaths*(*n*) returns the set of paths starting at node *n* and either terminating at a node with no incoming edges (i.e. no dependencies) or containing a cycle. This is accomplished in the definition by stating that if the first node on the path, m_0, is dependent on another, m_j, then the path must contain a cycle, i.e. m_j must be on the path.The resulting set of paths contain all the nodes that the specified node is dependent on.

DEFINITION 36. DEPENDENCY PATHS
For a dependency graph *G* and a node *n*,
$depPaths(n) = \{ \pi \in paths(G) \mid \pi = < m_0, m_1, ..., m_k, n >$, where if $\exists\, m_j$ such that $m_j \xrightarrow{d} m_0$ then $m_j \in \pi$ $\}$.

■

The original definition of a slice can be reformulated using *depPaths* as given in Definition 37. Note that if a criterion node n_c has no dependencies, *depPaths*(n_c) would return a single path containing only n_c.

DEFINITION 37. SLICING BEHAVIOR TREES

A slice set $nodes\_slice_{\varphi}(B)$ of a BT control flow graph *B*, with respect to a formula φ, with a criterion C_{φ}, is defined as:

$nodes\_slice_{\varphi}(B) = \{n_x \mid \exists\, n_c \in C_{\varphi}$ where $\exists\, \pi \in depPaths(n_c)$ and $n_x \in \pi\}$.

■

A path is known as *infeasible* if it is a member of *depPaths*(n_c) for some $n_c \in C_{\varphi}$ and does not form a threaded witness, as stated in Definition 38. A path is termed *feasible* iff it is not infeasible. If such a path is discovered, then *none* of the nodes on that path are added to the slice during that traversal of the graph. This does not, however, prevent nodes that are able to execute from being included in the slice. Recall that a node may belong to more than one path in the dependency graph. If a node on an infeasible path is still able to execute as it also belongs to a feasible path, then it will be included in the slice when the other path is explored on a future traversal of the graph.

DEFINITION 38. INFEASIBLE PATHS

For a node $n \in C_{\varphi}$, for a path $\pi \in depPaths(n)$, π is known as *infeasible* with respect to *n* iff it is not a threaded witness. The function $infPath_n(\pi)$ returns true iff π is infeasible with respect to *n*.

■

Example.
Returning to the example in Figure 40, *depPaths*(C[c]) would return the entire path shown in Figure 41. Due to the conflict between D[d] and C[c], the path is not a threaded witness. Therefore, it is infeasible and none of the nodes will be added to the slice.

■

If all the dependency paths from a criterion node, n_c, that contain a particular node, n_x, are infeasible, then n_x is designated as infeasible with respect to the criterion node as given in Definition 39. This means the node cannot influence the criterion node. The definition also labels nodes which have no dependency path to the criterion node as infeasible. This is appropriate, since such nodes will not have any influence over the criterion node.

DEFINITION 39. INFEASIBLE NODES

For a node n_x and a criterion node $n_c \in C_\varphi$, the function $inf(n_x, n_c)$ returns true iff $\forall \pi \in depPaths(n_c)$ such that $n_x \in \pi$, $infPath(\pi)$.

■

The definitions presented so far are useful for identifying nodes which are unable to influence the criterion node because all of the associated dependency paths are infeasible. As well as this, there are cases where a node cannot influence the criterion node even though it is on a feasible dependency path. These cases involve conditional nodes and synchronisation nodes. Assume there is a criterion node n_c which is control dependent on a conditional node n_g. If the conditional node is found to be infeasible, n_c can never execute, regardless of whether or not it has other dependencies which lead to valid paths. A similar situation occurs for synchronisation nodes, because a node cannot execute until all of its synchronising nodes are ready to execute as well. The control and synchronisation dependency types are effectively stronger than other dependencies, since the dependent node *requires* the other in order to execute, whereas for other dependencies, the dependent node may still be able to execute without it. For example, a guard node may be able to execute even if the corresponding state realisation is infeasible, if there is another state realisation performing the same action. For this reason, control and synchronisation dependencies must be handled differently than the other dependency types. Otherwise, in the above situation, if the conditional node n_g was not added to the slice as it is infeasible, the criterion node n_c will be able to execute unconditionally in the slice, despite being restricted by n_g in the original model. This can obviously lead to many traces in the slice which are impossible in the original model.

The solution to this problem is to label nodes as *strongly infeasible* if they are synchronisation-dependent or transitively control-dependent on an infeasible node. Definition 40 gives the formal definition of strong infeasibility.

DEFINITION 40. STRONG INFEASIBILITY

For a node n_x and a criterion node $n_c \in C_\varphi$, the function $strongInf(n_x,n_c)$ returns true iff $\exists\, n_a$ such that $n_a \overset{cd^+/sd}{\rightarrowtail} n_x$ and ($inf(n_a, n_c)$ or $strongInf(n_a, n_c)$).

■

The infeasibility due to conditional nodes is also transitive. If a node *n* is termed strongly infeasible, then any nodes which are control dependent on *n* are also strongly infeasible, since they cannot execute without *n* either. Since control dependence occurs between a conditional node and its descendents, the result is that all the descendents of a strongly infeasible node will be labelled strongly infeasible. The descendents of a synch infeasible node will all be labelled strongly infeasible, due to their control dependence to the synchronisation node.

The final method for creating the slice is summarised in Definition 41. As in the original definition of slicing, the slice is the set of nodes that can be reached via a backwards search starting from the criterion node. The previous definition of slicing using threaded witnesses ignores individual nodes that cannot be reached by a valid path from the criterion node. This definition instead ignores *all* nodes along maximal paths where the path is not a threaded witness. A node is only added to the slice if it is on a dependency path from a criterion node which is not infeasible and if the node is not conditional infeasible or synchronisation infeasible with respect to the criterion node. The extra requirements about strong infeasibility ensure that even if there is a valid dependency path, the node will not be added if it is strongly infeasible.

DEFINITION 41. NODES_INF

For a slice $G_2 = \langle N,E, start, end\rangle$ produced from a BT control flow graph G_1 for a formula φ, the slice set produced by removing infeasible paths, $nodes\_inf_\varphi(G_1)$, is defined as:

$nodes\_inf_{\varphi}(G_1) = \{n_x \in N \mid \neg\, inf(n_x, n_c)$, for some $n_c \in C_{\varphi}$, and $\neg\, strongInf(n_y, n_c)\} \cup \{n_c \mid n_c \in C_{\varphi}$ and $size(depPaths(n_c)) = 0\}$.

■

DEFINITION 42. SLICE_INF

Let B be a transition system corresponding to a BT control flow graph G and S be a transition system such that $S = slice_{\varphi}(B)$ for some formula φ. Then, the function $slice\_inf_{\varphi}(S)$ returns the transition system of the slice created from the slice set $nodes\_inf_{\varphi}(G)$.

■

After finding all the nodes to add to the slice, a criterion node may itself not have been added to the slice. There are two reasons why this can occur: either all of its dependencies were on infeasible paths or it does not have any dependencies. In the latter case, the criterion node would not be in the slice because no valid dependency path was found from it. This is the reason for the second part of the definition of a slice, i.e. $\{n_c \mid n_c \in C_{\varphi}$ and $size(depPaths(n_c)) = 0\}$. Any criterion nodes which never had any dependencies must be added to the slice without any further considerations. This implies that the criterion node is always executable.

The other possibility is that the criterion node has dependencies, all of which lie on infeasible paths. This means that the criterion node is known to be unreachable. Since the criterion is a temporal logic theorem, there would normally be several criterion nodes, so the remaining criterion nodes are used for creating the slice. If it is the only criterion node representing that component, then that part of the formula is replaced with "false". For example, if the criterion is $\mathsf{G}(P = p \Rightarrow Q = q)$, and if all the state-realisation nodes involving P are unreachable, then the criterion is replaced with $\mathsf{G}(false \Rightarrow Q{=}q)$.

The following two examples illustrate how infeasible paths can be used to reduce the size of the resulting slice. The first example contains a graph with two terminating dependency paths and the second example contains a graph with a cyclic dependency path.

Example.

As an example, consider the Behavior Tree in Figure 43. Assume the criterion is {C}, which means that the slicing process would begin with the node C[c]. This node is control dependent on the node B?b?. B?b? is interference dependent on two nodes: the B[b] in the middle thread (labelled with the number 1 for ease of explanation) and the B[b] in the last thread (labelled with the number 2). In the following, the middle B[b] node will be referred to as B1 and the other B[b] node as B2. The node B1 is control dependent on D?d?, which is in turn data dependent on D[d] in the left thread.

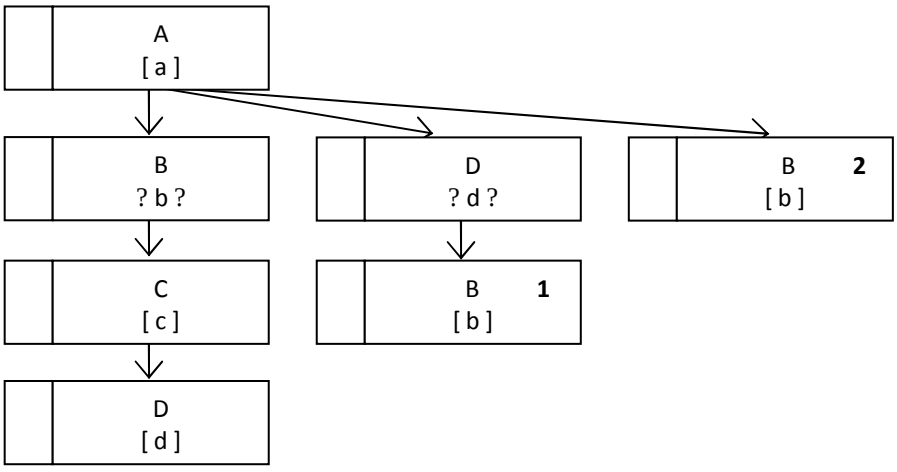


Figure 43. Example Behavior Tree

The dependency graph for this example is shown in Figure 44. Since there is no reversion, D[d] cannot execute before C[c], so it cannot influence the slicing criterion. As in the previous section, the notion of threaded witness can be used to identify this situation. The D[d] node will not be included in the slice because the chain of dependencies from D[d] to C[c] does not form a threaded witness. However, the path upto D?d?, which includes the nodes D?d?, B1, B?b? and C[c], does form a Threaded Witness. Due to this, all these nodes will remain in the slice. The resulting slice is imprecise, because there is no use in including B1 and D?d?. These nodes cannot execute unless D[d] does, so they cannot influence the criterion node. On the other hand, B?b? can influence the criterion node, due to its dependency to B2. If B2 executes, the guard B?b? will be satisfied. The infeasible path approach can correctly identify which nodes to keep in this situation. There are two maximal paths in the dependency graph starting at C[c]: one ending at D[d] and one ending at B2. The node B?b? belongs to both of these paths. The path ending at D[d] is infeasible, so none of the nodes on that path will be added to the slice. The second path is not infeasible, so the nodes C[c], B?b? and B2 will be added to the slice, which produces the desired result.

■

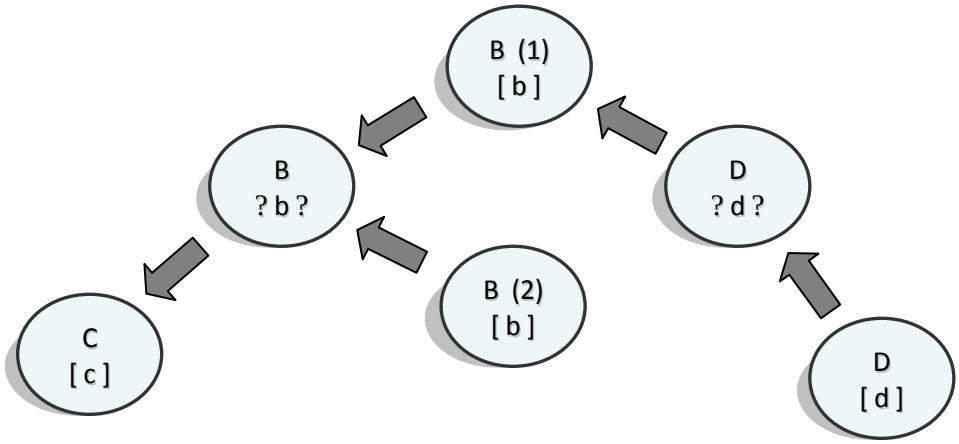


Figure 44. Dependence Graph for the Behavior Tree in Figure 43.

Example.

For a second example, consider the Behavior Tree shown in Figure 45. Again assume that the slicing criterion is C[c]. The dependency graph for this Behavior Tree is shown in Figure 46. As seen in the figure, there is a loop in the dependency graph. This is because the node C?c? depends on the initial criterion node C[c]. Since there are no reversions or reference nodes in the tree, it is impossible for both C?c? to be dependent on C[c] and for C[c] to be dependent on C?c?. The threaded witness approach identifies this conflict as being caused by the second occurrence of C[c]. This results in all of the nodes remaining in the slice.

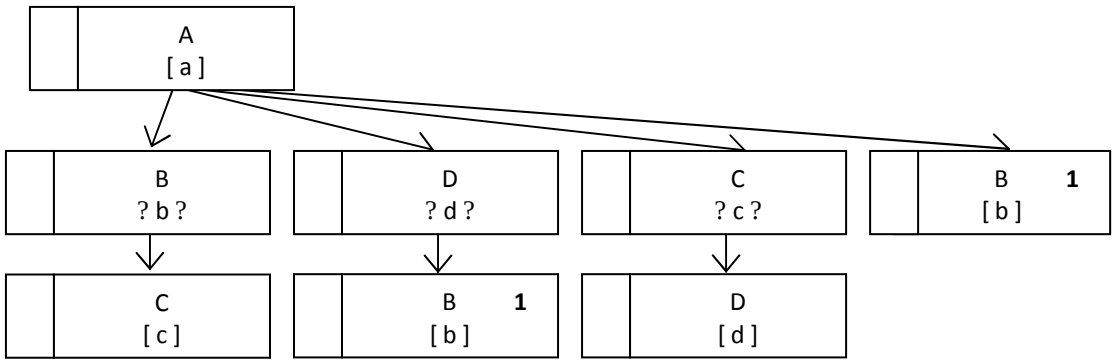


Figure 45. Example Behavior Tree.

Using the infeasible path approach, there are two maximal paths in the graph: the cyclic path starting and ending at C[c] and the path starting at C[c] and ending at B2. The cyclic path forms an infeasible path. Therefore, none of these nodes will be added to the slice. On the other hand, the path ending at B2 does not form an infeasible path, so the nodes C[c], B?b? and B2 will be all added to the slice. This is correct since these are the only nodes that can influence the criterion node. The resulting slice has far fewer nodes than the slices obtained by the threaded witness method.

■

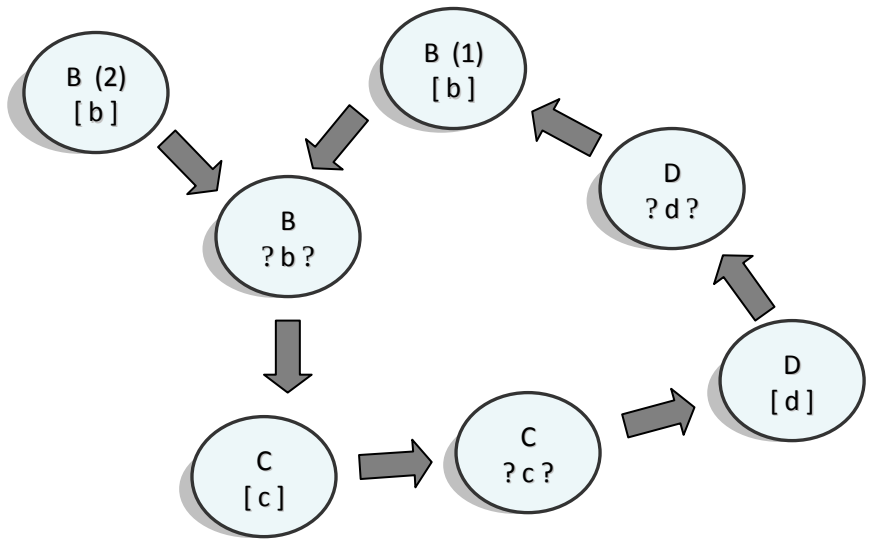


Figure 46. Dependence Graph for the BT in Figure 45.

Initialisation Nodes

Even though a node may have had one or more dependencies that were ignored using the infeasible paths method, it will often still have a dependency to an initialisation node. For example, in Figure 44, the node D?d? would also be dependent on the initialisation node for the component *D*. Since D?d? now has no other dependencies, the guard will be guaranteed to be satisfied if the initial value for *D* is *d*, and would not be satisfied otherwise. Using a simple evaluation of the initial node, the outcome of the guard can be decided. If the guard cannot be satisfied, then the node will not be able to influence the slicing criterion and can be removed. This also applies to attribute nodes; see the Section 4.4.1 for an example of a program involving attributes.

4.3 Proof of Correctness

The infeasible path approach produces more precise slices. In this section, it will be shown that the slices produced are also correct, an essential requirement if the slices are to be used for verification. As was done in the previous chapter, the notion of bisimulation will be used to provide the correctness result. However, unlike the earlier proof, which used a form of weak bisimulation, this proof uses strong bisimulation. A slice is bisimilar to the slice with infeasible paths removed. The strong bisimilarity arises from the fact that the infeasible paths are paths which could never have executed, even in the normal slice. Therefore, despite the normal slice containing extra nodes, the actual behaviour of both systems is exactly the same.

THEOREM 3.
Let G_1 be a slice and G_2 be the slice obtained after infeasible paths are removed from G_1. Then, the transition system *S* corresponding to G_1 is bisimilar to the transition system *S-Inf* corresponding to G_2.

i.e. $S \approx_b$ *S-Inf*.

Proof.

Let $S = (\mathcal{S}_1, AP_1, \mathcal{T}_1, \mathcal{L}_1, \mathcal{N}_1, \longrightarrow_1)$ and *S-Inf* $= (\mathcal{S}_2, AP_2, \mathcal{T}_2, \mathcal{L}_2, \mathcal{N}_2 \longrightarrow_2)$. In the following, let *s*, *s'*, s_0, s_1, ... range over $\mathcal{S}_1$ and *t*, *t'*, t_0, t_1, ... range over $\mathcal{S}_2$.

$\forall n \in init(S), \exists m \in init(S\text{-}Inf)$.

$\Rightarrow \forall s_i \in \mathcal{T}_1, \exists t_i \in \mathcal{T}_2$ such that $\mathcal{L}(s_i) = \mathcal{L}(t_i)$.

A relation $\mathcal{R}$ can be constructed such that $\forall s_i$ and t_i as above, $(s_i, t_i) \in \mathcal{R}$, and

$\forall s, t$ such that $s \mathcal{R} t$, if $\exists s' \in \mathcal{S}_1, \exists t' \in \mathcal{S}_2$ and $\exists n \in \mathcal{N}_1$, such that $s \xrightarrow{n} s'$ and $t \xrightarrow{n} t'$, then $s' \mathcal{R} t'$.

To show that $\mathcal{R}$ is a bisimulation, the following must hold:

$\forall s, t$ such that $s \mathcal{R} t$,

(i) $\mathcal{L}(s) = \mathcal{L}(t)$,

(ii) if $\exists s' \in \mathcal{S}_1$ and $\exists n \in \mathcal{N}_1$ such that $s \xrightarrow{n} s'$, then $\exists t' \in \mathcal{S}_2$ such that $t \xrightarrow{n} t'$, where $s' \mathcal{R} t'$ and

(iii) if $\exists t' \in \mathcal{S}_2$ and $\exists n \in \mathcal{N}_2$ such that $t \xrightarrow{n} t'$, then $\exists s' \in \mathcal{S}_1$ such that $s \xrightarrow{n} s'$, where $s' \mathcal{R} t'$.

(i) As shown above, $\forall s_i \in \mathcal{T}_1$, and $t_i \in \mathcal{T}_2$ such that $s_i \mathcal{R} t_i$, $\mathcal{L}(s_i) = \mathcal{L}(t_i)$.

$\forall s'', t''$ such that $s'' \mathcal{R} t''$ and $\mathcal{L}(s'') = \mathcal{L}(t'')$, if $s \xrightarrow{n} s'$ and $t \xrightarrow{n} t'$,

$\Rightarrow \mathcal{L}(s) = \mathcal{L}(s'') + updates(n)$ and $\mathcal{L}(t) = \mathcal{L}(t'') + updates(n)$.

$\Rightarrow \mathcal{L}(s) = \mathcal{L}(t)$.

(ii) $\forall s, t$ such that $s \mathcal{R} t$, if $s \xrightarrow{n} s'$, then $\exists t'$ such that $t \xrightarrow{n} t'$, where $s' \mathcal{R} t'$.

By contradiction, assume $\exists s, t, s'$ such that $s \mathcal{R} t$ and $s \xrightarrow{n} s'$ but $\nexists t'$ such that $t \xrightarrow{n} t'$ and $s' \mathcal{R} t'$.

$\Rightarrow \exists n$ such that $s \xrightarrow{n} s'$but $\nexists t'$ such that $t \xrightarrow{n} t'$.

Case 1: $n \notin \mathcal{N}_2$.

$\forall n_c \in C_\varphi$, either $inf(n, n_c)$ (Case 1a) or $strongInf(n, n_c)$ (Case 1b).

Case 1a: $\forall n_c \in C_\varphi$, $inf(n, n_c)$

$\Rightarrow \forall \pi \in depPaths(n_c)$ such that $n \in \pi$, $infPath(\pi)$.

$\Rightarrow \nexists s'$ such that $s \xrightarrow{n} s'$,
which contradicts the assumption.

Case 1b: $\forall n_c \in C_\varphi$, $strongInf(n, n_c)$

$\Rightarrow \exists n_a$ such that
$n_a \overset{cd/sd}{\rightarrowtail} n_x$ and $inf(n_a, n_c)$.

$inf(n_a, n_c) \Rightarrow \forall \pi \in depPaths(n_c)$ such that $n_a \in \pi$ and $infPath(\pi)$.

$\Rightarrow \nexists s_j, s_{j+1}$ such that $s_j \xrightarrow{n_a} s_{j+1}$

$\Rightarrow \nexists s'$ such that $s \xrightarrow{n} s'$, by Definition 16 and Definition 20,
which contradicts the assumption.

Case 2: $n \in \mathcal{N}_2$

$\Rightarrow n \in ready_v(s)$ but $n \notin ready_v(t)$, for some thread *v*.

$\Rightarrow \exists n_x \in \mathcal{N}_1$ such that $n_x \overset{d^+}{\rightarrowtail} n$ and $n_x \notin \mathcal{N}_2$.

$\Rightarrow$ either $inf(n_x, n_c)$ or $strongInf(n_x, n_c)$.
$\Rightarrow \nexists s_j, s_{j+1}$ such that $s_j \xrightarrow{n_x} s_{j+1}$ in S.
$\Rightarrow n \notin ready_v(s)$,
which contradicts the assumption.

(iii) $\forall s, t$ such that $s \mathcal{R} t$, if $t \xrightarrow{n} t'$, then $\exists s'$ such that $s \xrightarrow{n} s'$, where $s' \mathcal{R} t'$.
By contradiction, assume $\exists s, t, t'$ such that $s \mathcal{R} t$ and $t \xrightarrow{n} t'$ but $\nexists s'$ such that $s \xrightarrow{n} s'$ and $s' \mathcal{R} t'$.
$\Rightarrow \exists n$ such that $t \xrightarrow{n} t'$but $\nexists s'$ such that $s \xrightarrow{n} s'$.
Case 1: $n \notin \mathcal{N}_1$.

$\mathcal{N}_2 \subseteq \mathcal{N}_1$
$\Rightarrow n \in \mathcal{N}_1$,
which contradicts the assumption.

Case 2: $n \in \mathcal{N}_1$

$\Rightarrow n \in ready_v(t)$ but $n \notin ready_v(s)$, for some thread v.
There are two cases:
Case 2a: $\exists n_x \in \mathcal{N}_2$ such that $n_x \overset{d^+}{\rightarrowtail} n$ and $n_x \notin \mathcal{N}_1$.

$\mathcal{N}_2 \subseteq \mathcal{N}_1$
$\Rightarrow n \in \mathcal{N}_1$,
which contradicts the assumption.

Case 2b: $\exists n_x \in \mathcal{N}_1$ such that $n_x \overset{d^+}{\rightarrowtail} n$ and $n_x \notin \mathcal{N}_2$, where n_x being in $\mathcal{N}_2$ prevents n from executing.
$\Rightarrow n_x \overset{cd^+}{\rightarrowtail} n$ or $n_x \overset{sd}{\rightarrowtail} n$
$n_x \notin \mathcal{N}_2$
$\Rightarrow \forall n_c \in C_\varphi$, $inf(n_x, n_c)$ or $strongInf(n_x, n_c)$
$\Rightarrow strongInf(n, n_c)$, by Definition 40
$\Rightarrow n \notin \mathcal{N}_2$, by Definition 41,
which contradicts the assumption.

□

4.4 Slicing Algorithm

The algorithm for removing infeasible paths is given on the following page. It is based on the algorithm given by Krinke for threaded witnesses. The algorithm recursively explores each node *n* in the dependency graph. An array of nodes *lastReached* is maintained, which records the last node reached so far in each thread. Additionally, the algorithm maintains a list *path*, which contains the nodes that were traversed so far to reach *n*. Another parameter is *last*, which is the node that is dependent on *n* that called this function at an earlier iteration. The final parameter is a boolean *criterion*, which is true if *n* is the criterion node that started this search.

First, lines 3-5 check that *n* is reachable by comparing it with the nodes in the *lastReached* array. If it is reachable, the boolean *inf* is set to false. Next, line 6 checks whether the *inf* boolean is false and the *strongInf* boolean, which is an attribute of the node, is also false. The *strongInf* variable indicates whether the node has already been identified as being strongly infeasible. In such cases, the node will not be included into the slice even if there is another valid dependency path containing it. If *inf* and

strongInf are both false, *n* is added to the current *path* at line 7. Then, lines 8-9 update the *lastReached* array to contain *n* in each of its threads. Lines 10-14 then recursively call the same function for each of the nodes *m* on which *n* is dependent. If, for some *m*, the recursive call returns *false*, it indicates that the path through *m* is infeasible. If one of the calls returns *true*, this is recorded in lines 13-14 using a boolean *pathExists*. Additionally, if *n* is control or synchronisation dependent on some node *m* and *m* turns out to be infeasible, then the *strongInf* attribute of *n* would have changed. Thus, line 15 checks whether the *pathExists* variable is true and makes sure that *n*'s *strongInf* attribute is still *false*. If this is the case, it means there is a feasible path containing *n*. At line 16 it is checked whether *n* is the criterion node that is the starting point of this backwards traversal of the dependency graph. If so, the entire feasible path is added to the slice, in line 17. Otherwise, the function simply returns true, to indicate to the previous node that there was a feasible path upto *n*.

Unlike the algorithm for creating the slice, given in Chapter 3, this algorithm does not prevent a node from being explored more than once. Therefore, the algorithm is exponential in the worst case, in the case where each node is dependent on every other node. However, in practice such a situation would rarely occur. Furthermore, as future work it is planned that a polynomial-time algorithm can be devised, which stores the required information in such a way as to avoid multiple traversals of the same paths in the graph.

```
     bool checkInf (array lastReached, node n, list path, node last, bool criterion){
1      t = getThreads(n);
2      bool inf = true;
3      for each t do
4          if (reachable(lastReached[t], n)) then
5              inf = false;
           end if
       next t
6      if (inf == false AND n.strongInf == false) then
7          path.add(n);
8          for each t do
9              lastReached[t] = n;
           next t
10         bool pathExists = false;
11         for each m in n.getDep() do
12             bool b = checkInf(lastReached, m, path, n);
13             if (b) then
14                 pathExists = true;
               end if
           next m
15         if (pathExists AND n.strongInf == false) then
16             if (criterion == true) then
17                 includePath(path);
               end if
18             return true;
           end if
19         return false;
20     else   // if inf == true or n.strongInf == true
21         if (depType(p, n) in {cd, sd} then
22             p.strongInf = true;
           end if
23         return false;
       end if
```

4.4.1 Side Note: Application to Programs

This approach is also relevant to programs. Consider the following extract of a program:

```
Thread 1                          Thread 2
if (y = 5){  (1)                  if (z = 3){  (4)
     c:=4;   (2)                       y:=5;   (5)
}else{                            }
     z:=3;   (3)
}
```

Assume that the slicing criterion is {c}, so statement 2 is the starting point for slicing. The dependency path through the corresponding PDG is <3, 4, 5, 1, 2>. Statements 3 and 2 do not form a Threaded Witness, so this is an infeasible path. Statement 4 is also dependent on the initial value of *z* and statement 1 is also dependent on the initial value of *y*. After ignoring the nodes in the infeasible path, nodes 2, 1, 5 and 4 all still remain in the slice due to these dependencies to initialisation nodes. However, a simple analysis could decide whether or not the guards at 4 and 1 hold. For example, if *z* is initially set to 1, the guard at 4 will never hold, so it can be removed. This illustrates that the Infeasible Path approach can reduce the size of program slices, as well as BT models.

This chapter described a method for reducing slices further, by removing nodes which lie on infeasible paths. The technique can be of use when a slice is still too large for model checking. The method was proved to produce correct slices. The next chapter will introduce a novel method for handling properties containing the *next* temporal logic operator.

5 SLICING WITH THE NEXT OPERATOR

The slicing algorithms described so far ensure the preservation of CTL^*_{-X} properties, i.e. properties without the *next* operator, X. This chapter introduces an approach for including extra nodes into the slice, in order to preserve *all* CTL^* properties. The approach is based on the observation that properties with the X operator may not be preserved on the slice because certain stuttering nodes have been deemed irrelevant and removed. By identifying these essential stuttering nodes and re-inserting them into the slice, the X properties can be preserved. The technique for preserving X properties can be applied for any language, by utilising a proposed new form of branching bisimulation called *next-preserving branching bisimulation*. Section 5.1 introduces the problem and discusses other solutions for it. Section 5.2 describes the proposed approach and gives the definitions for next-preserving branching bisimulation. Section 5.3 provides a proof which shows that next-preserving branching bisimulation preserves full CTL^*. Finally, Section 5.4 demonstrates the application of next-preserving branching bisimulation by describing how to create a next-preserving branching bisimilar slice from a Behavior Tree.

5.1 The problem of removing stuttering nodes

When slicing is used for reduction of models for verification purposes, it is essential that the final slice preserves the same properties as the original model. In other words, a property holds on the original model if and only if it holds on the slice. Therefore, the range of properties that satisfy this requirement should be as large as possible, for the benefit of the user who wants to verify a property. Unfortunately, slicing does not normally preserve properties containing the X operator. This problem is not unique to Behavior Tree slicing; all forms of published slicing algorithms do not claim to preserve properties with X. Refer to Section 1.2 for a discussion about other slicing algorithms.

The reason for this problem can be illustrated by a simple example. Consider the trace of behaviour shown in Figure 47. Assume that at state s_0, the following property is to be verified: Xp. Obviously, the property holds, since s_1 satisfies p. However, imagine that a slicing algorithm removed s_1, because it is an unnecessary stuttering step, identical to the previous step. Now, the property no longer holds because s_2 does not satisfy p.

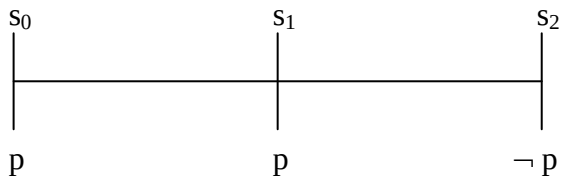


Figure 47. The stuttering problem.

This problem arises because slicing algorithms only collect nodes which are either observable or in some way influence an observable node. Properties with the *next* operator, however, can even be

influenced by nodes which do not have any impact on the variables in the property. There is no way to identify such nodes using normal slicing techniques.

Vasudevan et al. (2005) proposed an approach termed *antecedent conditioned slicing*, for slicing properties that conform to specific formats. This included formulas with the *next* operator in the format $\mathsf{G}(p \Rightarrow \mathsf{X}^k q)$, where p and q are formulas and X^k represents the X operator applied k times. Based on *conditioned slicing* (Canfora, et al., 1998), which restricts a slice to statements satisfying a given condition, this approach restricts slices to statements in which the antecedent of the formula holds. However, they did not present a proof of correctness nor give details of how they ensure the preservation of such formulas when stuttering steps are removed.

Many authors choose to simply restrict formulas to those without X. Lamport (1983) argued against the use of the *next* operator. His argument was that a step has no meaning in real-life continuous-time systems as it is a concept used in models. He argued that a step is only of interest if it represents a change in some property of the system and therefore stuttering steps should not be distinguished. Lamport suggested that any property involving the *next* operator could be re-stated in terms of some other characteristics of the system. For example, instead of stating that an event *x* should happen in the next step, it should be stated that *x* will happen *before* some other event occurs. Dams (1996) gave a similar argument against the *next* operator.

Despite these arguments, the *next* operator is often useful in practice. Since model checking is performed on discrete models, it is often useful to specify properties using the notion of steps, where each step represents a certain period of time, or a certain phase of the system's behaviour. In particular, using the *next* operator can be an effective technique for specifying a requirement that something will occur in a certain period of time. It is not always possible to state the property in terms of other events in the system, as suggested by Lamport. It may be the case that an event must happen within a certain period of time or phase of the system operation. The modeller may decide that this can be represented by a number of steps, without it being restricted to any particular steps. The closest alternative would be to use the *eventually* (F) operator, but for some requirements this may be too weak, as it does not provide any guarantee of the period in which something will occur. The *next* operator is therefore essential for specifying such properties. An example of this is the property $\mathsf{G}(CH_4 = high \Rightarrow \mathsf{XXX}(alarm = sounded))$, used in the mine pump case study presented in Section 6.2. In other words, it is always the case that when the methane is at a high level, three steps later the alarm should have sounded. Obviously, in the real system there would be no notion of steps, so "three steps later" would be meaningless. However, it is known that under normal operation, the system should perform certain actions before sounding the alarm. Due to the level of granularity of the model, the actions correspond to three steps in the model, so can therefore be referred to by three instances of the *next* operator in the formula.

The mine pump example introduces the main difficulty with using the *next* operator on sliced systems: the difference in granularity between the original model and the slice. Since irrelevant steps are removed, a slice step may represent multiple steps in the original model. This presents an obvious difficulty when verifying properties that refer specifically to a certain number of steps. The property itself gives no clue as to whether the user was referring to steps in the original model or in the slice. However, recall that the motivation of slicing is to enable the verification of properties that may have not been viable on the original model. Therefore, if a user specifies that *x* occurs within *m* number of steps, *m* is referring to the steps in the original model. The ideal goal is to verify such a property on the slice, despite the differences in the two notions of steps.

Regardless of the arguments for and against the use of the *next* operator, since many common properties contain *next*, it is beneficial to provide the option of slicing with the *next* operator, for those who would find it useful.

5.2 Process of Slicing with the Next Operator

As explained in the previous section, properties containing the *next* operator are referring to steps in the original model, which may not necessarily correspond to steps in the slice. However, it is only necessary to preserve the correct number of steps in the regions of the transition system which are relevant for the property. For example, in the mine pump case described above, it is only important to

ensure that the state of the *alarm* component in the slice three steps after the CH_4 component reaches *high* is the same as it was in the original three steps after CH_4 = *high*. At these specific locations, the steps in the slice should correspond exactly to steps in the original model, but at all other locations, the size of the steps are irrelevant.

The goal of this approach is to replace certain nodes that were removed by slicing, in order to ensure that the size of steps are preserved at these important locations. The question is, how does one determine which nodes should be replaced? The answer lies in the observation that the only time a property Xφ will not be preserved is when at some point, the next state evaluates φ differently than the state two steps later, i.e. the next next step. In fact, it turns out that there are only a finite number of places where this phenomena occurs.

5.2.1 Identifying the relevant locations

Formulas expressed in CTL* which contain the X operator will always contain either an E or A operator somewhere before the X. Recall from Section 2.1 that the X operator is defined over paths, not states, which is the reason why there will always be an E or A. Note that unlike for CTL, the E or A does not have to necessarily be the operator immediately preceding the X, for example, the formula E(p ∧ Xq) is valid in CTL* but not in CTL. Nevertheless, there will always be an outer E or A surrounding any sub-formula containing X. There are therefore two possibilities to consider: if the X refers to a specific path (E) or all paths (A).

First consider the simplest case, where the formula is AXp, where p is an atomic proposition. Consider the transition system shown in Figure 48. The diagram on the left is the original model and the diagram on the right is its slice. In the original system, at state s_1, p is *true*, so $s_0 \vDash$ AXp. In the slice, state s_1 has been removed, so s_0 transitions directly to s_2. This presents a problem, since s_2 does not satisfy p, and thus s_0 does not satisfy AXp, which would result in a false counterexample. An identical situation occurs for EXp.

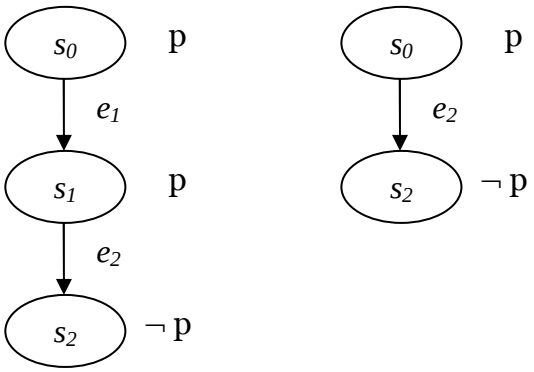


Figure 48. A Model and its Slice

The converse problem also exists, whereby a false positive result could be obtained. Consider the transition system in Figure 49, which is the inverse of the previous transition system. In this case, $s_0 \nvDash$ AXp in the original model, but $s_0 \vDash$ AXp in the slice. Again, the same situation occurs for EXp.

The two cases above are not just simple examples; they demonstrate the general problem. In a normal system, there would probably be multiple paths emanating from s_0, defining the many possible paths of execution of the system from that point. If the formula is AXp, then paths that conform to the first case must be preserved, in order to prevent spurious counterexamples. As well as this, paths that conform to the second case must also be preserved, to ensure that a real counterexample is not lost. If the formula is EXp, a path of the first type must be preserved since it may be the only path that satisfies Xp, in which case the formula EXp will only be satisfied if the path is preserved. Similarly, a path of the second type must be preserved since there may be no paths that satisfy Xp. In this case, if Xp holds on the path after slicing, the formula EXp would hold on the slice whereas it did not hold originally.

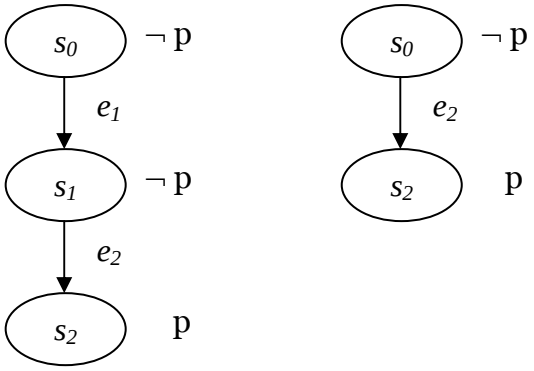


Figure 49. A Model and its Slice

The important thing to observe about the above two examples is that the transition e_2 is an *observable* transition, because it modifies p, which is a variable mentioned in the formula. This is the reason that e_2 remains in the slice in both cases. For these cases, a simple solution would be to locate all observable transitions, and determine whether a stuttering transition executes immediately before it. If so, the stuttering transition must be included in the slice, preserving the original behaviour of the system.

In general, this solution is sufficient for almost all types of properties. Assume that the property is AXφ or EXφ, where φ is some arbitrary property not containing the X operator, and that in the diagrams above, all p's are replaced with φ's. Consider each of the path operators:

- Let φ = Fp. Assume there is a path on which Fp is true at s_1 but false at s_2. Since only a particular path is being considered, the only way for the status of Fp to change from one state to another is if p became false at s_2. Therefore, the same method for including extra transitions can be used for this situation. The converse case, where Fp is false at s_1 but true at s_2, is not possible. For any particular path, if p can eventually be reached from s_2, then it can be reached from s_1 as well.

- Let φ = Gp. Assume there is a path on which Gp is true at s_1 but false at s_2. By the definition of G, this is not possible. The same logic shows that it is not possible for Gp to be false at s_1 but true at s_2. Therefore, AXGp or EXGp will be preserved in the slice without any extra nodes being added.

For these simple cases, the proposed method for including additional transitions, whereby an extra stuttering transition is included for every observable one, works well. In fact, it will be demonstrated in Section 5.3 that the method is suitable for almost all types of formulas. The only exception is when the formula is of the form AXEφ or EXAφ. Intuitively, this makes sense because a path formula will only change its validity from one state to another if one of the atomic propositions in it change. The sub-formulas Eφ and Aφ are instead evaluated over states and so the same method is not suitable for these types of formulas.

To demonstrate the reason why it does not work for state formulas, consider the diagrams in Figure 50. The transition system shown on the left is the original model and the system on the right is its slice, where the transition e_1 and state t_1 have been removed. Assume the formula is AXE(Fp). There are two possible paths from t_0: the path $<t_0, t_1, t_2, t_3>$ and the path $<t_0, t_1, t_4>$. The formula holds at state t_0 on the original model, because on all of the paths from t_0, the next state is t_1, and from t_1 there exists a path on which p is eventually true. On the slice, however, the formula does not hold. As in the original model there are still two possible paths, but this time the next step after t_0 differs on each path. On one path the next step is t_2, for which there does exist a path where eventually p holds, but on the other path the next step is t_4, from which there is no possible way to reach a state where p holds. Therefore, the formula holds on the original model but not on the slice.

The inverse problem exists for formulas of the form EXA(Fp). Consider again the diagrams in Figure 50. The property does not hold at t_0 on the original model, since for both possible paths, on the next state t_1, the property AFp does not hold. On the other hand, t_1 has been removed from the slice, so the next state differs depending on which path is chosen. On one path, t_2 is the next step, and AFp *does* hold at t_2, so EXA(Fp) holds on the slice.

Note that this problem does not occur for EXEφ or AXAφ. This can be easily seen by considering the diagrams above using the formula AXA(Fp). The problem with AXE(Fp) described previously is that at t_1, there exists a path where Fp holds, but at t_4 in the slice, no such path exists. However, at t_1, AFp does not hold due to the path through t_4, so AXA(Fp) does not hold on both models. Similarly, if the formula is EXE(Fp), then unlike for EXA(Fp), the formula holds on both models, because at t_1, EFp holds whereas AFp does not.

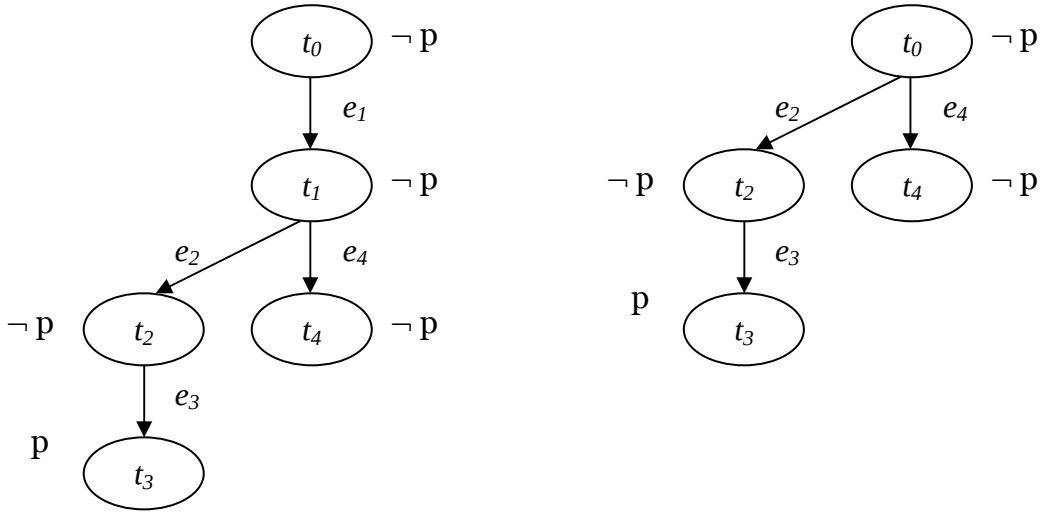


Figure 50. A model and its slice, to illustrate differences in AXE or EXA formulas.

The problems with EXAφ and AXEφ are caused by a state having multiple paths emanating from it, where φ may hold on some of the paths but not on others. Note that the problem would not occur if both paths were still possible after one branch was chosen. For example, if t_3 and t_4 looped back to t_0 in the previous figures. Therefore another criteria for identifying these situations is to determine whether one of the branches leads to a path which is not possible via the other branch.

In conclusion, extra stuttering nodes must be included into the slice before every observable transition and before every branching point, such that one branch leads to a path which is unreachable via the other branch.

5.2.2 Approach for Preserving Next

Based on the previous discussion, a procedure for creating next-preserving slices becomes evident. Extra stuttering steps must be placed into the slice at various locations. Specifically, stuttering steps are required before observable steps and certain types of branching paths. The branches are those where one of the paths performs an observable steps that is either not present on the other path at all or does not occur on the other path within the same number of steps. This is formalised as a function *diffPaths*, defined below, which returns true if and only if there are two paths satisfying the above criteria, starting at the given state.

DEFINITION 43. DIFFPATHS

Let T be a doubly-labelled transition system such that $T = (\mathcal{S}, AP, \mathcal{I}, \mathcal{L}, \mathcal{N}, \longrightarrow)$.

For a state $s \in \mathcal{S}$ and a CTL* path formula φ, $diffPaths_{\varphi}(s)$ iff:

$\exists \rho_1 = < s, a_j, s_j, a_{j+1} ...>$ and $\exists \rho_2 = < s, a_k, s_k, a_{k+1}...>$, where $s_j \neq s_k$ and either:

i) $\exists a_i \in \rho_1$ such that $obs_\varphi(a_i)$ and $a_i \notin \rho_2$ or vice versa, or

ii) $\exists m \geq 0$ and $\exists a_i \in \rho_1 \langle\!\langle s_{j+m}]$ such that $obs_\varphi(a_i)$ and $a_i \notin \rho_2 \langle\!\langle s_{k+m}]$ or vice versa.

■

5.2.3 Number of Stuttering Transitions Required

The next consideration is the number of stuttering nodes required. The number depends on the formula to be verified, specifically the number of X operators surrounding each atomic proposition. As will be seen, in order to ensure that the same verification result is obtained on the slice, an extra stuttering step must be included for every X operator. For example, if the formula is AXX*p*, where *p* is some atomic proposition, two stuttering steps are required before any observable transition or branching transitions. If the formula is EX($p \wedge$ X*q*), then two stuttering steps are required. This is because the *q* proposition is nested inside two X operators. Even though the *p* proposition is only surrounded by one X operator, the number of stuttering steps required corresponds to the maximum number of nested X operators. The number of X operators is referred to as the *x-depth* of the formula, given by the following definition, which is based on the definition given in Kučera & Strejček (2005) for LTL formulas.

DEFINITION 44. X-DEPTH

The *x-depth* of a CTL* formula is given by the following, where ψ_1 and ψ_2 are state or path formulas and φ, φ_1 and φ_2 are path formulas:

x-depth(ψ) = 0, where $\psi \in$ AP,
x-depth($\psi_1 \wedge \psi_2$) = *max*(*x-depth*(ψ_1), *x-depth*(ψ_2)),
x-depth($\neg \psi_1$) = *x-depth*(ψ_1),
x-depth(Eφ) = *x-depth*(φ),
x-depth(φ_1 U φ_2) = *max*(*x-depth*(φ_1), *x-depth*(φ_2)),
x-depth(Xφ) = *x-depth*(φ) + 1.

■

5.3 Preservation of Full CTL*

The discussion so far gave a general explanation for why the inclusion of stuttering steps at certain locations allows properties with the X operator to be preserved. This section proves this fact conclusively, by the proposal of a new form of branching bisimulation that requires both transition systems to contain extra stuttering steps according to the proposed approach. This new bisimulation is termed *next-preserving branching bisimulation*. The new bisimulation is an extension of branching bisimulation with explicit divergence, as was discussed in Section 2.2.5. As was seen in that previous section, branching bisimulation with explicit divergence preserves CTL$^*_{-X}$, which is the variant of CTL* excluding the next step operator. If two systems are related by a next-preserving branching bisimulation, it means they are related by a branching bisimulation with explicit divergence, as well as containing extra stuttering steps. Basing the new type of bisimulation on branching bisimulation ensures that it is able to preserve CTL$^*_{-X}$ as well. It remains to be shown that next-preserving branching bisimulation additionally preserves properties that contain X.

The following three definitions explain how the next-preserving branching bisimulation operates over states, transition systems and paths, respectively. A next-preserving branching bisimulation must satisfy two criteria. The first criterion is that if a state *s* is related to a state *t* in another transition system via a next-preserving branching bisimulation, then *s* is related to *t* via a branching bisimulation with explicit divergence. This requires that every observable step taken from *s* must be matched by an

observable step taken from *t*, possibly preceded by any number of stuttering steps, and vice versa. Additionally, every divergent path from *s* must be matched by a divergent path from *t* and vice versa.

For next-preserving branching bisimulation, there is an additional requirement that if *s* is followed by a number of stuttering steps and then an observable step, *t* must also be followed by a number of stuttering steps before the matching observable step and vice versa. The same holds for the case where *s* is followed by a number of stuttering steps and then reaches a state *s'*, such that $diffPaths_{\varphi}(s')$. These are the two cases of the second criterion in the definition. In these cases, the number of stuttering steps required before *t* is the minimum of the *x-depth* of the formula φ and the number of stuttering steps before *s*. The reason for this is that $x\text{-}depth(\varphi)$ stuttering steps are necessary in order to preserve the formula. However, it may be the case that there are less than $x\text{-}depth(\varphi)$ steps after *s*. This indicates that the first system may not have satisfied the formula. Since the aim is to obtain the same verification result using the second system, it is only necessary to have as many stuttering steps as there were after *s*. On the other hand, there may be more than $x\text{-}depth(\varphi)$ steps after *s*, in which case the additional steps are unnecessary.

DEFINITION 45. NEXT-PRESERVING BRANCHING BISIMULATION OVER STATES

Let T_1, T_2 be doubly-labelled transition systems such that $T_i = (\mathcal{S}_i, AP_i, \mathcal{I}_i, \mathcal{L}_i, \mathcal{N}_i, \longrightarrow_i)$, for $i \in \{1,2\}$. A relation $\mathcal{R}$ is a *next-preserving branching bisimulation* with respect to a CTL* formula φ iff

(i) $\mathcal{R}$ is a branching bisimulation with explicit divergence and

(ii) for every $s \mathcal{R} t$, where $s \in T_1$ and $t \in T_2$:

$\forall \rho_1 \in runs(T_1)$, such that $\rho_1 = < s, a_0, s_0, a_1, s_1, ..., s_{j-1}, a_j, s_j>$ and $\forall i$ such that $0 \leq i < j$, $\neg\, obs_{\varphi}(a_i)$, if either:

- $obs_{\varphi}(a_j)$ or
- $diffPaths_{\varphi}(s_{j-1})$, then:

$\exists \rho_2 \in runs(T_2)$, such that $\rho_2 = < t, b_0, t_0, b_1, t_1, ..., t_k, b_k, t_{k+1}>$, $b_k = a_j$, and for some $k \geq min(j - 1, x\text{-}depth(\varphi))$, $\forall i$ such that $0 \leq i < k$, $\neg\, obs_{\varphi}(b_i)$.

Two states *s* and *t* are *next-preserving branching bisimilar*, with respect to a CTL* formula φ, denoted $s \overset{\star}{=}_{\varphi} t$, iff there exists a next-preserving branching bisimulation $\mathcal{R}$ with respect to φ such that $s \mathcal{R} t$.

■

The definition above states that a state *s* is *next-preserving branching bisimilar* to another state *t* iff two criteria hold. The first criterion is that *s* and *t* are related by a branching bisimulation with explicit divergence. The second criterion states that if *s* is followed by *j*-1 stuttering steps to a state s_{j-1}, which is either followed by an observable step a_j or satisfies $diffPaths_{\varphi}$, then *t* must be followed by *k* stuttering steps, where *k* is the minimum of *j* and the *x-depth* of the formula. The following definitions extend this to transition systems and paths.

DEFINITION 46. NEXT-PRESERVING BRANCHING BISIMULATION OF TRANSITION SYSTEMS

Let T_1, T_2 be doubly-labelled transition systems such that $T_i = (\mathcal{S}_i, AP_i, \mathcal{I}_i, \mathcal{L}_i, \mathcal{N}_i, \longrightarrow_i)$, for $i \in \{1,2\}$. T_1 and T_2 are *next-preserving branching bisimilar*, with respect to a formula φ, denoted $T_1 \overset{\star}{=}_{\varphi} T_2$, iff $s_0 \overset{\star}{=}_{\varphi} t_0$ for all $s_0 \in I_1$ and $t \in I_2$.

■

DEFINITION 47. NEXT-PRESERVING BRANCHING BISIMULATION OVER PATHS

A path π_1 is *next-preserving branching bisimilar to a* path π_2, denoted $\pi_1 \overset{\star}{=}_{\varphi} \pi_2$ iff for every $s_i \in \pi_1$, there exists a $t_i \in \pi_2$ such that $s_i \overset{\star}{=}_{\varphi} t_i$ and vice versa.

■

Some auxiliary results will now be established concerning next-preserving branching bisimulation. Using these results, Theorem 4 will prove that next-preserving branching bisimilar systems preserves full CTL*. The following lemma, Lemma 7, demonstrates the existence of next-preserving bisimilar paths when two systems are next-preserving bisimilar. This result arises directly from the definition of next-preserving branching bisimilar transition systems. It will be used later in Theorem 4, for the case where the property involves a E operator, specifying that there exists a path.

LEMMA 7. EXISTENCE OF NEXT-PRESERVING BRANCHING BISIMILAR PATHS

Let T_1, T_2 be doubly-labelled transition systems such that $T_i = (\mathcal{S}_i, AP_i, \mathcal{I}_i, \mathcal{L}_i, \mathcal{N}_i, \longrightarrow_i)$, for $i \in \{1,2\}$.

Then, $T_1 \overset{\star}{=}_\varphi T_2$, $\Rightarrow$ $\forall \pi_1 \in T_1$, $\exists \pi_2 \in T_2$ such that $\pi_1 \overset{\star}{=}_\varphi \pi_2$.

Proof. By induction over the length m of the path $\pi_1 = <s_0, s_1, ..., s_m>$.

Base case: $m = 0$

$T_1 \overset{\star}{=}_\varphi T_2$

$\Rightarrow \forall s_0 \in I_1$, $\exists t_0 \in I_2$ such that $s_0 \overset{\star}{=}_\varphi t_0$, by Definition 46

$\Rightarrow \exists \pi_2 \in T_2$, such that $\pi_2 = <t_0>$, where $\pi_1 \overset{\star}{=}_\varphi \pi_2$.

Induction step:

Assume it holds for $m = k$, i.e. $\pi_1 = <s_0, s_1, ..., s_k>$ and $\exists \pi_2 \in T_2$ such that $\pi_2 = <t_0, t_1, ..., t_j>$ and $\pi_1 \overset{\star}{=}_\varphi \pi_2$.

For $m = k + 1$:

$s_k \overset{\star}{=}_\varphi t_j$, by assumption and Definition 47.

$\Rightarrow \forall s' \in \mathcal{S}_1$, such that $s \xrightarrow{a} s'$, either $s' \overset{\star}{=}_\varphi t_j$ or $\exists t', t'' \in \mathcal{S}_2$ such that $t_j \overset{*}{\dashrightarrow} t'' \xrightarrow{a} t'$

and $s' \overset{\star}{=}_\varphi t'$, by criterion (i) of Definition 45

$\Rightarrow \forall \pi_1' = \pi_1 \frown s'$, $\exists \pi_2' \in \{\pi_2 \frown <t_{j+1}, ..., t'', t'>, \pi_2 \frown < t'>, \pi_2\}$ such that $\pi_1' \overset{\star}{=}_\varphi \pi_2'$.

□

As was discussed previously, for formulas containing an EXA or AXE pattern, normal slicing methods do not ensure the preservation of the formulas. Recall that A is an operator derived from E. The problem arises when there is a state with more than one path emanating from it, where one path leads to the satisfaction of a formula but another does not. If the state was reached by a stuttering step, it may be left out of the slice, which may cause one or more of the paths to become unreachable at the next step, thus changing the outcome of the formula. The proposed solution replaces stuttering steps before any state s for which $diffPaths_\varphi(s)$ holds. In other words, a state s which leads to two different paths, where on one path there is an observable node that either does not occur on the other path within the same number of steps or does not occur at all. The reason for this requirement is that if there are two paths starting from a state s, such that one satisfies a formula but the other does not, then one of the paths must have executed an observable step that the other was unable to match within the required time. This concept is shown by the following lemma, Lemma 8. It is necessary for proving the case where the formula contains an X followed by an E in Lemma 9.

Assume there is a state s which leads to two different paths, one starting with s_1 and the other with s_k. An example is given in Figure 51 below. Assume that there is a state s_j on the first path such that the suffix starting at s_j satisfies the given CTL* path formula, φ. Further assume that on the second path, the suffix starting at s_{k+j}, i.e. j steps after s_k, does not satisfy φ. (Note that the state j steps after s_0 on the first path is compared with the state j steps after s_k, not s_0, on the second path. This is necessary

in order to produce the required conditions needed for Lemma 9.) If these conditions hold, the lemma states that there is an observable step on one of the paths, such that either the observable step is before s_j (or s_{k+j}) and the other path has no identical step before s_{k+j} (or s_j), or the other path has no matching observable step at all.

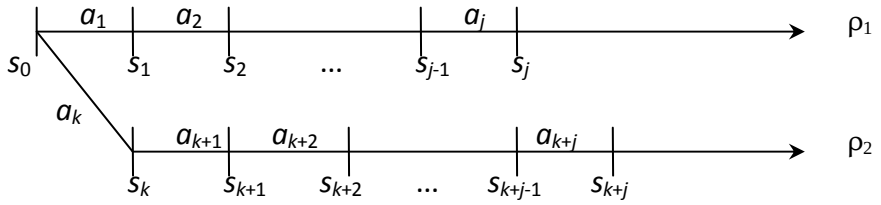


Figure 51. The two paths for Lemma 8.

Example.
Consider again the transition system shown on the left of Figure 50. The state t_1 in the diagram corresponds to s_0 in the lemma, as it has two successors, t_2 and t_4. In the diagram, the path starting from t_4 does not satisfy the formula EFp, whereas the path starting from t_1 does. (Using the terminology of the lemma, t_1 corresponds to s_j and t_4 corresponds to s_{k+j}, where $j = 0$). This example satisfies Lemma 8, since there is an observable step, i.e. t_2 to t_3, on the path $< t_1, t_2, t_3 >$ but there is no matching observable step on the path $< t_1, t_4 >$. ■

LEMMA 8. OBSERVABLE STEPS ON BRANCHING PATHS

Let T be a doubly-labelled transition system such that $T = (\mathcal{S}, AP, \mathcal{I}, \mathcal{L}, \mathcal{N}, \longrightarrow)$.

For a state $s_0 \in \mathcal{S}$ and a CTL* path formula φ,

if $\exists\, \rho_1 = < s_0, a_1, s_1, a_2, s_2 ...>$ and $\exists\, \rho_2 = < s_0, a_k, s_k, a_{k+1}, s_{k+1} ...>$, such that for some $j \geq 0$, $\rho_1[s_j\rangle\rangle \models \varphi$ and $\rho_2[s_{k+j}\rangle\rangle \not\models \varphi$, and $\forall i$ such that $0 \leq i < j$, $\rho_1[s_i\rangle\rangle \models \varphi$,

then $diffPaths_\varphi(s_0)$.

Proof.
By structural induction over φ.
Let $\rho_1 = < s_0, a_1, s_1, a_2, s_2 ...>$ and $\rho_2 = < s_0, a_k, s_k, a_{k+1}, s_{k+1} ...>$. Let $\rho_1[s_j\rangle\rangle \models \varphi$ and $\rho_2[s_{k+j}\rangle\rangle \not\models \varphi$.

Assuming that the Lemma holds for ψ_1, ψ_2, φ_1 and φ_2.

- $\varphi = \psi$:
  - $\psi \in AP$:

    $\rho_1[s_j\rangle\rangle \models \varphi$
    $\Leftrightarrow s_j \models \psi$ ....(1)
    $\rho_2[s_{k+j}\rangle\rangle \not\models \varphi$
    $\Leftrightarrow s_{k+j} \not\models \psi$ ....(2)
    $\Rightarrow s_{k+j-1} \xrightarrow{\alpha} s_{k+j}$ and $obs_\varphi(\alpha)$, from (1) and (2),
    $\Rightarrow s_{j-1} \xrightarrow{\beta} s_j$ and $\alpha \neq \beta$, from (1),
    $\Rightarrow \exists\, a_i \in \rho_2\langle\langle s_{k+j}]$ such that $obs_\varphi(a_i)$ and $a_i \notin \rho_1\langle\langle s_j]$,
    $\Rightarrow diffPaths_\varphi(s_0)$, by criterion (ii) of Definition 43.

  - $\psi = \neg\, \psi_1$:

$\rho_1[s_j\rangle\!\rangle \models \varphi$
$\Leftrightarrow s_j \models \neg \psi_1$
$\Leftrightarrow s_j \not\models \psi_1$
$\Leftrightarrow \rho_1[s_j\rangle\!\rangle \not\models \psi_1$ ....(1)
$\rho_2[s_{k+j}\rangle\!\rangle \not\models \varphi$
$\Leftrightarrow s_{k+j} \not\models \neg \psi_1$
$\Leftrightarrow s_{k+j} \models \psi_1$
$\Leftrightarrow \rho_2[s_{k+j}\rangle\!\rangle \models \psi_1$ .....(2)
$\Rightarrow$ *diffPaths*$_\varphi(s_0)$, by (1) and (2) and the induction assumption.

- $\psi = \psi_1 \wedge \psi_2$
  $\rho_1[s_j\rangle\!\rangle \models \varphi$
  $\Leftrightarrow s_j \models \psi_1 \wedge \psi_2$
  $\Leftrightarrow s_j \models \psi_1$ and $s_j \models \psi_2$
  $\Leftrightarrow \rho_1[s_j\rangle\!\rangle \models \psi_1$ and $\rho_1[s_j\rangle\!\rangle \models \psi_2$ ...(1)
  $\rho_2[s_{k+j}\rangle\!\rangle \not\models \varphi$
  $\Leftrightarrow s_{k+j} \not\models \psi_1 \wedge \psi_2$
  $\Leftrightarrow s_j \not\models \psi_1$ or $s_j \not\models \psi_2$
  $\Leftrightarrow \rho_1[s_{k+j}\rangle\!\rangle \not\models \psi_1$ or $\rho_1[s_{k+j}\rangle\!\rangle \not\models \psi_2$ ...(2)
  $\Rightarrow$ *diffPaths*$_\varphi(s_0)$, by (1) and (2) and the induction assumption.
- $\psi = \mathsf{E}\varphi_2$:
  $\rho_1[s_j\rangle\!\rangle \models \varphi$
  $\Leftrightarrow s_j \models \mathsf{E}\varphi_2$
  $\Leftrightarrow \exists \rho_3 = < s_j... >$ such that $\rho_3 \models \varphi_2$. ....(1)
  $\rho_2[s_{k+j}\rangle\!\rangle \not\models \varphi$
  $\Leftrightarrow s_{k+j} \not\models \mathsf{E}\varphi_2$
  $\Leftrightarrow \forall \rho_4 = < s_{k+j}... >, \rho_4 \not\models \varphi_2$
  $\Rightarrow \exists \rho_4 = < s_{k+j}... >$ such that $\rho_4 \not\models \varphi$ ...(2).
  $\Rightarrow$ *diffPaths*$_\varphi(s_0)$, by (1) and (2) and the induction assumption.

- $\varphi = \neg \varphi_1$:
  $\rho_1[s_j\rangle\!\rangle \models \varphi$
  $\Leftrightarrow \rho_1[s_j\rangle\!\rangle \not\models \varphi_1$ .....(1)
  $\rho_2[s_{k+j}\rangle\!\rangle \not\models \varphi$
  $\Leftrightarrow \rho_2[s_{k+j}\rangle\!\rangle \models \varphi_1$ ....(2)
  $\Rightarrow$ *diffPaths*$_\varphi(s_0)$, by (1) and (2) and the induction assumption.

- $\varphi = \varphi_1 \wedge \varphi_2$:
  $\rho_1[s_j\rangle\!\rangle \models \varphi$
  $\Leftrightarrow \rho_1[s_j\rangle\!\rangle \models \varphi_1 \wedge \varphi_2$
  $\Leftrightarrow \rho_1[s_j\rangle\!\rangle \models \varphi_1$ and $\rho_1[s_j\rangle\!\rangle \models \varphi_2$ ....(1)

$\rho_2[s_{k+j}\rangle \not\models \varphi$

$\Leftrightarrow \rho_2[s_{k+j}\rangle \not\models \varphi_1 \wedge \varphi_2$

$\Leftrightarrow \rho_2[s_{k+j}\rangle \not\models \varphi_1$ and $\rho_2[s_{k+j}\rangle \not\models \varphi_2$ ....(2)

From (1) and (2), either:

$\rho_1[s_j\rangle \models \varphi_1$ and $\rho_2[s_{k+j}\rangle \not\models \varphi_1$ or

$\rho_1[s_j\rangle \models \varphi_2$ and $\rho_2[s_{k+j}\rangle \not\models \varphi_2$

$\Rightarrow$ $diffPaths_{\varphi}(s_0)$, by the induction assumption.

- $\varphi = X\varphi_1$

  $\rho_1[s_j\rangle \models \varphi$

  $\Leftrightarrow \rho_1[s_{j+1}\rangle \models \varphi_1$ ....(1)

  $\rho_2[s_{k+j}\rangle \not\models \varphi$

  $\Leftrightarrow \rho_2[s_{k+j+1}\rangle \not\models \varphi_1$ ....(2)

  From (1) and (2) and the induction assumption, either (i) or (ii) of Definition 43 holds for s_{j+1} in ρ_1 and s_{k+j+1} in ρ_2.

  Therefore, either (i) or (ii) of Definition 43 holds for s_j in ρ_1 and s_{k+j} in ρ_2

  $\Rightarrow$ $diffPaths_{\varphi}(s_0)$.

- $\varphi = \varphi_1 \cup \varphi_2$

  $\rho_1[s_j\rangle \models \varphi_1 \cup \varphi_2$

  $\Leftrightarrow \exists v$ such that $v \geq j$ and $\rho[s_v\rangle \models \varphi_2$ ....(1)

  and $\forall w$ such that $0 \leq w < v$, $\rho[s_w\rangle \models \varphi_1$. ....(2)

  $\rho_2[s_{k+j}\rangle \not\models \varphi_1 \cup \varphi_2$

  $\Leftrightarrow$ either $\exists x$ such that $0 \leq x < y$, for some y, where $\rho_2[s_{k+x}\rangle \not\models \varphi_1$ ...(3)

  or $\nexists y$ such that $y \geq j$ and $\rho_2[s_{k+y}\rangle \models \varphi_2$ .....(4)

  From (1) and (4), either criterion (i) or (ii) of Definition 43 holds for s_v in ρ_1 and s_{k+v} in ρ_2, or

  from (2) and (3), either criterion (i) or (ii) of Definition 43 holds for s_x in ρ_1 and s_{k+x} in ρ_2.

  $v \geq j$ and $x \geq j$,

  so criterion (i) of Definition 43 holds for s_j in ρ_1 and s_{k+j} in ρ_2

  $\Rightarrow$ $diffPaths_{\varphi}(s_0)$.

□

The following lemma, Lemma 9, uses the previous result. Lemma 9 is for the case of Theorem 4 in which the formula contains an X operator. Assume the property to be verified is φ, such that $\varphi = X\varphi'$.

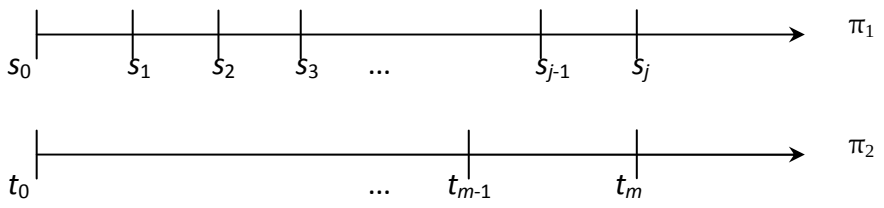


Figure 52. The two paths for Lemma 9.

Assume there are two next-preserving branching bisimilar paths, where a state s_j on one path is next-preserving branching bisimilar to a state t_m on the other path and all the steps before s_{j-1} are stuttering, as shown in Figure 52. Further assume that the suffix of the path starting at s_j does not satisfy the property φ', while the suffix starting at s_{j-1} does, or vice versa. Assume that on the second path, all the stuttering steps have been removed, such as by using the normal slicing method, leaving only t_m. This will cause Xφ' to evaluate to *false* at t_0, whereas it evaluated to *true* at s_0, due to the stuttering steps. This lemma shows that such cases are not possible if the paths are next-preserving bisimilar, since there must be a certain number of stuttering steps before t_m. This holds because at some state *s*' on the path, one of the sub-formulas φ_1 of φ must change its value. This occurs if either an observable step occurs at *s*' or if *s*' leads to another path on which φ_1 is satisfied. In the latter case, the result from Lemma 8 shows that $diffPaths_{\varphi}(s')$ must hold. Let *t*' represent the equivalent state on the second path. Since the paths are next-preserving bisimilar with respect to φ, there will be stuttering steps before all observable and *diffPaths* states. Therefore, there will be *k* stuttering steps before *t*', where *k* is the minimum of the *x-depth* of φ and the number of stuttering steps before *s*'. If the formula φ has an *x-depth* of *d* and the formula φ_1 has an *x-depth* of d_2, then the point at which φ_1 changes its value, i.e. *t*', occurs $d - d_2 + 1$ steps after t_{m-1}. Then, as the following lemma shows, it follows that if there are *d* steps before *t*', there must be $d - d_2$ steps before t_{m-1}. Using this result, Theorem 4 will show that there are enough stuttering steps before t_{m-1} to ensure that the formula Xφ' evaluates to the same value at both s_0 and t_0.

Example.
The following example illustrates how the *x-depth*(φ) - *x-depth*(φ_1) requirement ensures the correct number of stuttering steps are before t_m. Let φ = Xφ', where φ' = X*p*, for some atomic proposition, *p*. Assume that at s_{j-1}, φ' is satisfied, while it is not at s_j. Thus, s_{j-1} satisfies X*p*, but s_j does not, so at the state s_j, the sub-formula *p* holds while at s_{j+1} it does not. The *x-depth* of *p* is 0, while the *x-depth* of φ is 2. Thus, there will be 2 - 0 = 2 stuttering steps before s_j, which implies that there is 1 stuttering step before s_{j-1}, so there is one stuttering step before t_{m-1}. This satisfies the requirement that there is 1 stuttering step before s_{j-1}. (The *x-depth* of φ' is 1, while the *x-depth* of φ is 2, so there are required to be 2 - 1 = 1 stuttering steps before s_{j-1}). ■

LEMMA 9. INCLUSION OF STUTTERING NODES

For a path formula φ and two paths $\pi_1 = <s_0, s_1, s_2, ... >$ and $\pi_2 = <t_0, t_1, t_2, ... >$,
where $\pi_1 \overset{\star}{=}_{\varphi} \pi_2$ and $\exists s_j, t_m$ such that $t_m \overset{\star}{=}_{\varphi} s_j$ and $s_0 \dashrightarrow^{j-1} s_{j-1}$ and for some formula φ_1, which is a sub-formula of φ:

A) if $\pi_1[s_{j-1}\rangle\rangle \vDash \varphi_1$ and $\pi_2[s_j\rangle\rangle \nvDash \varphi_1$, where $j > 1$, then $t_0 \dashrightarrow^{k} t_{m-1}$,
where $k \geq min(x\text{-}depth(\varphi) - x\text{-}depth(\varphi_1), j\text{-}1)$ and

B) if $\pi_1[s_{j-1}\rangle\rangle \nvDash \varphi_1$ and $\pi_2[s_j\rangle\rangle \vDash \varphi_1$, where $j > 1$, then $t_0 \dashrightarrow^{k} t_{m-1}$,
where $k \geq min(x\text{-}depth(\varphi) - x\text{-}depth(\varphi_1), j\text{-}1)$

Proof. By structural induction over the formula φ_l.
Induction assumption: Assume that (A) and (B) hold for the formulas ψ_2 and ψ_3.

- $\varphi_1 = \psi_1$, for some state formula ψ_1.
  Then, $\pi_1[s_{j-1}\rangle\rangle \vDash \varphi_l \Leftrightarrow s_{j-1} \vDash \psi_1$ and $\pi_1[s_j\rangle\rangle \vDash \varphi_l \Leftrightarrow s_j \vDash \psi_1$.
  - $\psi_1 \in$ AP:
    For Case (A):
    Assume $s_{j-1} \vDash \psi_1$ but $s_j \nvDash \psi_1$.
    $\Rightarrow s_{j-1} \xrightarrow{\alpha} s_j$, where $obs_{\varphi}(\alpha)$ .....(1)

$s_j \overset{\star}{=}_{\varphi} t_m$.

$\Rightarrow t_0 \overset{k}{\dashrightarrow} t_{m-1}$, where $k \geq min(x\text{-}depth(\varphi), j\text{-}1)$, by criterion (ii) of Definition 45 and (1).

$x\text{-}depth(\varphi_1) = 0$, since $\varphi_1 \in AP$.

$\Rightarrow x\text{-}depth(\varphi) - x\text{-}depth(\varphi_1) = x\text{-}depth(\varphi)$

$\Rightarrow k \geq min(x\text{-}depth(\varphi) - x\text{-}depth(\varphi_1), j\text{-}1)$

For Case (B): Similar reasoning as for Case (A).

- $\psi_1 = \psi_2 \wedge \psi_3$, for some state formulas ψ_2 and ψ_3

  For Case (A):

  Assume $s_{j-1} \vDash \psi_1$ but $s_j \nvDash \psi_1$.

  $s_{j-1} \vDash \psi_1$

  $\Rightarrow s_{j-1} \vDash \psi_2$ and $s_{j-1} \vDash \psi_3$ .....(1)

  $s_j \nvDash \psi_1$

  $\Rightarrow$ either $s_j \nvDash \psi_2$ or $s_j \nvDash \psi_3$ .....(2)

  $\Rightarrow$ either $s_{j-1} \vDash \psi_2$ and $s_j \nvDash \psi_2$ or

  $s_{j-1} \vDash \psi_3$ and $s_j \nvDash \psi_3$, from (1) and (2).

  $\Rightarrow t_0 \overset{k}{\dashrightarrow} t_{m-1}$, where $k \geq min(x\text{-}depth(\varphi) - x\text{-}depth(\varphi_1), j\text{-}1)$, by assumption.

  For Case (B): Similar reasoning as for Case (A).

- $\psi_1 = \neg \psi_2$, for some state formula ψ_2

  For Case (A):

  Assume $s_{j-1} \vDash \psi_1$ but $s_j \nvDash \psi_1$

  $s_{j-1} \vDash \psi_1$

  $\Rightarrow s_{j-1} \nvDash \psi_2$. .....(1)

  $s_j \nvDash \psi_1$

  $\Rightarrow s_j \vDash \psi_2$ .....(2)

  $\Rightarrow t_0 \overset{k}{\dashrightarrow} t_{m-1}$, where $k \geq min(x\text{-}depth(\varphi) - x\text{-}depth(\varphi_1), j\text{-}1)$, by assumption, (1) and (2).

  For Case (B): Similar reasoning as for Case (A).

- $\psi_1 = E\varphi_2$, for some path formula φ_2

  For Case (A):

  Assume $s_{j-1} \vDash \psi_1$ but $s_j \nvDash \psi_1$

  $s_{j-1} \vDash \psi_1$

  $\Rightarrow \exists \pi_3 \in paths(s_{j-1})$ such that $\pi_3 \vDash \varphi_2$

  but $\forall \pi_4 \in paths(s_j)$, $\pi_4 \nvDash \varphi_2$

  Let $\rho_1 = < s_{j-1}, a_r, s_r, a_{r+1}, s_{r+1} \ldots > = run(\pi_3)$

  and $\rho_2 = < s_{j-1}, a_j, s_j, a_{j+1}, s_{j+1} \ldots >$

  $\rho_1[s_{j-1}\rangle\!\rangle \vDash \varphi_2$ but $\rho_2[s_j\rangle\!\rangle \nvDash \varphi_2$

By Lemma 8 either:

(a) $\exists\, a_i \in \rho_1$ such that $obs_\varphi(a_i)$ and $a_i \notin \rho_2$ or vice versa, or

(b) $\exists\, m \geq 0$ and $\exists\, a_i \in \rho_1\langle\!\langle s_{r+m}]$ such that $obs_\varphi(a_i)$ and $a_i \notin \rho_2\langle\!\langle s_{j+m+1}]$ or vice versa.

$s_j \stackrel{\star}{=}_\varphi t_m$

$\Rightarrow t_0 \xrightarrow{\quad}{}^{k} t_{m-1}$, where $k \geq min(x\text{-}depth(\varphi),\ j\text{-}1)$,
by criterion (iii) of Definition 45 and (a) and (b) above.

$x\text{-}depth(\varphi) \geq x\text{-}depth(\varphi) - x\text{-}depth(\varphi_1)$

$\Rightarrow k \geq min(x\text{-}depth(\varphi) - x\text{-}depth(\varphi_1),\ j\text{-}1)$

For Case (B):

Assume $s_{j-1} \not\models \psi_1$ but $s_j \models \psi_1$

$s_j \models \psi_1$

$\Rightarrow \exists\, \pi_3 \in paths(s_j)$ such that $\pi_3 \models \varphi_2$, ...(1)

$s_{j-1} \not\models \psi_1$

$\Rightarrow \forall\, \pi_4 \in paths(s_{j-1}), \pi_4 \models \varphi_2$. .....(2)

$\exists\, \pi_5 = \langle s_{j-1} \rangle \frown \pi_3$, from (1)

Let $\rho_1 = run(\pi_5)$ and $\rho_2 = run(\pi_3)$

Case 1: $\rho_1 \models \varphi_2$

$\Rightarrow \pi_5 \in paths(s_{j-1})$ and π_5,
which contradicts (2), so criterion (B) holds vacuously.

Case 2: $\rho_1[s_{j-1}\rangle\!\rangle \not\models \varphi_2$

$\Rightarrow \rho_1[s_{j-1}\rangle\!\rangle \not\models \varphi_2$ but $\rho_2[s_j\rangle\!\rangle \models \varphi_2$
Using similar reasoning as for Case (A) above,
$\Rightarrow k \geq min(x\text{-}depth(\varphi) - x\text{-}depth(\varphi_1),\ j\text{-}1)$

Assume that (A) and (B) hold for the formulas φ_2 and φ_3.

- $\varphi_1 = \varphi_2 \wedge \varphi_3$, for some path formulas φ_2 and φ_3.

For Case (A):

Assume $\pi_1[s_{j-1}\rangle\!\rangle \models \varphi_1$ but $\pi_1[s_j\rangle\!\rangle \not\models \varphi_1$

$\pi_1[s_{j-1}\rangle\!\rangle \models \varphi_1$

$\Rightarrow \pi_1[s_{j-1}\rangle\!\rangle \models \varphi_2$ and $\pi_1[s_{j-1}\rangle\!\rangle \models \varphi_3$ .....(1)

$\pi_1[s_j\rangle\!\rangle \not\models \varphi_1$

$\Rightarrow$ either $\pi_1[s_j\rangle\!\rangle \not\models \varphi_2$ or $\pi_1[s_j\rangle\!\rangle \not\models \varphi_3$.....(2)

$\Rightarrow$ either $\pi_1[s_{j-1}\rangle\!\rangle \models \varphi_2$ and $\pi_1[s_j\rangle\!\rangle \not\models \varphi_2$ or
$\pi_1[s_{j-1}\rangle\!\rangle \models \varphi_3$ and $\pi_1[s_j\rangle\!\rangle \not\models \varphi_3$, from (1) and (2).

$\Rightarrow t_0 \xrightarrow{\quad}{}^{k} t_{m-1}$, where $k \geq min(x\text{-}depth(\varphi) - d,\ j\text{-}1)$, ...(3)
where $d = x\text{-}depth(\varphi_2)$ or $x\text{-}depth(\varphi_3)$, by assumption

$x\text{-}depth(\varphi_1) = max(x\text{-}depth(\varphi_2), x\text{-}depth(\varphi_3))$, by Definition 44

$\Rightarrow t_0 \xrightarrow{\quad}{}^{k} t_{m-1}$, where $k \geq min(x\text{-}depth(\varphi) - x\text{-}depth(\varphi_1),\ j\text{-}1)$, by (3).

For Case (B): Similar reasoning as for Case (A).

- $\varphi_1 = \neg \varphi_2$, for some path formula φ_2

  For Case (A):

  Assume $\pi_1[s_{j-1}\rangle \models \varphi$ but $\pi_1[s_j\rangle \not\models \varphi_1$

  $\pi_1[s_{j-1}\rangle \models \varphi_1$

  $\Rightarrow \pi_1[s_{j-1}\rangle \not\models \varphi_2$. .....(1)

  $\pi_1[s_j\rangle \not\models \neg \varphi_2$

  $\Rightarrow \pi_1[s_j\rangle \models \varphi_2$ .....(2)

  $\Rightarrow t_0 \dashrightarrow^k t_{m-1}$, where $k \geq min(x\text{-}depth(\varphi) - x\text{-}depth(\varphi_1), j\text{-}1)$, by assumption, (1) and (2).

  For Case (B): Similar reasoning as for Case (A).

- $\varphi_1 = \varphi_2 \mathbin{U} \varphi_3$, for some path formulas φ_2 and φ_3

  For Case (A):

  Assume $\pi_1[s_{j-1}\rangle \models \varphi_2 \mathbin{U} \varphi_3$ but $\pi_1[s_j\rangle \not\models \varphi_2 \mathbin{U} \varphi_3$.

  $\pi_1[s_{j-1}\rangle \models \varphi_2 \mathbin{U} \varphi_3$

  $\Rightarrow \exists k$ such that $\forall s_i \in \pi_1$, where $j - 1 \leq i < k$, $\pi_1[s_i\rangle \models \varphi_2$, and $\pi_1[s_k\rangle \models \varphi_3$, by the definition of $\models$

  $\pi_1[s_j\rangle \not\models \varphi_2 \mathbin{U} \varphi_3$

  $\Rightarrow \pi_1[s_j\rangle \not\models \varphi_2$ and $\pi_1[s_j\rangle \not\models \varphi_3$

  $\Rightarrow \pi_1[s_{j-1}\rangle \models \varphi_3$ and $\pi_1[s_j\rangle \not\models \varphi_3$, because otherwise $\pi_1[s_j\rangle$ would still satisfy φ_1.

  $\Rightarrow t_0 \dashrightarrow^k t_{m-1}$, where $k \geq min(x\text{-}depth(\varphi) - x\text{-}depth(\varphi_1), j\text{-}1)$, by assumption.

  For Case (B):

  Assume $\pi_1[s_{j-1}\rangle \not\models \varphi_2 \mathbin{U} \varphi_3$ but $\pi_1[s_j\rangle \models \varphi_2 \mathbin{U} \varphi_3$.

  $\pi_1[s_j\rangle \models \varphi_2 \mathbin{U} \varphi_3$

  $\Rightarrow \exists k > j$ such that $\forall s_i \in \pi_1$, where $j \leq i < k$, $\pi_1[s_i\rangle \models \varphi_2$, and $\pi_1[s_k\rangle \models \varphi_3$, by the definition of $\models$ .....(1)

  $\Rightarrow$ either $\pi_1[s_j\rangle \models \varphi_3$ or $\pi_1[s_j\rangle \models \varphi_2$, from (1) .....(2)

  $\pi_1[s_{j-1}\rangle \not\models \varphi_2 \mathbin{U} \varphi_3$

  $\Rightarrow$ either $\nexists s_k$ such that $k \geq j - 1$, where $\pi_1[s_k\rangle \models \varphi_3$ or

  $\exists k$ such that $k \geq j - 1$, where $\pi_1[s_k\rangle \models \varphi_3$ and $\exists i$ such that $j - 1 \leq i < k$ where $\pi_1[s_i\rangle \not\models \varphi_2$, by the definition of $\models$

  $\Rightarrow \pi_1[s_{j-1}\rangle \not\models \varphi_2$ and $\pi_1[s_{j-1}\rangle \not\models \varphi_3$, from (1)

  $\Rightarrow$ either $\pi_1[s_{j-1}\rangle \not\models \varphi_2$ and $\pi_1[s_j\rangle \models \varphi_2$, or

  $\pi_1[s_{j-1}\rangle \not\models \varphi_3$ and $\pi_1[s_j\rangle \models \varphi_3$, from (2).

  $x\text{-}depth(\varphi_1) = max(x\text{-}depth(\varphi_2), x\text{-}depth(\varphi_3))$, by Definition 44

  $\Rightarrow t_0 \dashrightarrow^k t_{m-1}$, where $k \geq min(x\text{-}depth(\varphi) - x\text{-}depth(\varphi_1), j\text{-}1)$, by assumption.

- $\varphi_1 = \mathsf{X}\varphi_2$, for some path formula φ_2.

  For Case (A):

  Assume $\pi_1[s_{j-1}\rangle \models \varphi_1$ but $\pi_1[s_j\rangle \not\models \varphi_1$

$\pi_1[s_{j-1}\rangle \models \mathsf{X}\varphi_2$

$\Rightarrow \pi_1[s_j\rangle \models \varphi_2$ .....(1)

$\pi_1[s_j\rangle \not\models \mathsf{X}\varphi_2$

$\Rightarrow \pi_1[s_{j+1}\rangle \not\models \varphi_2$ .....(2)

Let $s_j \xrightarrow{a_j} s_{j+1}$

Case 1: $obs_\varphi(a_j)$

$\Rightarrow t_0 \dashrightarrow^k t_m$, where $k \geq min(x\text{-}depth(\varphi), j)$, by criterion (ii) of Definition 45, since $s_j \overset{\star}{=}_\varphi t_m$.

$x\text{-}depth(\varphi) = x\text{-}depth(\varphi_1) + 1$ .....(3)

$\Rightarrow t_0 \dashrightarrow^k t_{m-1}$, where $k \geq min(x\text{-}depth(\varphi_1), j\text{-}1)$ .....(4)

$x\text{-}depth(\varphi) - x\text{-}depth(\varphi_1) = x\text{-}depth(\varphi_1) + 1 - x\text{-}depth(\varphi_1) = 1$, from (3)

$x\text{-}depth(\varphi_1) \geq 1$

$\Rightarrow x\text{-}depth(\varphi_1) \geq x\text{-}depth(\varphi) - x\text{-}depth(\varphi_1)$

$\Rightarrow k \geq min(x\text{-}depth(\varphi) - x\text{-}depth(\varphi_1),\ j\text{-}1)$, from (4).

Case 2: $\neg\, obs_\varphi(a_j)$

$\Rightarrow s_0 \dashrightarrow^j s_j$ .....(3)

$\pi_1 \overset{\star}{=}_\varphi \pi_2$ and $\varphi = \mathsf{X}\varphi_2$

$\Rightarrow t_0 \dashrightarrow^k t_m$, where $k \geq min(x\text{-}depth(\varphi) - x\text{-}depth(\varphi_2), j)$, by assumption that Lemma 9 holds for φ_2, with (1), (2) and (3).

$x\text{-}depth(\varphi_1) = x\text{-}depth(\varphi_2) + 1$

$\Rightarrow x\text{-}depth(\varphi) - x\text{-}depth(\varphi_2) = x\text{-}depth(\varphi) - x\text{-}depth(\varphi_1) + 1$

$\Rightarrow t_0 \dashrightarrow^k t_{m-1}$, where $k \geq min(x\text{-}depth(\varphi) - x\text{-}depth(\varphi_1), j\text{-}1)$

For Case (B):

Assume $\pi_1[s_{j-1}\rangle \not\models \varphi_1$ but $\pi_1[s_j\rangle \models \varphi_1$

$\pi_1[s_{j-1}\rangle \not\models \mathsf{X}\varphi_2$

$\Rightarrow \pi_1[s_j\rangle \not\models \varphi_2$ .....(1)

$\pi_1[s_j\rangle \models \mathsf{X}\varphi_2$

$\Rightarrow \pi_1[s_{j+1}\rangle \models \varphi_2$ .....(2)

The remainder of this case is the same as for Case (A).

□

Theorem 4 is the main result of this section, which demonstrates that two transition systems which are next-preserving branching bisimilar preserve the same CTL* formulas. The proof is divided into two sections: a proof that if two states are next-preserving branching bisimilar, they preserve CTL* state formulas and the same for two next-preserving branching bisimilar paths and the preservation of CTL* path formulas. Recall that one of the CTL* state formulas is $\mathsf{E}\varphi$, where φ is a path formula. This is where the result of Lemma 7 is used, to show that there will always exist a matching path in the other system. The second section, concerning the proof of the preservation of path formulas, is where the X operator is examined, since it is a path operator. This section utilises the results of Lemma 9, as explained previously.

THEOREM 4. NEXT-PRESERVING BRANCHING BISIMULATION PRESERVES FULL CTL*
For two doubly-labelled transition systems T_1 and T_2, $T_1 \overset{\star}{=}_\psi T_2 \Rightarrow (T_1 \vDash \psi \Leftrightarrow T_2 \vDash \psi$, for all $\psi \in$ CTL*).

Proof.
Assume $T_1 \overset{\star}{=}_\psi T_2$.

For this it is required to show that:
(i) For all state formulas $\psi \in$ CTL* and states $s_i \in T_1$ and $t_i \in T_2$, $s_i \overset{\star}{=}_\psi t_i \Rightarrow (s_i \vDash \psi \Leftrightarrow t_i \vDash \psi)$ and
(ii) for all path formulas $\varphi \in$ CTL* and paths π_1 and π_2, $\pi_1 \overset{\star}{=}_\varphi \pi_2 \Rightarrow (\pi_1 \vDash \varphi \Leftrightarrow \pi_2 \vDash \varphi)$.

(By induction over the formula).

Assume that state formulas $\psi_1 \in$ CTL* and $\psi_2 \in$ CTL* are preserved by $\overset{\star}{=}_{\psi_1}$ and $\overset{\star}{=}_{\psi_2}$, respectively, i.e. $T_1 \vDash \psi_1 \Leftrightarrow T_2 \vDash \psi_1$ and $T_1 \vDash \psi_2 \Leftrightarrow T_2 \vDash \psi_2$.

State formulas:
Since $\overset{\star}{=}_\psi \Rightarrow \triangleq$, all $\psi \in$ CTL$^*_{-X}$ are preserved, by Theorem 1.

For a formula ψ which contains X, if:

- $\psi \in$ AP: $\psi \in$ CTL$^*_{-X}$
  Therefore ψ is preserved.

- $\psi = \psi_1 \wedge \psi_2$:
  $T_1 \vDash \psi$
  $\Leftrightarrow s_0 \vDash \psi$
  $\Leftrightarrow s_0 \vDash \psi_1$ and $s_0 \vDash \psi_2$.
  $\Leftrightarrow t_0 \vDash \psi_1$ and $t_0 \vDash \psi_2$, by assumption.
  $\Leftrightarrow t_0 \vDash \psi$
  $\Leftrightarrow T_2 \vDash \psi$

- $\psi = \neg \psi_1$:
  $T_1 \vDash \psi$
  $\Leftrightarrow s_0 \vDash \psi$
  $\Leftrightarrow s_0 \nvDash \psi_1$
  $\Leftrightarrow t_0 \nvDash \psi_1$, by assumption.
  $\Leftrightarrow t_0 \vDash \psi$
  $\Leftrightarrow T_2 \vDash \psi$

- $\psi = \mathsf{E}\varphi_1$, for some path formula φ_1.
  $T_1 \vDash \varphi_1$
  $\Leftrightarrow \exists \pi_1 \in paths(s_0)$ such that $\pi_1 \vDash \varphi_1$
  $\Leftrightarrow \exists \pi_2 \in paths(t_0)$ such that $\pi_2 \vDash \varphi_1$, by Lemma 7.

$\Leftrightarrow T_2 \vDash \mathsf{E}\varphi_1$

$\Leftrightarrow T_2 \vDash \psi$

For path formulas:

Assume statement (i) holds for state formula ψ_1 and statement (ii) holds for path formulas φ_1, φ_2, and φ_3. Let $\pi_1 \overset{\star}{=}_{\varphi_1} \pi_2$, $\pi_1 \overset{\star}{=}_{\varphi_2} \pi_2$ and $\pi_1 \overset{\star}{=}_{\varphi_3} \pi_2$. Let $first(\pi_1) = s_0$ and $first(\pi_2) = t_0$. Assume $\pi_1 \vDash \varphi$ and $\pi_2 \nvDash \varphi$. The reverse case holds using the same reasoning.

- $\varphi = \psi_1$, for some state formula ψ_1:

  $\pi_1 \vDash \varphi$

  $\Leftrightarrow s_0 \vDash \psi_1$

  $\Leftrightarrow t_0 \vDash \psi_1$, since $s_0 \overset{\star}{=}_{\psi_1} t_0$, by assumption.

  $\Leftrightarrow t_0 \vDash \psi_1$

  $\Leftrightarrow \pi_2 \vDash \varphi$

- $\varphi = \varphi_1 \wedge \varphi_2$:

  $\pi_1 \vDash \varphi_1 \wedge \varphi_2$

  $\Leftrightarrow \pi_1 \vDash \varphi_1$ and $\pi_1 \vDash \varphi_2$

  $\Leftrightarrow \pi_2 \vDash \varphi_1$ and $\pi_2 \vDash \varphi_2$, by assumption

  $\Leftrightarrow \pi_2 \vDash \varphi_1 \wedge \varphi_2$

- $\varphi = \neg \varphi_1$

  $\pi_1 \vDash \neg \varphi_1$

  $\Leftrightarrow \pi_1 \nvDash \varphi_1$

  $\Leftrightarrow \pi_2 \nvDash \varphi_1$, by assumption

  $\Leftrightarrow \pi_2 \vDash \neg \varphi_1$

- $\varphi = \varphi_1 \mathsf{U} \varphi_2$

  $\pi_1 \vDash \varphi_1 \mathsf{U} \varphi_2$

  $\Leftrightarrow \exists k$ such that $\forall s_i \in \pi_1$, where $0 \leq i \leq k$, $\pi_1[s_i\rangle\!\rangle \vDash \varphi_1$ and $\pi_1[s_k\rangle\!\rangle \vDash \varphi_2$

  $\forall s_j \in \pi_1, \exists t_j \in \pi_2$ such that $s_j \overset{\star}{=}_{\varphi_2} t_j$

  $\Rightarrow \exists t_k \in \pi_2$ such that $s_k \overset{\star}{=}_{\varphi_2} t_k$

  $\pi_1[s_k\rangle\!\rangle \vDash \varphi_2$ iff $\pi_2[t_k\rangle\!\rangle \vDash \varphi_2$, by assumption. .....(1)

  $\forall s_i, \exists t_i \in \varphi_2$ such that $s_i \overset{\star}{=}_{\varphi_1} t_i$

  $\Rightarrow \forall s_i, t_i, \pi_1[s_i\rangle\!\rangle \vDash \varphi_1 \Leftrightarrow \pi_2[t_i\rangle\!\rangle \vDash \varphi_1$, by assumption. .....(2)

  $\Rightarrow \varphi_2 \vDash \varphi_1 \mathsf{U} \varphi_2$, by (1) and (2).

- $\varphi = \mathsf{X}\varphi_1$

  Proof by contradiction.

  $\pi_1 \vDash \mathsf{X}\varphi_1$ and $\pi_2 \nvDash \mathsf{X}\varphi_1$

  $\Rightarrow \pi_1[s_1\rangle\!\rangle \vDash \varphi_1$ and $\pi_2[t_1\rangle\!\rangle \nvDash \varphi_1$ .....(1)

Let t_1 be t_m and t_0 be t_{m-1}.

$\pi_1 \overset{\star}{=}_{\varphi_1} \pi_2$, by assumption.

$\Rightarrow \exists j > 1$ such that $s_j \overset{\star}{=}_{\varphi_1} t_m$, by Definition 47. .....(2)

$\Rightarrow \pi_1[s_j\rangle\!\rangle \not\models \varphi_1$, by assumption. .....(3)

$s_0 \overset{\star}{=}_{\varphi_1} t_0$, by assumption .....(4)

Let $t_{m-1} \xrightarrow{b_m} t_m$ and $s_{j-1} \xrightarrow{a_j} s_j$

$\Rightarrow s_0 \dashrightarrow^{j-1} s_{j-1}$ and $a_j = b_m$ and $s_{j-1} \overset{\star}{=}_{\varphi_1} t_{m-1}$, by (2), (4) and criterion (i) of Definition 45 .....(5)

$\Rightarrow s_0 \overset{\star}{=}_{\varphi_1} s_{j-1}$, by transitivity of $\overset{\star}{=}$.

$\Rightarrow \pi_1[s_{j-1}\rangle\!\rangle \models \varphi_1$, by assumption.

$\Rightarrow t_0 \dashrightarrow^{k} t_{m-1}$, where $k = min(x\text{-}depth(\varphi) - x\text{-}depth(\varphi_1), j\text{-}1)$, by (1), (2), (5) and Lemma 9.

$x\text{-}depth(\varphi) - x\text{-}depth(\varphi_1) = 1$

$\Rightarrow k = min(1, j\text{-}1)$

Therefore there is at least one step from t_0 to t_{m-1}, so t_m is not the first step.

$\Rightarrow \pi_2 \models \mathsf{X}\varphi_1$

which contradicts the assumption.

□

5.4 Next-Preserving Branching Bisimulation of Behavior Trees

In the previous section, a technique was proposed for producing reduced models that preserve properties containing the *next* operator, by inserting extra stuttering nodes at certain locations in the transition system. This section discusses how to identify such locations in a Behavior Tree model and how to locate suitable extra stuttering nodes.

5.4.1 Locations Requiring Extra Nodes

For a given BT control flow graph, the first step is to identify the states which are followed by an observable transition (criterion (i) of Definition 45) and those which are followed by two or more transitions (criterion (ii) of Definition 45). Locating observable transitions is simple: every observable transition in the transition system corresponds to an observable node in the tree. Therefore, to satisfy criterion (i), for every observable node there must be $x\text{-}depth(\varphi)$ extra stuttering nodes in the slice, where φ is the property to be verified.

Finding states that satisfy criterion (ii) is not so straight-forward. It might appear that the branching locations would correspond directly to branching points in the Behavior Tree, but this is not always the case. There are some branching locations in the transition system which do not correspond to an explicit branching point in the tree. An example of this is a selection node. It implicitly represents two branches: the branch where the selection condition holds and the branch where it does not, leading to a termination state. As well as this, branching nodes in the Behavior Tree do not always correspond to locations where an extra stuttering node must be added, due to the requirement in criterion (ii) that the state must satisfy *diffPaths*. For example, consider concurrent branches. Even when one branch is chosen, it does not prevent the other branch from executing as well, since they are parallel threads. Any observable nodes in one branch are therefore reachable on *all* corresponding paths in the transition system.

The Behavior Tree constructs which lead to branching in the transition system are: alternative branching, concurrent branching, thread kills, reversions, reference nodes and conditional nodes. Each of these will be discussed below.

Concurrent branching: In BT control flow graphs, concurrent branching occurs when a node *n* has more than one child, each connected to *n* by a concurrent edge. Obviously, the state immediately after *n* has executed has multiple branches in the transition system. However, this is not the only set of branches in the transition system caused by the concurrent nodes. Every node in each thread can be interleaved with the nodes in the other threads. After each node has executed, there are several possible choices over which node can execute next. Therefore, each state that results from one of the nodes executing will have multiple branches.

Despite this, none of these states satisfy criterion (ii) of Definition 45, as they do not satisfy the *diffPaths* requirement. Concurrent branches never cause the other branches to terminate, unless there is an explicit thread kill node or if one of the branches ends with a reversion which terminates the other threads. These cases can be identified by locating thread kill and reversion nodes, as described below. The definition of *diffPaths* requires that there is an observable node on one path that either cannot execute on the other path at all or cannot execute within the same number of steps. However, concurrent branching allows full interleaving semantics. Therefore, the situation required by *diffPaths* never occurs. If an observable node can be reached by executing one of the concurrent branches in the Behavior Tree, it is always possible to find a path from the other branch that will also lead to the same observable node executing, within the required time. Therefore there is no need to add stuttering nodes before concurrent branching points.

Alternative branching: This is different to concurrent branching, because once the choice of which branch to take has been made, the other branches will be terminated. There is no interleaving behaviour between the nodes in the branches. Let *n* be a node with more than one child, such that the children form an alternative branching group. The only state which has multiple outgoing transitions due to this branching group is the state immediately after *n* has executed, where the choice of which branch to take has not yet been made. Once a branch has been chosen, the other branches will terminate. This satisfies the requirement that there is an observable node in one path of the transition system that is unreachable via the other path.Therefore, alternative branches correspond to locations where stuttering nodes are necessary. There are cases where the other paths are still reachable, such as if the branches have reversions which cause the alternative branching point to be reached again. Since this is not always the case, it is safer to always add stuttering nodes to alternative branching points. If the other branches are reachable, the unnecessary stuttering nodes would not significantly increase the size of the slice.

Thread kill nodes: When a thread kill node executes, it terminates the thread in question. Thus, in the corresponding transition system, the state after the thread kill node executes leads to a path from which the terminated thread is unreachable. At the previous step, the thread is still reachable, because there are at least two possible branches: the thread kill step or a node in a concurrent branch. Consequently, stuttering nodes must be added before thread kill nodes.

Reversion nodes: Reversion nodes seem to be likely candidates for this type of behaviour, because after the reversion executes, some threads are terminated. However, the terminated threads are still reachable, since after the reversion is taken, at some point the threads will be re-started. Therefore there is no need to include extra stuttering nodes before reversions.

Reference nodes: Reference nodes are different to reversions, primarily because they do not cause any threads to be terminated. As a result any threads which were executing in parallel to the reference node will continue to execute. Nodes in alternative branches to the reference node would have already been terminated when the alternative branching choice was made prior to reaching the reference node. Hence it is not necessary to include extra stuttering nodes before reference nodes.

Conditional nodes: As discussed in Section 3.1, conditional nodes correspond to two possible transitions in the transition system: the branch where the condition holds and the branch where it does not. Consequently, at the step before the condition is evaluated, the two paths are still possible, but after one is chosen, the other path may be no longer possible. Since the verification property can never be referring to a value on the implicit *false* branch of the condition, the only relevant case is where the false branch has been chosen and the *true* branch is no longer reachable. For guard and synchronisation nodes, even when the false branch has been chosen, the true branch can still eventually be reached, since the false branch is a loop back to the conditional node. Thus, the only conditional nodes which can prevent a path from being reached are selection nodes, where the false branch leads to the END state. This is the only case which requires additional stuttering nodes.

The following lemma formalises the previous discussion, showing that the only cases which produce branches where one path leads to an observable step that is unreachable from the other path are alternative branches, thread kill and selection nodes. In the lemma, the term *reachable* is used to denote that a node can be reached on the other path within the required number of steps.

LEMMA 10. BRANCHES IN BEHAVIOR TREES

Let $S = (\mathcal{S}_1, AP_1, \mathcal{I}_1, \mathcal{L}_1, \mathcal{N}_1, \longrightarrow_1)$ be a doubly-labelled transition system. Let *branching* = $\{n \mid \exists\, s, s', s'', s''' \in \mathcal{S}_1$ where $s \xrightarrow{n} s' \longrightarrow s''$ and $s' \longrightarrow s'''$ and $s'' \neq s'''$ and $\exists$ a node $m \in \mathcal{N}_1$ such that m is reachable from s'' but not from s''' and $obs_\varphi(m)\}$.

Then, $\forall\, n \in$ *branching*, either:

(a) $\exists\, n_1, n_2$ such that $parent(n_1) = parent(n_2) = n$ and $alt(n_1, n_2)$, or
(b) $flag(n) = threadKill$ or
(c) $type(n) = selection$.

Proof.

Let n be a node such that $\exists\, s, s', s'', s''' \in \mathcal{S}_1$ where $s \xrightarrow{n} s' \longrightarrow s''$ and $s' \longrightarrow s'''$ and $s'' \neq s'''$.
There are several cases:

Case 1:
$parent(n_y) = parent(n_z) = n_x$ and $alt(n_y, n_z)$,
which satisfies (a).

Case 2:
$conc(n_y, n_z)$ and $flag(n_y) \neq threadKill$ and $flag(n_z) \neq threadKill$
$\Rightarrow m$ is reachable from n_y and n_z,
$\Rightarrow m$ is reachable from s'' and s'''.

Case 3:
$conc(n_y, n_z)$ and $flag(n_y) = threadKill$
which satisfies (b).

Case 4:
$conc(n_y, n_z)$ and $flag(n_y) = reversion$
Let n_a be a node such that $\exists\, v \in threads(n_a)$ which is terminated by n_y,
$\Rightarrow v \in threads(target(n_y))$
$\Rightarrow n_a \in desc(target(n_y))$
so n_a can be reached again.
$\Rightarrow m$ is reachable from n_y and n_z,
$\Rightarrow m$ is reachable from s'' and s'''.

Case 5:
$conditional(n)$

Case 5.1: *type*(*n*) ≠ *selection*
Let *e* = *edge*(*n*, *child*(*n*,0)), where *label*(*e*) = *false*.
⇒ *e* = *edge*(*n*, *n*), i.e. the edge loops back to *n*.
⇒ *m* is reachable from n_y and n_z,
⇒ *m* is reachable from *s*'' and *s*'''.
Case 5.2: *type*(*n*) = *selection*
which satisfies (c).

□

A set $extra_{\varphi}(G_1, G_2)$ contains the stuttering nodes that must be added for a given BT control flow graph G_1 and its slice G_2. For every observable node, alternative branching group, thread kill and selection node, there must be *x-depth*(φ) stuttering nodes in $extra_{\varphi}(G_1, G_2)$, or at least as many stuttering nodes as existed in the original model.

A function *nodes_next* is introduced in the following definition. It returns the set of nodes that encompass the final next-preserving slice. This includes the nodes from the normal slice as well as the nodes in the *extra* set.

DEFINITION 48. NEXT PRESERVING SLICE SET
For a slice $G_2 = <N_2,E_2, start_2, end_2>$ produced from a BT control flow graph $G_1 = <N_1,E_1, start_1, end_1>$ for a formula φ, the slice set that preserves the *next* operator, $nodes\_next_{\varphi}(G_1)$, is defined as:
$nodes\_next_{\varphi}(G_1) = N_2 \cup extra_{\varphi}(G_1, G_2)$.

■

5.4.2 Finding Extra Stuttering Nodes

The next step is to locate a set of suitable nodes to be included back into the slice. Let *n* be the node for which extra stuttering nodes must be included before it. The nodes should all be stuttering and should be able to execute immediately before *n*. If there are stuttering nodes in the slice that *always* execute before *n*, i.e. on every trace, then these nodes should be selected instead of nodes that only execute before *n* on *some* traces.

If *n* has a parent which is stuttering and is not already in the slice, the parent is a suitable choice. The parent is guaranteed to be able to execute before *n*. On the other hand, nodes in concurrent branches may execute before *n* but are not guaranteed to always do so. If a stuttering parent existed in the original model, it always had to execute before *n*. If a concurrent node is included into the slice instead of the parent, then the slice would have traces where no stuttering node executes before *n*, a trace which was impossible in the original model. The parent should therefore be the first choice of which stuttering node to include. For some formulas, more than one stuttering node may be necessary. In these cases, if there are more ancestors which are stuttering, these should be included next.

If the parent is not stuttering, or there are not enough stuttering ancestors to form the set of nodes to be included, the next choice is to include concurrent nodes. In most cases, the nodes in parallel threads can be interleaved in any order, so a concurrent node may execute before *n* or after it. There are traces where *n* executes first, unlike for the case of ancestor nodes. Nevertheless, assuming that all stuttering ancestors have been already included, this is not a problem, since the original model must have also had such traces. The goal of including the stuttering steps in this case is to preserve at least one of the traces in which the concurrent node executes immediately before *n*. The only difficulty lies in the fact that some concurrent nodes may not be able to execute before *n* due to some dependencies. For this reason, when including concurrent nodes, their dependencies must also be considered. If any nodes have no dependencies, or only have the same dependencies as *n* itself, these should be the first choic-

es. (If a node has the same dependencies as *n*, then those dependencies must have been satisfied since *n* is able to execute).

If all concurrent nodes have dependencies, it is not easy to determine statically which ones will be able to execute at the right time. A useful characteristic of Behavior Trees is that the nodes in any given thread cannot execute until their ancestors have executed, so the root node of a thread must execute before any of its descendents. Since the aim is to locate nodes which can execute before *n*, it is unnecessary to search further down a thread than the root nodes. If a node further down can execute before *n*, the root node can execute before *n* as well. The reverse is not always true, because the descendents may have additional dependencies which the root does not have. If the original model had traces where a parallel stuttering node executed before *n*, it is sufficient to include any root node of a parallel thread which has no dependencies other than those that *n* has. If there are not enough of those nodes in the tree, the next choice is to include root nodes which have dependencies. Since it is not possible to determine which of the dependency conditions will be satisfied, each of the root nodes must be included in the slice. This way, if one of them is able to execute before *n*, it will be able to do so in the slice. Including some nodes which are not able to execute before *n* will not change the outcome of the verification; it will only cause the slice to be more imprecise.

A function *slice_next* is introduced in the following definition. It returns the transition system of the slice that contains the extra stuttering nodes as described.

DEFINITION 49. NEXT PRESERVING SLICE
Let B be a transition system corresponding to a BT control flow graph G and S be a transition system such that $S = slice_{\varphi}(B)$ for some formula φ. Then, the function $slice\_next_{\varphi}(S)$ returns the transition system of the slice created from the slice set $nodes\_next_{\varphi}(G)$.

■

5.4.3 Proof of Correctness

In this section, a proof of correctness will be presented which shows that if extra nodes are inserted into a slice of a Behavior Tree according to the method described in the previous sections, the resulting slice will be next-preserving branching bisimilar to the original model. Using the results of previous chapters, it is easily established that the slice given by *slice_next* is related to the original model by a branching bisimulation with explicit divergence. This satisfies the first critierion of Definition 45. The remaining two criteria also hold if stuttering nodes have been included using the method described in this section.

THEOREM 5.
Let B be a transition system corresponding to a BT control flow graph and S be a transition system such that $S = slice_{\varphi}(B)$ or $S = slice\_inf_{\varphi}(B)$. Then, the transition system $T = slice\_next_{\varphi}(S)$ is next-preserving branching bisimilar to S, i.e. $S \overset{\star}{=}_{\varphi} T$.

Proof.
Let $S = (\mathcal{S}_1, AP_1, \mathcal{I}_1, \mathcal{L}_1, \mathcal{N}_1, \longrightarrow_1)$ and $T = (\mathcal{S}_2, AP_2, \mathcal{I}_2, \mathcal{L}_2, \mathcal{N}_2 \longrightarrow_2)$. In the following, let s, s', s_0, s_1, ... range over $\mathcal{S}_1$ and t, t', t_0, t_1, ... range over $\mathcal{S}_2$.
There are three criteria for next-preserving branching bisimulation, as given in Definition 45.

Criterion (*i*):

If $S = slice_{\varphi}(B)$

$\Rightarrow S \triangleq B$, by Theorem 2.

Otherwise, if $S = slice\_inf_{\varphi}(B)$,

$\Rightarrow S \subseteq S'$, where S' the transition system such that $S' = slice_{\varphi}(B)$, by Theorem 3.

$S' \triangleq B$, by Theorem 2.

$\Rightarrow S \triangleq B$.

Criterion (ii):

$\forall s', s'', t''$, such that $s'' \mathcal{R} t''$, if $s \overset{j}{\dashrightarrow} s' \xrightarrow{n_x} s''$ and $obs_{\varphi}(n_x)$, then $t \overset{k}{\dashrightarrow} t'$, where $t' \mathcal{R} s'$ and $k \geq min(j, x\text{-}depth(\varphi))$.

This holds, by the definition of *extra*.

Criterion (iii):

$\forall s', s'', t''$, such that s' R t' and s'' R t'', if $s \overset{j}{\dashrightarrow} s' \longrightarrow s''$ and $s' \longrightarrow s'''$, where $s'' \neq s'''$ and $\exists \alpha \in \mathcal{N}_1$ such that $obs_{\varphi}(\alpha)$ and α is reachable from s'' but not from s''', then $t \overset{k}{\dashrightarrow} t'$, where $k \geq min(j, x\text{-}depth(\varphi))$.

If $s \overset{j}{\dashrightarrow} s' \xrightarrow{n_y} s''$ and $s' \xrightarrow{n_z} s'''$, where $s'' \neq s'''$,
then either: (where n_x is the node that executed immediately before s')

By Lemma 10, the only cases are:

(a) $\exists\, n_1, n_2$ such that $parent(n_1) = parent(n_2) = n$ and $alt(n_1, n_2)$, or

(b) $flag(n) = threadKill$ or

(c) $type(n) = selection$.

$\Rightarrow \exists k$ nodes in $\mathcal{N}_2$ that can execute immediately before m, where $k \geq min(j, x\text{-}depth(\psi))$, by the definition of *extra*.

□

Theorem 1 In this chapter, it has been shown that next-preserving branching bisimulation guarantees the preservation of all CTL* formulas, including the next step. This result can be used for any application that requires all CTL* formulas to be preserved. The user only has to demonstrate that a next-preserving branching bisimulation exists for their two models. Note that since this method involves including additional nodes, it is unnecessary for formulas which do not contain the X operator. For these formulas, it would be better to use the normal slicing algorithms presented in previous chapters. Furthermore, this chapter has demonstrated that a next-preserving branching bisimulation holds between a Behavior Tree B and a slice given by *slice_next*(B).

6 CASE STUDIES

This chapter demonstrates the techniques presented in previous chapters on two case studies. Section 6.1 describes how the slicing algorithms have been implemented as part of an existing Behavior Tree editor. The subsequent sections describe the case studies. For each case study, an existing Behavior Tree was used, along with a set of properties to be model checked. A set of slices were constructed, one for each property, using the algorithms given in Chapter 3. The time taken to verify the slices was then compared with the time taken for the original model. The two case studies were selected in order to demonstrate the use of slicing on different types of models, where the Behavior Trees have different structures. The first case study, a mine pump, is typical of many embedded systems, which have a software controller and hardware sensors and actuators. The components communicate with each other via message passing. This case study is described in Section 6.2. The second case study is a hospital information system. It is a typical database system, which stores information about various users of the system. The accessing of information is governed by a set of access control rules. The details of this case study are given in Section 6.3.

6.1 Slicing Implementation

The slicing algorithm presented in Chapter 3 has been implemented by the author. The slicer operates as part of the existing Behavior Tree editor called Integrare (Wen, et al., 2007). Integrare is a Behavior Tree drawing editor which also includes functionality such as extracting keywords from textual requirements, linking to a simulator and automatically translating Behavior Trees into the model checking languages SAL (de Moura, et al., 2004) and UPPAAL (Larsen, et al., 1997). A screenshot of the Integrare tool is given in Figure 53. The new slicing component links to the existing SAL translator, which is a function of Integrare. The SAL translator is described briefly in Section 2.3.3. Full details of the translation process can be found in Grunske et al. (2008). Figure 54 shows an example of the translation output given by the SAL translator.

The slicing component was written in Visual C++ (MFC), to allow it to be compatible with the existing Integrare source code. The slicer operates according to the algorithms given in Chapter 3. It takes a list of components as an input. These are the components mentioned in the temporal logic theorem to be model checked. The slicer then automatically creates a dependence graph for the selected Behavior Tree. The dependence graph is stored in memory, not explicitly shown to the user. Using the list of components, the nodes that form the slicing criterion are then identified. The slice is then created by traversing the dependence graph starting at the criterion nodes. The nodes collected during the traversal are re-formed into a syntactically correct Behavior Tree. The slice is then passed to the translator, which treats it as an ordinary Behavior Tree and translates it into the requested model checking language.

The overall system is illustrated in Figure 55. The new section of the tool, the slicer, is shown in a dotted box amidst the existing functions. These concepts are not limited to the Integrare tool; it simply demonstrates that the slicing algorithms of this thesis can be implemented. In a similar manner, the slicing algorithms could be implemented to interface with any Behavior Tree editor, such as the latest Behavior Tree editor TextBE (Myers, 2011).

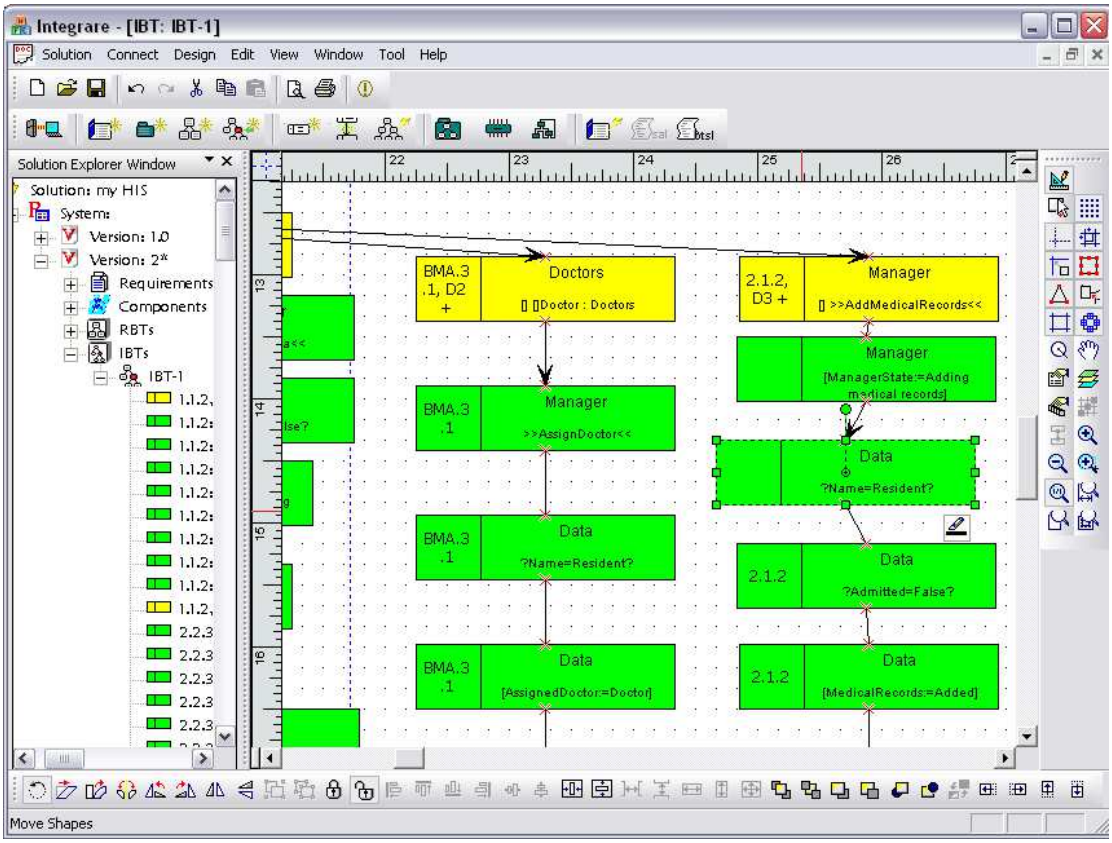


Figure 53. A Screenshot of the Integrare Drawing Pane

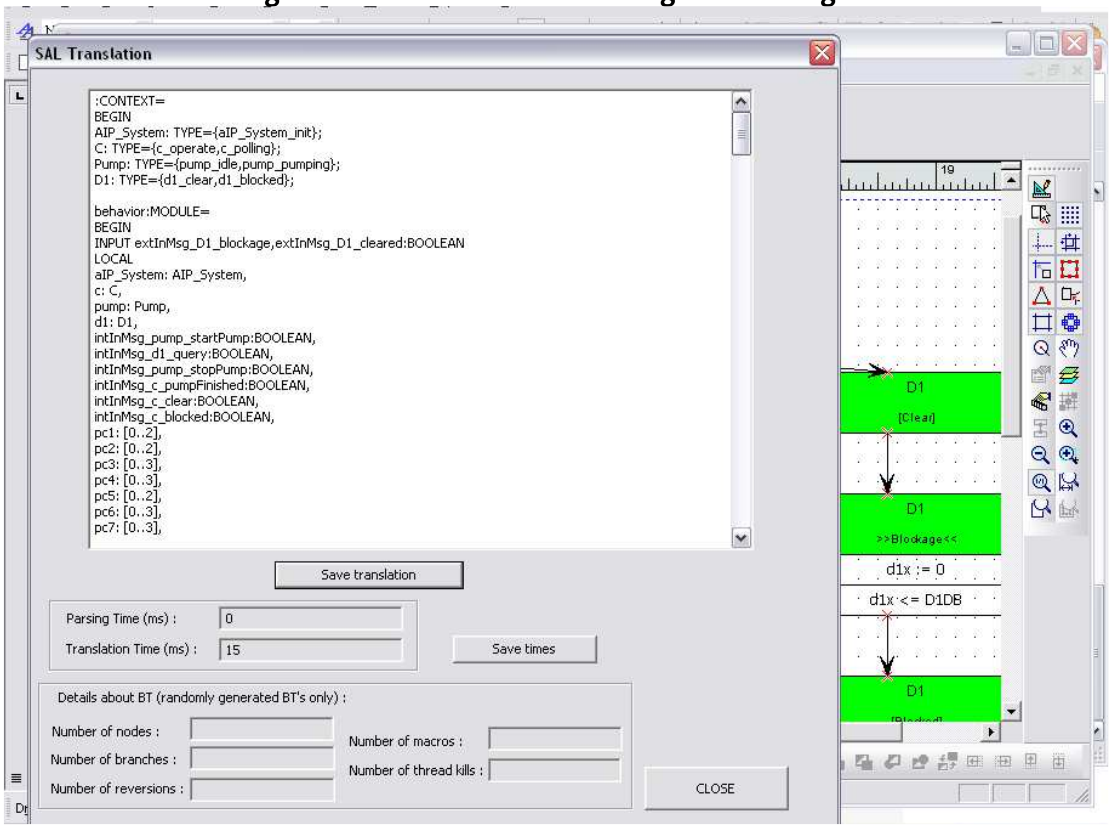


Figure 54. Screenshot Showing Translation Pop-up Window.

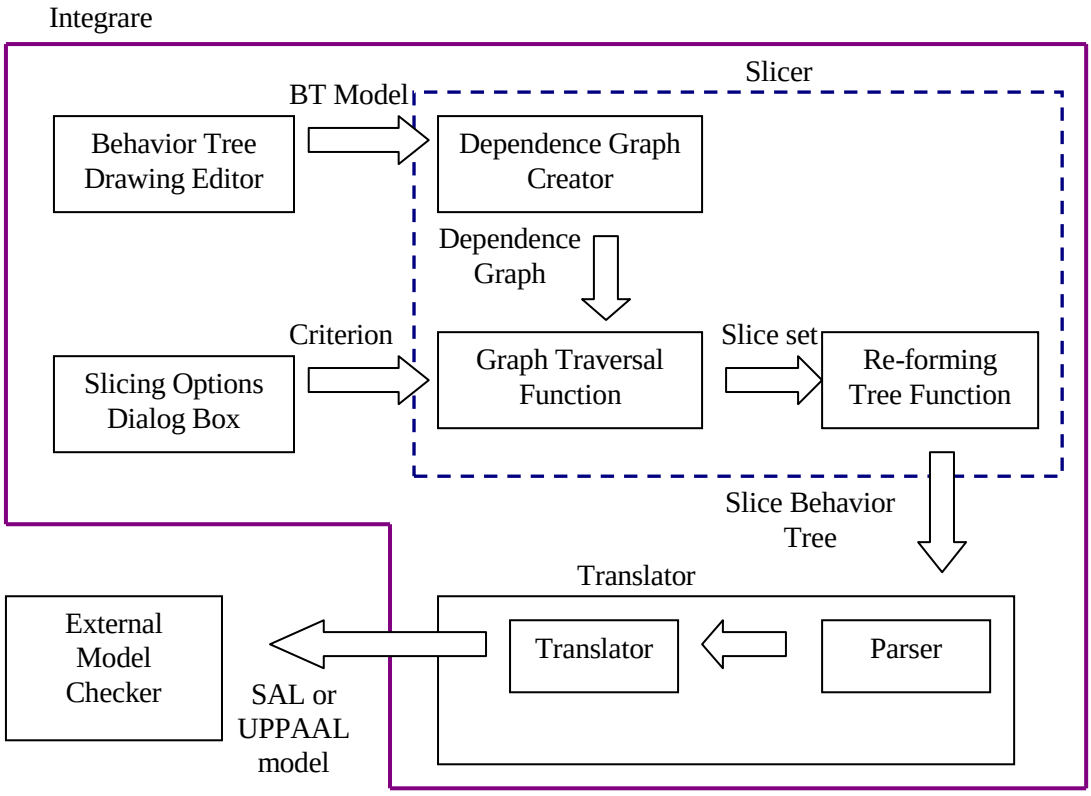


Figure 55. The Slicing Tool as Part of Integrare.

6.2 Mine Pump Case Study

This section presents a case study of a mine pump, taken from Grunske et al. (2011). The case study was selected because of its structure which is common to many embedded systems, where a software controller makes decisions and interacts with hardware sensors and actuators. The system also receives input from the environment. Two more systems in the same style are given by Grunske et al. (2011).

The case study models a system which controls the amount of water in a mine using a water pump. Two sensors indicate the current level of the water: a low water sensor indicates when the water has reached a low level, while a high water sensor indicates when the water has reached a high level. The pump automatically activates when the water is high, pumping the water out until it is a normal level again. Similarly, when the water reaches a low level, the pump automatically turns off. There is a mechanism to allow a human operator to control the pump, as long as the water is between the high and low levels. A supervisor has higher authority. He or she may turn the pump on or off regardless of the level of the water. Three other sensors monitor the environment for health and safety reasons. These are: an airflow sensor, a CH_4 (methane) sensor and a CO (carbon monoxide) sensor. If the methane reaches a critical high level, it is imperative that the pump be turned off and remain unoperational until the methane levels have dropped back to a normal level. The other safety precaution is that if the airflow becomes critically low or the CO levels become critically high, the personnel must immediately evacuate the mine. In this case, the system must trigger an alarm to warn the personnel.

6.2.1 Behavior Tree of the Mine Pump

The Behavior Tree of the mine pump is designed in a style where each component is modelled separately in individual threads. This reflects the component-based structure of the actual system. Communication between the threads is accomplished using message passing. The full Behavior Tree

is too large to be shown here, so its basic structure is given in Figure 56. The tree consists of eleven main threads; one for each of the components, people and environmental aspects interacting with the system: the software controller, pump, supervisor, operator, personnel, CH_4 sensor, CO sensor, airflow sensor, low water sensor, high water sensor and the environment. The controller maintains internal representations of each of the sensors and the pump, in order to keep track of each component's status. The controller thread is sub-divided into a main thread which describes the behaviour of the controller and several threads which continually check the state of the sensors and pump and update the controller's internal representations of them.

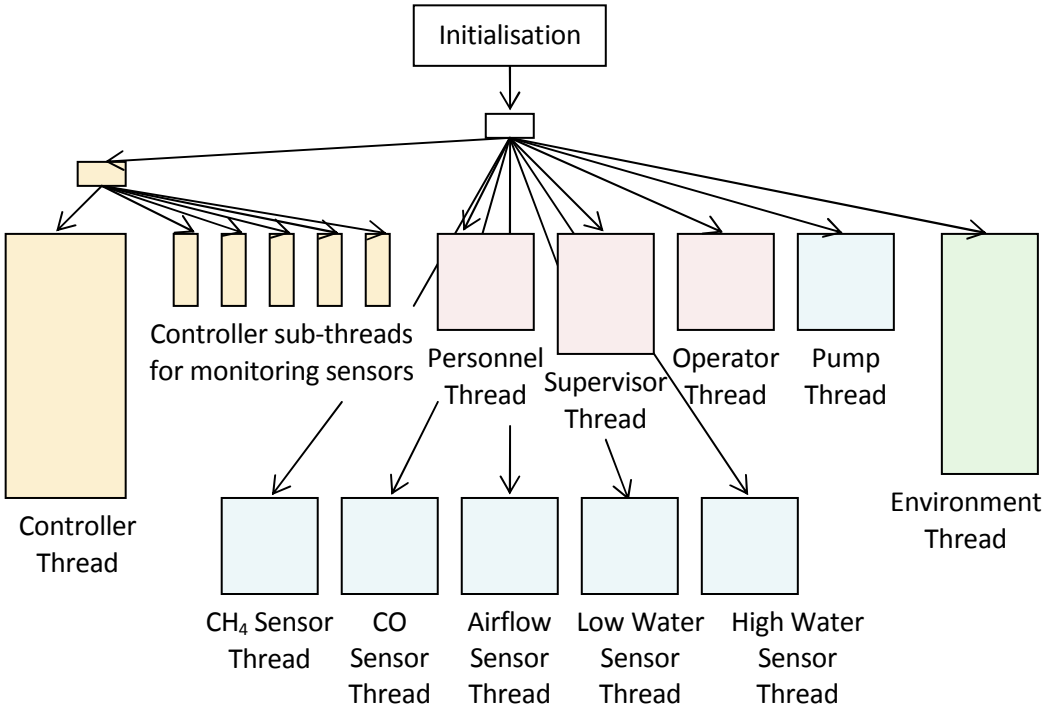


Figure 56. Overview of the Mine pump Behavior Tree.

A part of the controller thread is shown in Figure 57. The main responsibility of the controller is to control the pump by sending requests for it to turn on or off as required. The controller thread is guarded by a node that checks whether the CH_4 is at a normal level. If so, the pump can operate. The controller then decides its next actions depending on whether the pump is currently on or off. If it is on and the water level has reached a low level, the controller sends a message to turn off the pump. Alternatively, if the operator requests to turn off the pump, the controller checks whether the water is between the high and low levels and if so, sends a message to turn off the pump. The supervisor may also request to turn off the pump. (This behaviour has not been shown here to save space). This action is allowed regardless of the current water level. The behaviour of the controller when the pump is off is the exact opposite: the controller sends a message to turn on the pump if either the water level is too high, the supervisor requests it or the operator requests it and the water level is normal.

The supervisor and operator threads are quite similar. Both of them non-deterministically decide to turn on or off the pump and send requests to the controller as required. Neither of the threads perform any functions if the controller decides that the pump must remain non-operational due to high methane levels. The personnel thread simply waits to be instructed to either enter or exit the mineshaft. These messages are sent by the controller, depending on the state of the CO and airflow sensors. There are no output message nodes in the personnel thread.

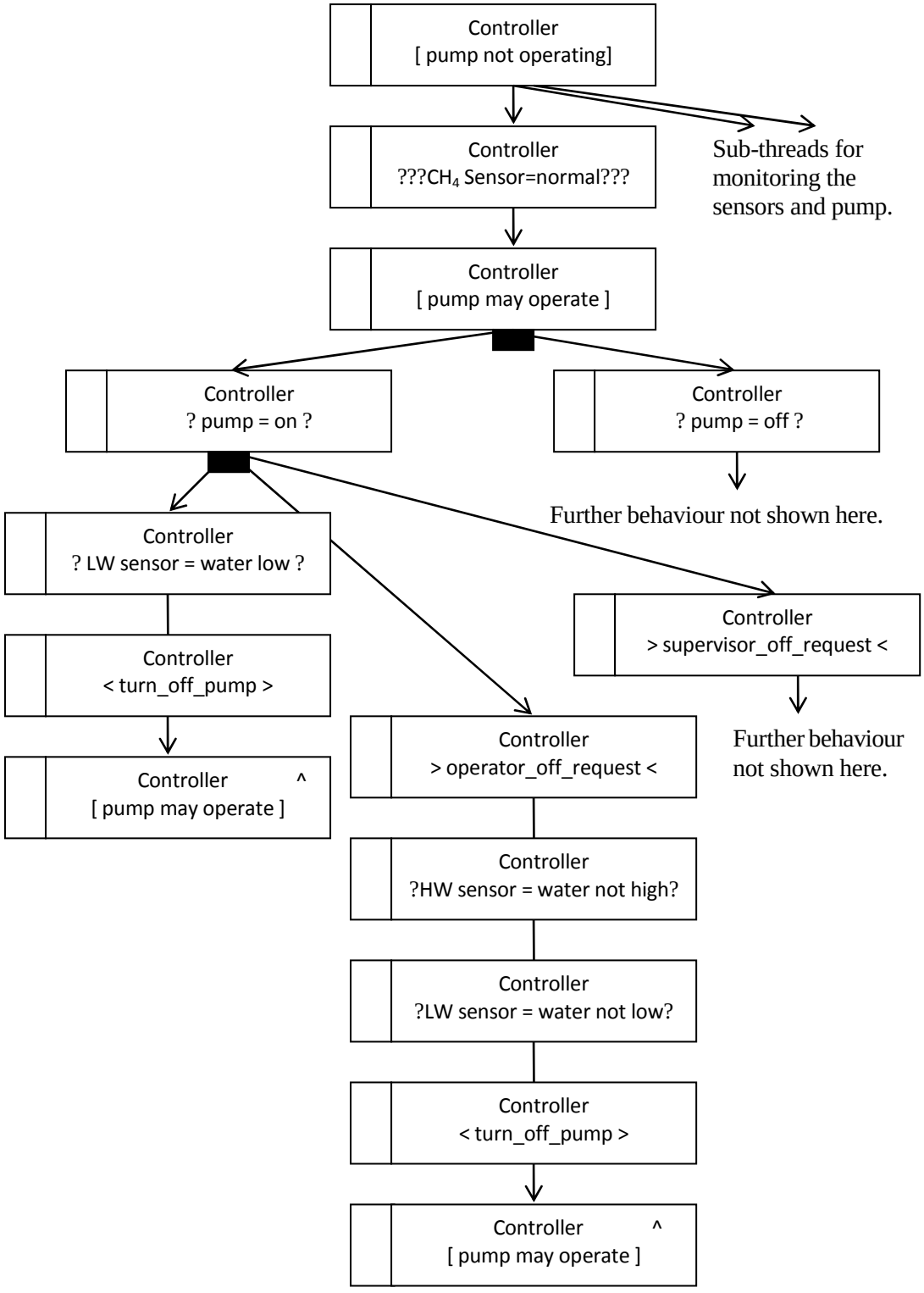


Figure 57. Part of the Controller Thread

The five sensor threads are all almost identical. Each waits for messages from the environment. After updating the state to reflect the environmental change, each sensor then sends out a message to the controller to inform it of its change in state. These actions all occur atomically. For example, the CO sensor, shown on the left of Figure 58, changes to the *high CO* state when it receives the

high_CO_level message from the environment and to the *normal CO* state when it receives the *normal_CO_level* message. It then sends out the messages *high_CO_detected* and *normal_CO_detected*, respectively.

The pump thread is also very similar to the sensors, except that instead of responding to messages from the environment, it responds to messages from the controller, requesting the pump to turn on or off.

The environment thread represents the external environment. It is divided into four threads, representing the methane, the airflow, the carbon monoxide and the water in the mine pump. External input messages model the environmental changes. Each external input message causes a change in state of one of the environment attributes. Additionally, a message is sent out to the corresponding sensor to inform it of the change. This occurs atomically, to prevent erroneous behaviour caused by interleavings with other threads before the message has been sent. Part of the environment thread describing the carbon monoxide is shown on the right of Figure 58. Note that the *high_CO_level* output message corresponds to the input message the CO sensor is waiting for.

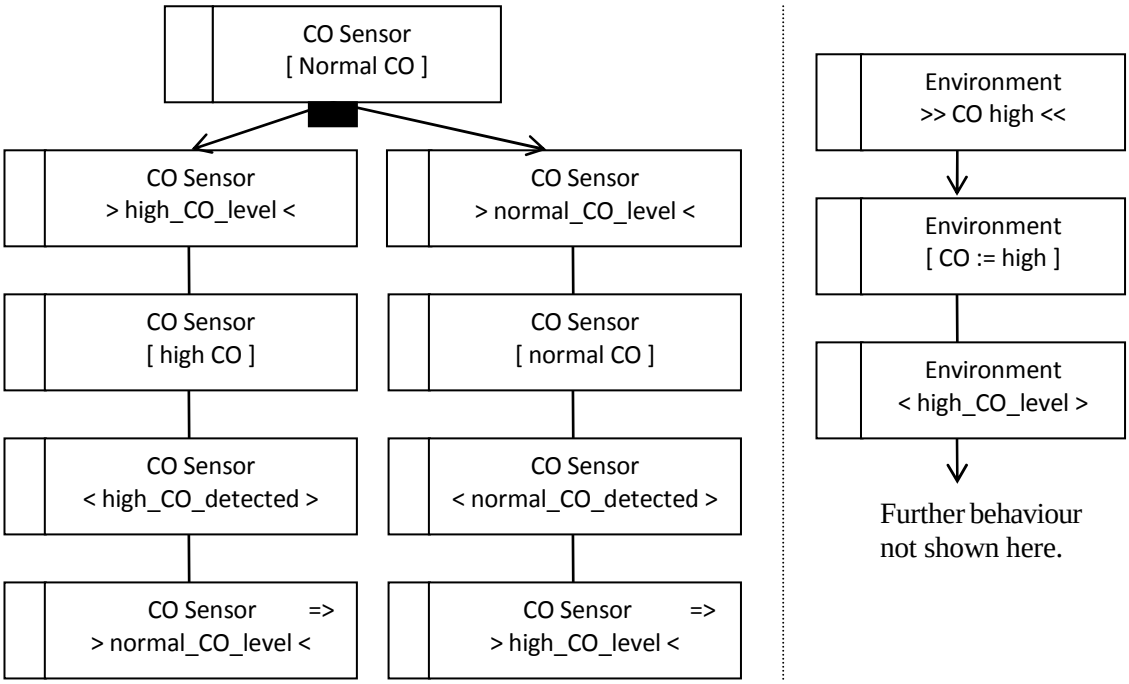


Figure 58. CO Sensor Thread and Environment CO Thread

In (Grunske, et al., 2011), this mine pump Behavior Tree was used for Failure Modes and Effects Analysis (FMEA). However, the model used here contains some modifications in order to more accurately reflect the requirements. The most significant change is that in the version in (Grunske, et al., 2011), when the pump is turned off due to high methane levels, the supervisor can still turn it back on, whereas in this version, the pump remains non-operational until the methane returns to a normal level. Due to these differences, the model checking times given in (Grunske, et al., 2011) are significantly different to those given in the next section.

6.2.2 Slicing and Verification of the Mine Pump

There are three safety properties which the mine pump must fulfill. If the methane reaches a critically high level, it is dangerous to continue operating the pump so it must be switched off. If the airflow reaches a low level, or the carbon monoxide reaches a high level, it is unsafe for the personnel in the mineshaft, so they must evacuate. These properties have been formalised as temporal logic theorems as follows:

Th1. G (environment_CH_4 = high ⇒ F(pump = off))
It is always the case that if the methane is high, eventually the pump will be off.

Th2. G (environment_airflow = low ⇒ F(personnel = notInMineshaft))
It is always the case that if the airflow is low, eventually the personnel will not be in the mineshaft.

Th3. G (environment_CO = high ⇒ F(personnel = notInMineshaft))
It is always the case that if the carbon monoxide is high, eventually the personnel will not be in the mineshaft.

In (Grunske, et al., 2011), the properties were formalised slightly differently, with nested X operators instead of the F operator in each theorem. Using those formulas, the pump is required to turn off within a certain number of steps and similarly for the personnel in the mineshaft. The weaker forms of the theorems, as shown above, have been chosen here in order to demonstrate the application of the normal slicing algorithm presented in Chapter 3, without having to add extra nodes according to the method given in Chapter 5.

The first step is to create the dependency graph for the mine pump Behavior Tree. This can be reused for each of the theorems. The dependency graph is created internally by the slicing tool. Within each thread, there are control dependencies to selection and external input nodes. There are no data dependencies in the model, but there are interference dependencies between the main thread of the controller and the sub-threads which update the various attributes of the controller. There are message dependencies between many of the components. In particular, nodes in each sensor thread are message-dependent on some nodes in the environment threads. Nodes in the controller thread are message-dependent on nodes in each of the sensor threads, as well as the supervisor and operator threads. This reflects the role of the controller, which is to monitor the state of these other components and control the pump accordingly. There are no synchronisation dependencies. Finally, there are termination dependencies caused by some of the reversions. An interesting example of this is the reversion which executes after the controller learns that the methane is high. Its target is an ancestor of all of the controller sub-threads, so when the reversion executes, all of those threads are terminated. None of the controller's behaviour can be re-started until the methane is detected to be at a safe level again.

Theorem 1

The next step is to locate the nodes that will be in the slice, by performing backwards traversals of the dependency graph starting at the criterion nodes. Consider the first theorem. The slicing criterion is {Environment_CH_4, Pump}. Therefore the variables of interest are the pump and the attribute representing the methane in the environment. The criterion nodes are any state realisations which modify either of these variables. In this Behavior Tree, there are four such nodes: Pump[off], Pump[on], Environment[CH_4 := high] and Environment[CH_4 := normal].

Some of the relevant dependencies are given in Table 2. The criterion nodes are shown in italics. The two environment nodes only have control dependencies to the external input nodes Environment>>CH_4 high<< and Environment>>CH_4 normal<<. The external input nodes do not have any further dependencies, so the traversal ends there. The Pump [off] node has a control dependency to the internal input nodes Pump>turn_off_pump<. This in turn has message dependencies to four corresponding output message nodes in the controller thread, which have each been labelled with a number in the table to avoid ambiguity. As shown in the controller thread in Figure 57, the controller sends these messages in response to the state of the low water sensor or according to requests made by the supervisor or controller. The message is also sent if the CH_4 levels are high. The dependency chains starting at each Controller <turn_off_pump> node can be seen in the table. The first turn_off_pump node leads to nodes in the low water sensor and environment threads. The other two turn_off_pump nodes have dependencies to nodes in the supervisor and operator threads. (The dependencies of the supervisor and operator nodes have not been shown in the table).

The final slice set contains most nodes in the pump, supervisor, operator, low water sensor, high water sensor, environment_CH_4 and controller threads. In the controller thread, the main operation of the controller is included in the slice set, as well as the threads responsible for monitoring the states of the CH_4, low water and high water sensors and the pump. None of the nodes in the personnel, env_airflow, env_CO, airflow sensor and CO sensor threads are in the slice set. This is intuitively correct, as there is no interaction from these components that causes the pump to turn on or off.

Node	Dependent on	Dependency
Environment [CH_4 := high]	Environment >> CH_4 high <<	cd
Environment [CH_4 := normal]	Environment >> CH_4 normal <<	cd
Pump [off]	Pump > turn_off_pump <	cd
Pump > turn_off_pump <	Controller < turn_off_pump > 1	md
	Controller < turn_off_pump > 2	md
	Controller < turn_off_pump > 3	md
	Controller < turn_off_pump > 4	md
Controller < turn_off_pump > 1	Controller ???LW sensor=low???	cd
Controller ???LW sensor=low???	Controller [LW sensor:=low]	id
Controller [LW sensor:=low]	Controller > water_detected_low<	cd
Controller > water_detected_low<	LW Sensor < water_detected_low >	md
LW Sensor <water_detected_low >	LW Sensor > water_low <	cd
LW Sensor > water_low <	LW Sensor > water_not_low <	td
	Environment < water_low >	md
Environment < water_low >	Environment >> water below limits <<	cd
LW Sensor > water_not_low <	Environment < water_not_low >	md
Environment < water_not_low >	Environment >>water within limits <<	cd
Controller < turn_off_pump > 2	Controller > supervisor_off_request <	cd
Controller > supervisor_off_request <	Supervisor <supervisor_off_request >	md
Controller < turn_off_pump > 3	Controller ???LW sensor=not_low???	cd
	Controller ???LW sensor=not_high???	cd
	Controller > operator_off_request <	cd
Controller > operator_off_request <	Operator < operator_off_request >	md

Table 2. Some of the Dependencies Relevant for Th1 of the Mine pump

Next, the reversions and reference nodes must be added to the slice if necessary. If all of the reversions and reference nodes were added back to the slice, the final slice would contain the same number of interleaving threads as the original model. Even the threads which do not contain any relevant nodes, such as the airflow sensor thread, would have to remain in the slice due to their reversions.

Further reductions can be obtained using the approach presented in Section 3.4.3 for reducing the number of reversions and reference nodes. Using this approach, several reversions and reference nodes can be removed from the mine pump slice. All of these are nodes causing divergence, due to infinitely reverting inside unnecessary threads. All of these threads start after the same node, the root node Mine pump>>Ready<<. None of these diverging reversions and reference nodes have any control dependencies to nodes already in the slice. Thus, they can all execute in the same traces and only one of them is necessary. In this case, a reversion in the personnel thread has been chosen as the one to remain in the slice.

Similarly, there are sub-threads of the controller that are not necessary in the slice; in particular the sub-threads responsible for monitoring the CO sensor and airflow sensor. The reversions in these threads are all descendents of the node Controller?CH4=normal?. When compared to each other, it is found that only one of them is needed in the slice to represent the several equivalent divergent traces.

The final result is that only two threads which contain entirely stuttering behaviour must remain in the slice. This is significantly less than if all stuttering threads had remained.

The next stage is to translate the final slice into the SAL input language for model checking. The translation process includes the option of specifying the type of message passing used. In this case, non-buffered message passing was required, in order to prevent miscommunication caused by messages arriving too long after the corresponding state change occurred. Additionally, prioritisation was applied to the SAL model, in order to ensure that internal messages will be immediately received if the receiver is ready. To achieve this, internal messages are given the highest priority, followed by all other nodes except external inputs and finally external inputs are given the lowest priority. This effectively prevents spurious counterexamples in which the system responds to external input messages faster than its internal state realisations occur. The same approach was used in (Grunske, et al., 2011).

Table 3 compares the final slice for Th1 with the original model. The number of transitions is equivalent to the number of nodes in the control flow graph, counting each atomic block as a single transition. The number of PC's (program counters) reflects the amount of branching in the tree, since a new program counter is created for each alternative or concurrent branch. The number of threads shows the number of concurrent threads. All three of these measurements were reduced in the slice, although not by large amounts. Despite this, the time taken to verify the theorem was significantly reduced. Both models were verified using the SAL symbolic model checker, running on an AMD Opteron 6174 processor at 2.2 GHz. (The processor was part of a 48-core cluster, but only one processor was allocated to this process). To verify the theorem on the original model, the model checker did not provide a result in over 24 hours, at which point it was terminated manually. For the slice, the model checker was able to provide a response in just 1.5 hrs. The model checker found the property to be invalid but was unable to find the counterexample. Nevertheless, it is still an improvement over the original model, for which no result was given at all within 24 hours.

	No. of Transitions	No. of PC's	No. of Threads	Verification Time
Original	124	70	56	> 24 hrs
Slice	102	59	43	Approx.* 1.5 hrs

* The model checker did not provide the verification time statistics in this case, so the time was noted approximately.

Table 3. Original Model vs. Slice for Th1 of the Mine Pump

Theorems 1 and 2

The next two theorems are both very similar. The slicing criterion for Th2 is {Environment_airflow, Personnel} and the criterion for Th3 is {Environment_CO, Personnel}. The criterion nodes for Th2 are: Environment[airflow := low], Environment[airflow := normal], Personnel[in mineshaft] and Personnel[not in mineshaft]. Similarly, the criterion nodes for Th3 are: Environment[CO := high], Environment[CO := normal], Personnel[in mineshaft] and Personnel[not in mineshaft]. Table 4 gives some of the relevant dependencies for Th3. The environment nodes are only dependent on external input nodes, as for Th1. The Personnel[not in mineshaft] node has a control dependency to an internal input node, which is in turn message dependent on two controller output message nodes, both named Controller <evacuate_mineshaft>. One of these output messages is sent in response to the airflow being low, while the other is sent in response to the CO level being too high. This is the cause of the symmetry of the slices for both Th2 and Th3. Since both theorems have the same personnel nodes in their criterion set, the resultant slice sets both contain the nodes given in the table, i.e. nodes from the airflow sensor, CO sensor and the environment. The table does not list the dependencies relevant to the Personnel[in mineshaft] node, which are nodes from the controller, airflow sensor and CO sensor. There are also further dependencies not listed, which are due to termination dependencies from alternative branches.

All of the relevant nodes from the controller thread are in the sub-threads which monitor the sensors; the main controller thread is not relevant. This results in significantly smaller slices than the original model. For both theorems, the final slice sets do not contain any nodes in the pump, operator, supervisor, low water sensor and high water sensor threads. Again, this is intuitively correct, as none of these components influence the behaviour of the personnel.

Node	**Dependent on**	**Dependency**
Environment [CO := high]	Environment >> CO high <<	cd
Environment [CO := normal]	Environment >> CO normal <<	cd
Personnel [not in mineshaft]	Personnel > evacuate_mineshaft <	cd
Personnel > evacuate_mineshaft <	Controller < evacuate_mineshaft > 1	md
	Controller < evacuate_mineshaft > 2	md
Controller < evacuate_mineshaft > 1	Controller > low_airflow_detected <	cd
Controller > low_airflow_detected <	Airflow Sensor <low_airflow_detected>	md
Airflow Sensor >low_airflow_detected<	Airflow Sensor > low_airflow_level <	cd
Airflow Sensor > low_airflow_level <	Environment < low_airflow_level >	md
Environment < low_airflow_level >	Environment >> airflow low <<	cd
Controller < evacuate_mineshaft > 2	Controller > high_CO_detected <	cd
Controller > high _CO_detected <	CO Sensor < high _CO_detected >	md
CO Sensor > high _CO_detected <	CO Sensor > high _CO_level <	cd
CO Sensor > high _CO_level <	Environment < high _CO_level >	md
Environment < high _CO_level >	Environment >> CO high <<	cd

Table 4. Dependencies Relevant for Th3 of the Mine Pump

In the same way as for the first theorem, the number of reversions and reference nodes that remain makes a significant different to the size and number of threads in the slice. For theorems 2 and 3 in particular, there are many divergent threads which do not need to remain in the slice. Using the same approach as before, most of the reversions and reference nodes in these threads can be removed from the slice, since they are equivalent to others. As with the first theorem, only one reversion or reference node in a divergent thread is needed after Mine pump>>Ready<<. Out of the controller sub-threads with divergent behaviour, only one of the reversions or reference nodes in these threads is needed as well. The main controller thread can be left out of the slice completely. The following tables show statistics about the slices for Th2 and Th3, respectively.

	No. of Transitions	No. of PC's	No. of Threads	Verification Time
Original	124	70	56	10.5 hrs
Slice	41	23	21	3.69 s

Table 5. Original Model vs. Slice for Th2 of the Mine Pump

	No. of Transitions	No. of PC's	No. of Threads	Verification Time
Original	124	70	56	> 24 hrs
Slice	41	23	21	3.42 s

Table 6. Original Model vs. Slice for Th3 of the Mine Pump

As can be seen, both slices contain exactly the same number of transitions, program counters and threads. This is due to the symmetry of the tree, in which both the airflow and CO are monitored in exactly the same manner, requiring exactly the same number of nodes. The slices were much smaller than the original model, with less than half the number of transitions, program counters and threads. The verification times for both theorems on the original model were very large. Theorem 2 took 10.5 hours, while the model checker did not produce a result for Theorem 3 for over 24 hours. In comparison, the slices for both cases were extremely fast, both taking less than 4 seconds to be proved.

The results of this case study demonstrate that slicing can improve the verification time dramatically. However, it depends on several factors, including the property to be verified and the model. The first theorem involved the pump, so the final slice was not as small as for the other two theorems and the verification time was not as fast. Nevertheless, even for Theorem 1, the slice enabled a result to be obtained for a model which previously could not be verified.

6.3 Hospital Information System Case Study

The case study presented in this section is a model of a hospital information system. It describes a general information system that could be used for a hospital or health care facility. The requirements for it are a simplified version of those given by Zafar (2008), which were in turn based on the case study presented by Evered and Bögeholz (2004). The system has a typical database design, where information about each of the users is stored and accessed by others. Access control policies dictate which users are allowed to access which types of information. The use of Behavior Trees for modelling access control policies was proposed by Zafar et al. (2007).

The system manages information about each of the residents of the facility. Each resident may be associated with a representative who is able to sign documents on behalf of them. Doctors and managers are also users of the system. Managers may add personal details and previous medical records for a resident to the system before the resident is admitted, but are not permitted to add or update medical records afterwards. Furthermore, managers may delete a resident's medical records, but only if a certain period of time has elapsed since the resident left the facility. Doctors may add, delete or update medical records at any time and assign residents, doctors or managers to the appropriate access control lists. The access control lists specify which users have access to each document in the system. The documents include the medical records for each resident, the private notes of each doctor and the plan of care for each resident. Visiting doctors may be temporarily assigned access to a resident's medical records.

6.3.1 Behavior Tree of the Hospital Information System

The Behavior Tree used in this section is a modification of the one given by Zafar (2008). Another version of this Behavior Tree was used to demonstrate slicing in Yatapanage et al. (2010). However, the results obtained using that model differ from the results presented here due to the differences between the two models. This version rectifies some minor problems and more accurately reflects the requirements of the system.

The Behavior Tree uses the notion of *sets*, to represent the sets of each group of users: managers, doctors, residents and representatives and the set of data files and log files. There are four main threads in the Behavior Tree, each corresponding to a type of user. An overview of the Behavior Tree is given in Figure 59. The full Behavior Tree is too large to be shown here. Each thread uses *for-all* nodes to describe the behaviour of a particular user. For example, the manager thread describes the behaviour of each manager. Each thread consists of a set of alternative branches, each of which begin with an external input node. The external input nodes represent the action to be performed. For example, the manager has a branch with the external input node addPersonalDetails, which describes the behaviour of adding the personal details of a resident. Every branch ends with a reversion to the System [... := Ready] node of that thread.

Two of the branches of the manager thread are shown in Figure 60. After the system node, there is a *for-all* node, stating that the thread describes the behaviour for all elements, m, from the set of manag-

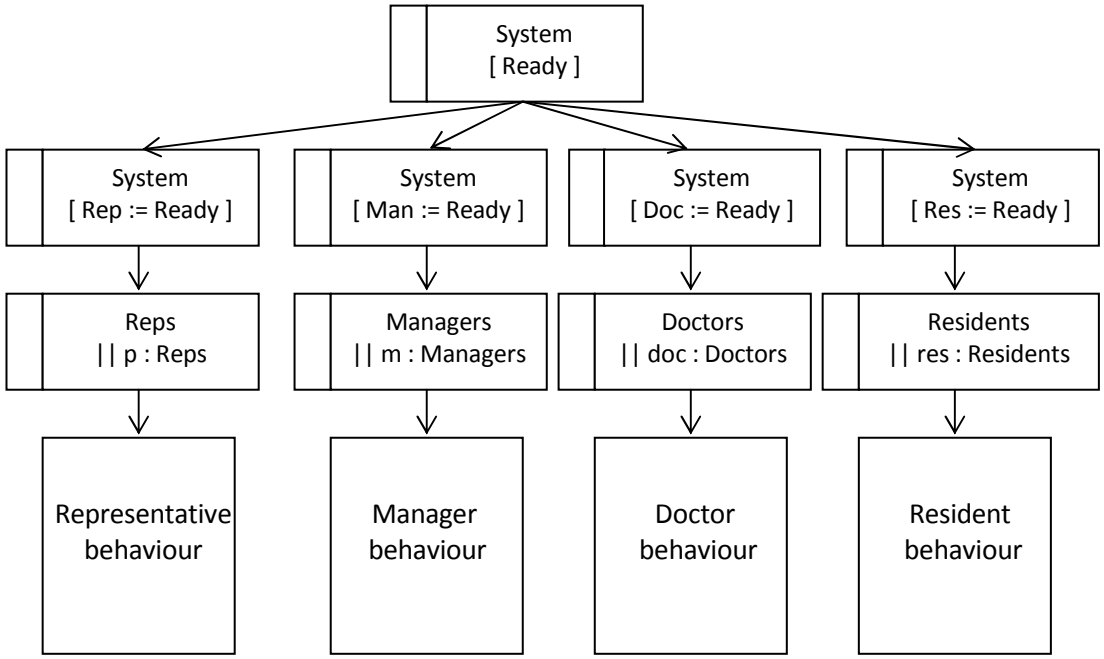


Figure 59. Overview of the Hospital Information System Behavior Tree.

ers. In the sub-tree below, every time a node refers to m, it is referring to the current element of the set of managers. Below the *for-all* node, there are two *for-one* nodes, one which selects a particular resident from the set of residents and one which selects a particular data file from the set of data files. The resident and data file are needed to allow the manager behaviour to be described in terms of a particular resident and data file. For example, when the manager decides to delete a data file, as shown in the figure, it is first checked that the data file belongs to the currently selected resident. The data file can only be deleted if its time attribute, representing the time that has elapsed since the resident left, is above the pre-defined limit. The other thread shown in the figure describes the manager adding medical records. The medical records are only added if the data file belongs to the current resident and if the data file's admitted attribute is false, representing that the resident has not yet been admitted to the facility. The other manager branches are all similar.

The other three threads are all designed in the same style, each with a set of alternative branching nodes. The doctor thread is the largest. The doctors are responsible for assigning users to the various access control lists. For example, the viewNotesACL attribute is the access control list that specifies which users have access to each of the private notes. The doctor may assign a resident to this list, in order to allow them to view the notes regarding them. This is shown on the right of Figure 61. The nodes shown in the figure are one of the branches of the doctor thread, where Doc is the current doctor, d is a chosen data file and res is a chosen resident. If the data file belongs to the chosen resident and the doctor assigned to the data file is the current doctor, then the doctor is allowed to assign the resident to the access control list. This is accomplished by updating the list to contain the resident. The doctor thread also describes the other functions of doctors, such as adding medical records and adding visiting doctors to access control lists.

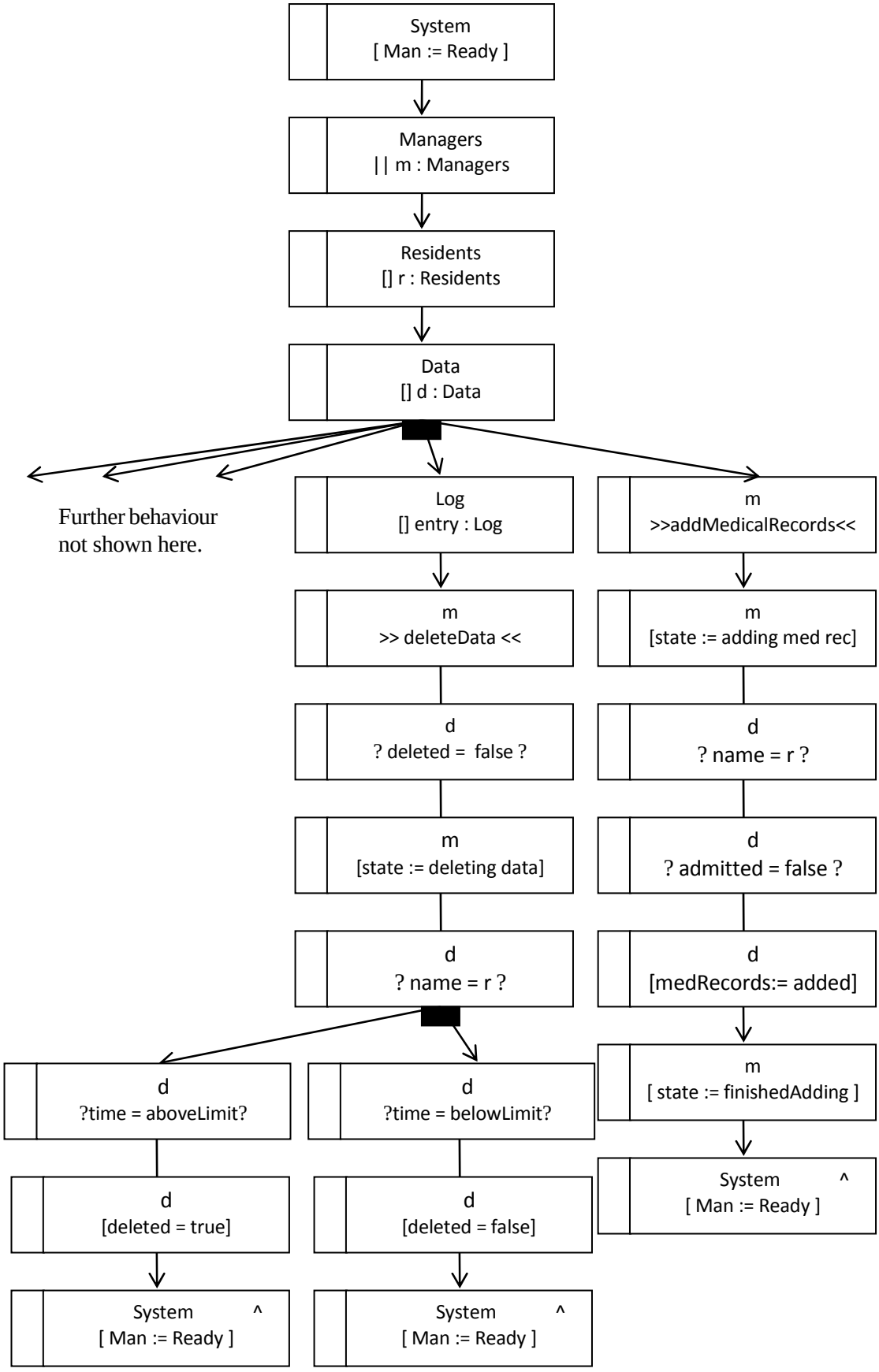


Figure 60. Part of the Manager thread.

The resident thread allows the residents to view the various data files associated with them, if they are permitted to do so according to the access control lists. A branch of the resident thread is shown on the left of Figure 61, where Res is the current resident and d is a chosen data file. When the resident requests to view a data file's private notes, it is first checked whether the data file belongs to the resident and whether the resident is a member of the access control list.

The thread for the representatives is the smallest thread in the Behavior Tree. The representatives are able to sign agreements on behalf of the patient and view their care plan.

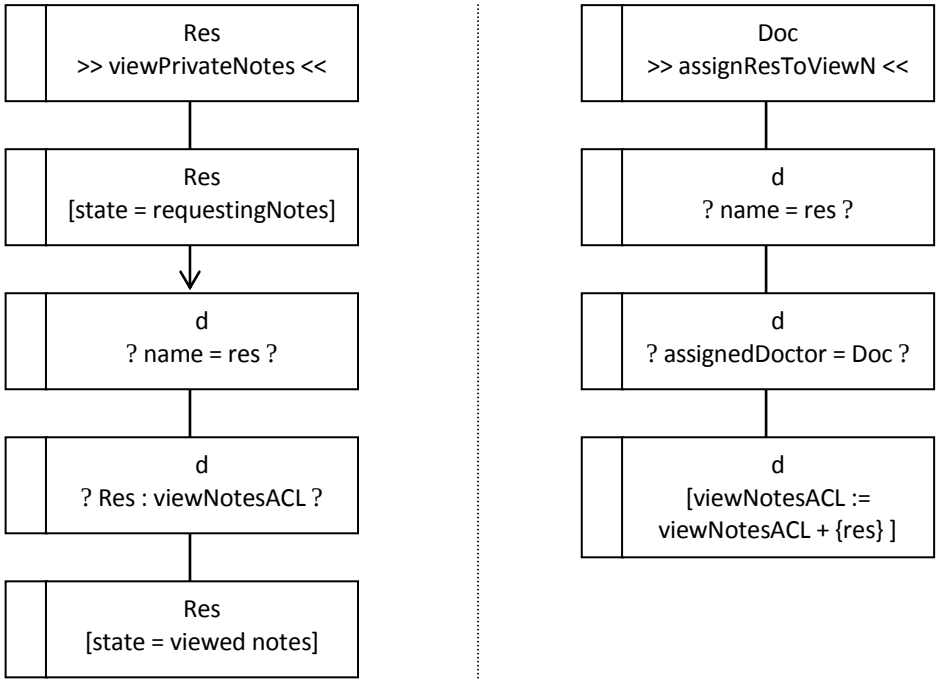


Figure 61. Part of the Resident and Doctor threads.

Unlike the previous case study, the components of the system do not communicate using messages. The users each access the central data files according to their own access control privileges. This style would be the typical design for any similar database system.

6.3.2 Slicing and Verification of the Hospital Information System

There are a number of privacy properties which must be satisfied in this system. In this section, three such properties will be investigated:

Th1. $\forall m \in Managers, d \in Data,$ G (m.state = deletingData and d.deleted = true $\Rightarrow$ d.leaveDate = greaterThanLimit)

For a given manager and a given data file, if the manager is deleting data and the data is deleted, the data file's leave date must be greater than the limit.

Th2. $\forall m \in Managers, d \in Data,$ G (m.state = addingMedicalRecords and d.medicalRecords = added $\Rightarrow$ d.admitted = false)

For a given manager and a given data file, if the manager is adding medical records and the data file's medical records have been added, the data file's *admitted* attribute must be false.

Th3. $\forall r \in Residents, d \in Data,$ G (r.state = viewingPrivateNotes and d.privateNotes = viewed $\Rightarrow r \in d$.viewNotesACL)

For a given resident and a given data file, if the resident is viewing private notes and the data file's private notes are being viewed, the resident must be a member of the data file's *viewNotes* access control list.

The first theorem describes the requirement that a manager cannot delete a data file until a certain period of time has elapsed since the resident associated with the data file has left the facility. Instead of using specific number values for this purpose, two abstract states have been used: *lessThanLimit* and *greaterThanLimit*. The property therefore holds if the *leaveDate* attribute of the data file is in the state *greaterThanLimit*.

The second theorem ensures that a manager cannot add medical records unless the resident has not yet been admitted to the facility, given by the *admitted* attribute of the data file.

The third theorem utilises an access control list, *viewNotesACL*, which is an attribute of a data file that specifies the set of users who are permitted to view the private notes in the data file. The property states that a resident must be a member of the *viewNotesACL* if they are viewing the private notes.

The slicing tool in Integrare (see Section 6.1) was used to create the slice set and re-form it into a syntactically correct Behavior Tree. However, the removal of unnecessary reversions was completed manually as the slicing tool did not have this feature available at the time.

Slicing of the Behavior Tree begins with the construction of the dependency graph. Each operation of the system is based on user requests, specified using external input messages. Due to this, most nodes in the tree are control dependent on external input messages. Additionally, some functions are governed by further conditions. For example, many of the operations are not performed unless the given data file belongs to the given resident, specified using the selection d?name = res?, where d is a chosen data file and Res is a chosen resident. These selection nodes are data or interference-dependent on the d[name := res] node in the manager thread. This node represents the assignment of a name to a data file by the manager. Several other interference dependencies exist between selection nodes and state realisations. This includes nodes which query whether a given user belongs to a certain access control list. Such nodes are interference dependent on nodes that update the access control lists using set addition.

There are no message or synchronisation dependencies in the Behavior Tree. Furthermore, despite there being many reversions, there are no termination dependencies caused by reversions. This is because the reversions are all situated at the ends of branches of alternative branching groups. Each reversion only reverts within a single thread, so no threads are terminated. The only termination dependencies are caused by the alternative branches themselves.

Theorem 1

The next stage of slicing is to perform backwards traversals of the dependency graph starting at the criterion nodes. For the first theorem, the criterion is: $\forall m \in$ Managers and $d \in$ Data,{m.state, d.deleted, d.leaveDate},. The criterion nodes are then: d [deleted := true], d [deleted := false], m [state := deletingData], m [state := addingMedRec] and m [state := finishedAdding]. Table 7 lists the dependencies that are reached starting at the criterion nodes. The criterion nodes are italicised in the table. All of the manager state realisations only have control dependencies to external input nodes. The d [deleted := true] node is control-dependent on d ?time = aboveLimit?. This node has two dependencies: a control dependency to d ?name = r? and a termination dependency to d ?time = belowLimit?, since it is the root of an alternative branch. The d ?name = r? node leads to a chain of control dependencies and also to nodes in another branch of the manager thread, such as d [new := false]. The d [deleted := false] node is very similar. It is control-dependent on d ?time = belowLimit?. This leads to the same chain of control dependencies. Finally, the external input nodes are termination-dependent on all the other external input nodes in the manager thread, as they are all linked by an alternative branching point. The final slice set contains very few nodes and all belong to the manager thread.

Node	Dependent on	Dependency
m [state := addingMedRec]	m >>addMedicalRecords<<	cd
d [deleted := true]	d ? time = aboveLimit ?	cd
d ? time = aboveLimit ?	d ? name = r ?	cd
	d ? time = belowLimit ?	td
d ? name = r ?	d ? deleted = false ?	cd
	d [name := r]	dd
d ? deleted = false ?	m >> deleteData <<	cd
d [name := r]	d ? new = true ?	cd
d ? new = true ?	d [new := false]	dd
d [new := false]	m >> addPersonalDetails <<	cd
d [deleted := false]	d ? time = belowLimit ?	cd
d ? time = belowLimit ?	d ? name = r ?	cd
	d ? time = aboveLimit ?	td
m [state := deletingData]	m >> deleteData <<	cd
m [state := finishedAdding]	m >> addMedicalRecords <<	cd
m >> addMedicalRecords <<	m >> updateCarePlan <<	td
	m >> viewCarePlan <<	td
	m >> viewPersonalDetails <<	td
	m >> addPersonalDetails <<	td
	m >> assignDoctor <<	td
	m >> deleteData <<	td

Table 7. Dependencies Relevant for Th1 of the HIS

Due to this, the entire doctor, resident and representative threads are divergent. As with the previous case study, when the reversions are added back to the slice, it is necessary to include some of the ones in these threads in order to preserve divergent traces. In this Behavior Tree, there are no reference nodes. Using the approach in Section 3.4.3, the number of reversions can be reduced as follows. In the doctor thread, there is a reversion at the end of each alternative branch, each of which revert to the root node of the doctor thread. Each reversion is transitively control-dependent on a number of selections, mostly involving queries on the access control lists. According to the technique given in Section 3.4.3, the only controlling nodes to consider are those which have dependencies to nodes in the slice set. Fortunately, none of the selections have dependencies to nodes in the slice set, except the selection d?name = res?, which occurs in every branch. For the reversions to be considered equivalent, they must all have a control dependency to a matching d?name = res? node. In this case, every reversion does have such a dependency. The reversions are also transitively control-dependent on external input nodes. To be considered equivalent, they must all have a dependency to an external input node. Again, this is satisfied. Therefore, only one of the reversions is necessary in the slice.

Using a similar line of reasoning, there is no need to include any reversions from the resident thread, as they are all equivalent to the reversions in the doctor thread. The reversions in the resident thread all revert to the root of the resident thread. Therefore, they have a different target node than the reversions in the doctor thread. However, both sets of reversions produce divergent traces. That is, neither target is an ancestor of a node in the slice set. For that reason, the targets can be considered as equivalent to each other, so the reversions in both threads are equivalent. This demonstrates that the approach for reducing reversions is well-suited for systems containing several similar branches of behaviour.

The reversions in the representative thread are required to be included in the slice, because they have control dependencies to nodes which are dependent on nodes in the slice set. As a result, these two reversions are included into the slice.

The final slice contains only nodes from the manager and representative threads and one branch from the doctor thread. The number of transitions and number of program counters (which indicates the number of branches) are significantly less than the original model, as shown in the table below. Each set in the Behavior Tree can be initialised to any value. The table lists the number of transitions and program counters for the model and slice when the sets only contain one user each.

	No. of Transitions	No. of PC's
Original	122	37
Slice	42	14

Table 8. Original Model vs. Slice for Th1 of the HIS

Various combinations of sets were investigated, to determine the impact slicing had on each type of user. Initially, all sets contained only one user each. In the subsequent experiments, one set contained two users, while the others still contained only one. In the 6th to 8th experiments, two of the sets contained two users each, while the other sets contained one each. Finally, in the last experiment, all sets contained two users each. The verification times for these experiments are given in Table 9. The original model was not verifiable. Even using only one user per set, the model checker was not able to provide a result for Theorem 1 in 24 hours, at which point it was terminated manually. In comparison, model checking Theorem 1 on the slice was accomplished extremely fast, taking less than two seconds. Increasing the size of the sets did not significantly increase the verification time. The set which had the greatest influence on verification time was the manager set. When the manager set was limited to one user, the verification time remained under 10 seconds. This was regardless of the sizes of the other sets, as demonstrated by Exp. 7, in which the doctor and resident sets were increased to two users each while the manager set contained only one user. With two managers, the time increased to one minute. Using two doctors with two managers did not have any impact. This is consistent with the structure of the slice, since the doctor thread in the slice contains only one branch and would therefore not significantly impact on the verification time. Most of the nodes in the slice are in the manager thread, so increasing the number of managers had the greatest impact. Two managers and two residents took nearly 8 minutes and two users in each set took approximately one hour. This demonstrates that the verification time still increases as the number of users in the sets are increased, but slicing gives a significant improvement over the original model which could not be verified at all.

Exp.	1 1 in each set.	2 2 Managers, 1 all others.	3 2 Doctors, 1 all others.	4 2 Residents 1 all others.	5 2 Represen. 1 all others.
Original	> 24 hrs	-	-	-	-
Slice	1.68s	64.05s	3.94s	6.09s	7.85s
Exp.	6 2 Doctors, 2 Managers, 1 all others.	7 2 Doctors, 2 Residents, 1 all others.	8 2 Managers, 2 Residents, 1 all others.	9 2 in each set.	
Original	-	-	-	-	
Slice	56.62s	9.11s	7.95 mins	64.75 mins	

Table 9. Verification Times for Th1.

Theorem 2

The criterion for Theorem 2 is: ∀ m ∈ Managers and d ∈ Data, {m.state, d.admitted, d.medicalRecords}. Unlike the previous theorem, there are criterion nodes from each of the threads. In the manager thread, there are nodes that modify the state of the manager and the *medicalRecords* attribute of data: m[state := deletingData], m[state := addingMedRec], m[state := finishedAdding] and d[medicalRecords := added]. In the doctor thread, there are nodes modifying the *medicalRecords* attribute:

d[medicalRecords := added] and d[medicalRecords := viewed]. Finally, there are the nodes d[medicalRecords := viewed] in the resident thread and Data [admitted := true] in the representative thread. Note that a corresponding initialisation text file, as mentioned in Section 2.3.3, sets the *admitted* attribute to *false* and the *medicalRecords* attribute to *notAdded* initially, for each d ∈ Data. These initial values are therefore used in the slice as well, as they correspond to the initialisation section of the underlying transition system.

The backwards traversals of the dependency graph starting at each of the manager states collects the same nodes as for the previous theorem, i.e. some external input nodes and selections. Some of the relevant dependencies are given in Table 10.

Node	**Dependent on**	**Dependency**
m [state := deletingData]	m >> deleteData <<	cd
m [state := addingMedRec]	m >>addMedicalRecords<<	cd
d [medicalRecords := added] (Manager thread)	d ? name = r ?	cd
d ? name = r ?	d ? deleted = false ?	cd
	d [name := r]	dd
d ? deleted = false ?	m >> deleteData <<	cd
d [name := r]	d ? new = true ?	cd
d ? new = true ?	d [new := false]	dd
d [new := false]	m >> addPersonalDetails <<	cd
d [admitted := true]	d ? admitted = false ?	cd
d ? admitted = false ?	d ? representative = person ?	cd
d ? representative = person ?	d [representative := person]	id
d [medicalRecords := viewed] (Doctor thread)	d ? res : viewMedicalACL ?	cd
d ? res : viewMedicalACL ?	d ? doc : viewMedicalACL ?	cd
d ? doc : viewMedicalACL ?	d ? name = res ?	cd
d ? name = res ?	d [name := r]	id

Table 10. Dependencies Relevant for Th2 of the HIS

The d[medicalRecords := added] node in the manager thread is control-dependent on d?name = r?, which results in the same nodes from the manager thread being included into the slice as for the first theorem. The d[admitted := true] node in the representative thread is transitively control-dependent on d [representative := person], which is in turn interference dependent on the corresponding state realisation in the manager thread. The nodes in the doctor and resident threads have dependencies to the access control lists *viewMedicalACL* and *addMedicalACL*, which are both updated by nodes in the doctor thread. For this reason, a large portion of the doctor thread must be included into the slice. Only a few reversions could be eliminated, as in this case most of the reversions do not produce divergent behaviour. The final slice is therefore almost as large as the original model, as shown by Table 11.

	No. of Transitions	No. of PC's
Original	122	37
Slice	116	27

Table 11. Original Model vs. Slice for Th2 of the HIS

As was done for the first theorem, various combinations of the sets were used, starting with one user in each set. The verification times for each experiment are given in Table 12. Using the original model with one user per set, the model checker was unable to provide a result in 24 hours. The slice

was extremely fast in comparison. It was able to be verified in 68 seconds using one user per set. The largest part of the slice is the doctor thread, so increasing the number of doctors had the greatest impact on verification time. When the doctor set was limited to one user, using two representatives increased the verification time to five minutes (Exp. 5), while using either two managers or two residents increased the time to seventeen minutes (Exp. 2 and 4). Increasing the doctor set to two users caused a larger increase in verification time, to 1.5 hours (Exp. 3). The final experiment combined two doctors with two managers. This increased the time to almost 8 hours. The largest parts of the slice are the doctor and manager threads, so this is the largest combination of two sets with two users each. Other combinations were not attempted as they would not provide any further insights. Since all other combinations of two sets with two users result in slices that are smaller than the slice in Exp. 6, they would be verified in faster time.

Exp.	1 1 in each set.	2 2 Managers, 1 all others.	3 2 Doctors, 1 all others.	4 2 Residents 1 all others.	5 2 Represen. 1 all others.
Original	> 24 hrs	-	-	-	-
Slice	68.53s	17.22 mins	1hr 29.9mins	17.41 mins	5.21 mins

Exp.	6 2 Doctors, 2 Managers, 1 all others.
Original	-
Slice	7hrs 52mins

Table 12. Verification Times for Th2.

For Theorem 3, the criterion is ∀r ∈ Residents and d ∈ Data, {r.state, d.privateNotes, d.viewNotesACL}. There are criterion nodes in the resident and doctor threads. In the resident thread, the nodes r[state := viewingPrivateNotes] and d[privateNotes := viewed] both modify variables in the criterion. In the doctor thread, the criterion nodes are: d[privateNotes := viewed], d[privateNotes := added], d[viewNotesACL := viewNotesACL + {res}] and d[viewNotesACL := viewNotesACL + {doc}]. The latter two are nodes which update the *viewNotesACL* list. As was done for the previous theorem, the initial values of the Resident's *state* attribute and Data's *privateNotes* and *viewNotesACL* attributes, given by the initialisation text file, will also remain in the slice. Some of the relevant dependencies are given in Table 13.

The node r[state := viewingPrivateNotes] only has a control dependency to an external input node. All the nodes involving the *privateNotes* attribute of Data have control dependencies to selections querying the *viewNotesACL* and *addNotesACL* access control lists. These lead to interference dependencies to state realisations involving these access control lists. These are all control-dependent on external input nodes and selections of the form d?assignedDoctor = doc?, which are in turn interference-dependent on the d[assignedDoctor := doc] node in the manager thread. Every criterion node is also transitively control-dependent on selections of the form d?name = r?, which are also interference-dependent on a node in the manager thread. The final slice contains nodes from all four threads. However, the doctor thread is not as large as for Theorem 2. By using the approach in Section 3.4.3, several reversions can be removed, resulting in fewer branches in the doctor thread. The size of the slice compared to the original model is given in Table 14.

The verification times for Theorem 3 are given in Table 15. Four combinations were attempted. The original model could not be verified in 24 hours, even with only one user per set. The slice took only 48 seconds for this case. When the number of managers was increased to two, the verification time increased to 2 hours. Increasing the number of doctors to two each resulted in a verification times of 50 mins. With two representatives but one of all other types of users, the verification time was 16 minutes. This is consistent with the layout of the slice, since the representative thread contains the fewest number of nodes and branches.

Node	Dependent on	Depend-ency
r [state := viewingPrivateNotes]	m >> viewPrivateNotes <<	cd
d [privateNotes := viewed] (Resident thread)	d ? r : viewNotesACL ?	cd
d ? r : viewNotesACL ?	d ? name = r ?	cd
	d [viewNotesACL := viewNotesACL + {r}]	id
	d [viewNotesACL := viewNotesACL + {doc}]	id
d ? name = r ?	m >> viewPrivateNotes <<	cd
d [privateNotes := viewed] (Doctor thread)	d ? r : viewNotesACL ? (Doc thread)	cd
d ? r : viewNotesACL ? (Doctor thread)	d ? doc : viewNotesACL ?	cd
d ? doc : viewNotesACL ?	d ? name = r ? (2)	cd
	d [viewNotesACL := viewNotesACL + {r}]	dd
	d [viewNotesACL := viewNotesACL + {doc}]	dd
d ? name = r ? (2)	d >> viewPrivateNotes <<	cd
d [viewNotesACL := viewNotesACL + {r}]	d?assignedDoctor = doc?	cd
d?assignedDoctor = doc?	d ? name = r ? (3)	cd
d ? name = r ? (3)	d >> assignResToViewNotesACL <<	cd
d [viewNotesACL := viewNotesACL + {doc}]	d?assignedDoctor = doc? (2)	cd
d?assignedDoctor = doc? (2)	d ? name = r ? (4)	cd
d ? name = r ? (4)	d >> assignDocToViewNotesACL <<	cd

Table 13. Some of the Dependencies Relevant for Th3 of the HIS

	No. of Transitions	No. of PC's
Original	122	37
Slice	114	26

Table 14. Original Model vs. Slice for Th3 of the HIS

Exp.	1 1 in each set.	2 2 Managers, 1 all others.	3 2 Doctors, 1 all others.	4 2 Represen. 1 all others.
Original	> 24 hrs	-	-	-
Slice	47.74s	2hrs 16.4mins	50.37 mins	15.8 mins

Table 15. Verification Times for Th3.

For all three theorems, the original model could not be verified even using only one user per set. The slices with one user per set produced results very quickly. As the sets were increased, the verification times increased, which demonstrates that slicing does not prevent the state explosion problem. However, it increases the range of cases which can be model checked. Slicing allowed results to be obtained for sets containing one or two users. For most systems, this would be sufficient to identify any problems with the design.

The two case studies presented in this chapter demonstrate the benefits of slicing Behavior Trees prior to model checking. As was seen by the two case studies, the reductions obtained are dependent on the model and the property. In some cases greater reductions in verification time are produced than for others. Despite this, significant reductions were obtained for all cases and slicing enabled models which were not previously verifiable to be model checked. The next chapter concludes the thesis.

7 CONCLUSION

7.1 Contributions

The primary contribution of this thesis is a technique for reducing Behavior Tree specifications using slicing) in order to alleviate the state explosion problem when model checking large systems. This slicing technique has been shown to preserve formulas expressed in the logic CTL^*_{-X}, by relating the slice, given by a function $slice_\varphi$, to the original model using branching bisimulation with explicit divergence. By evaluation on case studies, the slicing technique has been demonstrated to have the potential to reduce the verification time of large models dramatically, although the extent of reduction is dependent on several factors, such as the formula to be verified and the level of dependencies between the nodes of the Behavior Tree.

In addition, an optimisation technique has been presented for reducing slices further by removing infeasible paths. The approach has been demonstrated to remove more nodes than other previous related approaches. This technique, given by the function $slice\_inf_\varphi$, has also been shown to preserve CTL^*_{-X} formulas.

The final contribution is a novel method for producing slices that can preserve full CTL^* formulas, including formulas containing the X operator. No other slicing technique in the literature is able to handle such formulas. This technique, given by the function $slice\_next_\varphi$, has been shown to be correct by the use of a new type of branching bisimulation, termed *next-preserving branching bisimulation*, which has been shown to preserve full CTL^*.

For a transition system B corresponding to a BT control flow graph and a formula φ, the slicing functions can be composed in the following ways, where $\circ$ denotes the composition operator:

$slice\_inf_\varphi \circ slice_\varphi(B)$,
$slice\_next_\varphi \circ slice_\varphi(B)$ or
$slice\_next_\varphi \circ slice\_inf_\varphi \circ slice_\varphi(B)$.

This allows the user to select the most suitable techniques for their purpose. For example, a user may not require the use of the X operator and therefore does not need to use the $slice\_next_\varphi$ function, or a user may feel that the slice is small enough without the need for removing infeasible paths.

The slicing techniques presented in this thesis are an essential addition to the Behavior Engineering methodology, as they allow the verification of large Behavior Trees to be possible. As well as this, the results of this thesis are of benefit to the formal methods community in general. Concepts such as the infeasible path reduction and next-preserving slicing can be applied when slicing any programming or specification language. Additionally, some of the methods for producing Behavior Tree slices can be utilised for other similar languages. Furthermore, the concept of next-preserving branching bisimulation and the results about property preservation of full CTL^* that are presented in this thesis are a valuable theoretical contribution in general.

7.2 Future Work

The directions for future work include further evaluation of the slicing techniques on case studies, extending the tool support and exploration of further theoretical aspects of the approaches.

Case studies are necessary to evaluate the infeasible path approach presented in Chapter 4, in order to investigate how useful the reductions are in practice. Similarly, the technique given in Chapter 5 for preserving properties containing the X operator needs to be evaluated on case studies. This is especially important in order to determine whether the slices remain small enough to be beneficial even when the extra stuttering nodes are added back to the slice.

Evaluation of these techniques on case studies is only possible if the slicing tool can perform all of the necessary features. Therefore, another area of future work is to implement the algorithms of Chapters 4 and 5. Furthermore, it is planned that the slicing tool will be linked with the other Behavior Tree editor, TextBE (Myers, 2011), which is a freely available tool and is therefore more accessible to users than Integrare. Ideally, the slicing tool would display the resulting slice visually to the user and provide statistics about its size and structure. Additionally, users may find it useful to view the intermediate control flow graphs and program dependence graphs.

Another related avenue for future work is to devise a polynomial time algorithm for the infeasible path technique. Preliminary investigations into this suggest that a polynomial algorithm is possible, by storing the necessary information in such a way that no node needs to be explored more than once.

The technique for adding extra stuttering nodes to preserve properties with the X operator could be further optimised by considering the structure of the formula in more detail. There are cases where the proposed approach adds more stuttering nodes than is necessary. For example, if the formula does not contain the E operator after an X operator, it is not necessary to add stuttering nodes before branching locations. Furthermore, an approach could be developed to add certain numbers of stuttering nodes before some observable nodes, while adding different numbers before other observable nodes, based on the specific atomic propositions mentioned in the formula.

Finally, another direction for future research is to investigate other applications where next-preserving branching bisimulation could be useful, i.e. applications which cannot maintain a strong bisimulation but could benefit from the use of the next step operator.

REFERENCES

Amtoft, T. (2008). Slicing for modern program structures: a theory for eliminating irrelevant loops. *Information Processing Letters, 106*(2), 45-51.

Baier, C., & Katoen, J. P. (2008). *Principles of Model Checking*, MIT Press.

Behavior Engineering. Viewed 25/10/2011, www.behaviorengineering.org

Behavior Tree Group. (2007). Behavior Tree Notation v1.0. Retrieved from www.behaviorengineering.org on 25/10/2011

Binkley, D., Danicic, S., Gyimóthy, T., Harman, M., Kiss, Á., & Korel, B. (2006). A Formalisation of the Relationship Between Forms of Program Slicing. *Science of Computer Programming, 62*(3), 228-252.

Bloom, B. (1995). Structural Operational Semantics for Weak Bisimulations. *Theoretical Computer Science, 146*(1-2), 25-68.

Bordini, R. H., Fisher, M., Wooldridge, M., & Visser, W. (2009). Property-based Slicing for Agent Verification. *Journal of Logic and Computation, 19*(6), 1385-1425.

Brückner, I. (2007). Slicing Concurrent Real-Time Specifications for Verifications. In: *Proceedings of the 6th International Conference on Integrated Formal Methods (IFM 2007)*, (pp. 54-74): Vol. 4591 of Lecture Notes in Computer Science, Springer.

Brückner, I., & Wehrheim, H. (2005a). Slicing an Integrated Formal Method for Verification. In: *Proceedings of the 7th International Conference on Formal Engineering Methods (ICFEM 2005)*, (pp. 360-374): Vol. 3785 of Lecture Notes in Computer Science, Springer.

Brückner, I., & Wehrheim, H. (2005b). Slicing Object-Z Specifications for Verification. In: *Proceedings of the 4th International Conference of Z and B Users (ZB 2005)*, (pp. 149-157): Vol. 3455 of Lecture Notes in Computer Science, Springer.

Canfora, G., Cimitile, A., & De Lucia, A. (1998). Conditioned Program Slicing. *Information and Software Technology, 40*(11-12), 595-607.

Chen, F., & Roşu, G. (2006). Parametric and Termination-Sensitive Control Dependence. In: K. Yi (Ed.), *Proceedings of the 13th International Symposium on Static Analysis (SAS 2006)*, (pp. 387-404): Vol. 4134 of Lecture Notes in Computer Science, Springer.

Cheng, J. (1993). Slicing Concurrent Programs - A Graph Theoretical Approach. In: P. A. Fritzson (Ed.), *Proceedings of the 1st International Workshop on Automated Algorithmic Debugging (AADEBUG '93)*,(pp. 223-240): Vol. 749 of Lecture Notes in Computer Science, Springer.

Clarke, E.M., & Emerson, E.A. (1982). Design and Synthesis of Synchronisation Skeletons Using Branching Time Temporal Logic. In: *Proceedings of Logics of Programs: Workshop*, (pp. 52-71): Vol. 131 of Lecture Notes in Computer Science, Springer.

Clarke, E. M. , Emerson, E. A., & Sistla, A. P. (1986). Automatic Verification of Finite State Concurrent Systems using Temporal Logic Specifications. *ACM Transactions on Programming Languages and Systems, 8*(2), 244-263.

Clarke, E., Grumberg, O., Jha, S., Lu, Y., & Veith, H. (2001). Progress on the State Explosion Problem. In: *Informatics. 10 Years Back. 10 Years Ahead*, (pp. 176-194): Vol. 2000 of Lecture Notes in Computer Science, Springer.

Colvin, R. J., & Hayes, I. J. (2011). A Semantics for Behavior Trees Using CSP with Specification Commands. *Science of Computer Programming, 76*(10), 891-914.

Dams, D. (1996). *Abstract Interpretation and Partition Refinement for Model Checking*. PhD Thesis, Institute for Programming Research and Algorithmics, Eindhoven University of Technology.

Dams, D., Gerth, R., & Grumberg, O. (1997). Abstract Interpretation of Reactive Systems. *ACM Transactions on Programming Languages and Systems, 19*(2), 253-291.

Danicic, S., Harman, M., & Sivagurunathan, Y. (1995). A Parallel Algorithm for Static Program Slicing. *Information Processing Letters, 56*, 307-313.

de Moura, L., Owre, S., Rueß, H., Rushby, J., Shankar, N., Sorea, M., & Tiwari, A. (2004). SAL 2. In: R. Alur & D. Peled (Eds.), *Proceedings of the 16th International Conference on Computer-Aided Verification (CAV2004)*, (pp. 251-254): Vol. 3114 of Lecture Notes in Computer Science, Springer.

de Nicola, R., & Vaandrager, F. (1995). Three Logics for Branching Bisimulation. *Journal of the Association for Computing Machinery, 42*(2), 458-487.

Dromey, R. G. (2003). From requirements to design: formalizing the key steps. In: *Proceedings of International Conference on Software Engineering and Formal Methods (SEFM 2003)*, (pp. 2-11), IEEE.

Dromey, R. G. (2005). Genetic design: Amplifying our ability to deal with requirements complexity. In: *Proceedings of Models, Transformations and Tools*, (pp. 95-108): Vol. 3466 of Lecture Notes in Computer Science, Springer.

Dwyer, M. B., Hatcliff, J., Hoosier, M., Ranganath, V., Robby, & Wallentine, T. (2006). Evaluating the Effectiveness of Slicing for Model Reduction of Concurrent Object-Oriented Programs. In: *Proceedings of the 12th International Conference on Tools and Algorithms for the Analysis and Construction of Systems (TACAS 2006)*, (pp. 73-89): Vol. 3920 of Lecture Notes in Computer Science, Springer.

Evered, M., & Bögeholz, S. (2004). A Case Study in Access Control Requirements for a Health Information System. In: *Proceedings of the 2nd Workshop on Australasian Information Security, Data Mining and Web Intelligence, and Software Internationalisation*, (pp. 53-61), Vol. 32, Australian Computer Society.

Ganesh, V., Saidi, H., & Shankar, N. (1999). *Slicing SAL*: CSL Technical Report, Computer Science Laboratory, SRI International.

Grunske, L., Winter, K., & Yatapanage, N. (2008). Defining the Syntax of Visual Languages with Advanced Graph Grammars - A Case Study Using Behavior Trees. *Journal of Visual Languages and Computing, 19*(3), 343-379.

Grunske, L., Winter, K., Yatapanage, N., Zafar, S., & Lindsay, P. A. (2011). Experience with Fault Injection Experiments for FMEA. *Journal of Software Practice and Experience, 41*(11), 1233-1258.

Hassine, J., Dssouli, R., & Rilling, J. (2005). Applying Reduction Techniques to Software Functional Requirement Specifications. In: *Proceedings of the 4th International SDL and MSC Workshop on*

System Analysis and Modelling (SAM 2004), (pp. 138-153): Vol. 3319 of Lecture Notes in Computer Science, Springer.

Hatcliff, J., Corbett, J., Dwyer, M., Sokolowski, S., & Zheng, H. (1999). A Formal Study of Slicing for Multi-threaded Programs Using JVM Concurrency Primitives. In: A. Cortesi & G. File (Eds.), *Proceedings of the 6th International Symposium on Static Analysis (SAS '99)*, (pp. 1-18) Vol. 1694 of Lecture Notes in Computer Science, Springer.

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

Heimdahl, M. P. E., & Whalen, M. W. (1997). Reduction and Slicing of Heirarchical State Machines. In: M. Jazayeri & H. Schauer (Eds.), *Proceedings of the 6th European Software Engineering Conference Held Jointly with the 5th ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'97)*, (pp. 450-467) Vol. 1301 of Lecture Notes in Computer Science, Springer.

Korel, B., & Laski, J. (1988).Dynamic program slicing. *Information Processing Letters, 29*(3), 155-163.

Korel, B., Singh, I., Tahat, L., & Vaysburg, B. (2003). Slicing of state-based models. In: *Proceedings of the International Conference on Software Maintenance (ICSM 2003)*, (pp. 34-43), IEEE.

Krinke, J. (1998). Static slicing of threaded programs. *ACM SIGPLAN Notices, 33*(7), 35-42.

Krinke, J. (2003). *Advanced Slicing of Sequential and Concurrent Programs.* PhD Thesis, Fakultät Für Mathematik und Informatik, Universität Passau.

Kučera, A., & Strejček, J. (2005). The Stuttering Principle Revisited. *Acta Informatica, 41*(7/8), 415-434.

Kumar, S., & Horwitz, S. (2002). Better Slicing of Programs with Jumps and Switches. In: *Fundamental Approaches to Software Engineering*, (pp. 371-388): Vol. 2306 of Lecture Notes in Computer Science, Springer.

Labbé, S., & Gallois, J.-P. (2008). Slicing Communicating Automata Specifications: Polynomial Algorithms for Model Reduction. *Formal Aspects of Computing, 20*(6), 563-595.

Labbé, S., Gallois, J.-P., & Pouzet, M. (2007). Slicing communicating automata specifications for efficient model reduction. In: *Proceedings of the 2007 Australian Software Engineering Conference (ASWEC'07)*, (pp. 191-200), IEEE.

Lamport, L. (1983). What Good is Temporal Logic? In: R. E. A. Mason (Ed.), *Information Processing 83*, (pp. 657-668), Elsevier Science Publishers B.V. (North-Holland).

Lano, K., & Kolahdouz-Rahimi, S. (2010). Slicing of UML Models Using Model Transformations. In: D. C. Petriu, N. Rouquette & O. Haugen (Eds.), *Proceedings of the ACM/IEEE 13th International Conference on Model Driven Engineering Languages and Systems (MODELS 2010 Part II)*, (pp. 228-242): Vol. 6395 of Lecture Notes in Computer Science, Springer.

Larsen, K. G., Pettersson, P., & Yi, W. (1997). Uppaal in a Nutshell. *International Journal on Software Tools for Technology Transfer, 1*(1-2), 134-152.

Luangsodsai, A., & Fox, C. (2010). Concurrent Statechart Slicing. In: *Proceedings of the 2nd Computer Science and Electronic Engineering Conference (CEEC), Colchester, UK*, (pp. 1-7).

McMillan, K. (1992). *Symbolic Model Checking. An Approach to the State Explosion Problem.* PhD Thesis, Carnegie Mellon University.

Millett, L. I., & Teitelbaum, T. (2000). Issues in slicing PROMELA and its applications to model checking, protocol understanding and simulation. *International Journal on Software Tools for Technology Transfer, 2*(4), 343-349.

Müller-Olm, M., & Seidl, H. (2001). On Optimal Slicing of Parallel Programs. In: *Proceedings of the thirty-third annual ACM symposium on Theory of Computing, Hersonissos, Greece,* (pp. 647-656).

Myers, T. (2010). *The Foundations for a Scaleable Methodology for Systems Design.* PhD Thesis, School of Information and Communication Technology, Griffith University, Queensland, Australia.

Myers, T. (2011) textbe. Textual Editor for Behavior Engineering. Viewed 22nd April, 2012, http://code.google.com/p/textbe/

Nanda, M. G., & Ramesh, S. (2000). Slicing concurrent programs. *ACM SIGSOFT Software Engineering Notes, 25*(5), 180-190.

Nanda, M. G., & Ramesh, S. (2006). Interprocedural slicing of multithreaded programs with applications to Java. *ACM Transactions on Programming Languages and Systems, 28*(6), 1088-1144.

Oda, T., & Araki, K. (1993). Specification Slicing in Formal Methods of Software Development. In: *Proceedings of the 17th Annual International Computer Software and Applications Conference (COMSAC 93),* (pp. 313-319).

Odenbrett, M., Nguyen, V. Y., & Noll, T. (2010). Slicing AADL specifications for model checking. In: C. Muñoz (Ed.), *Proceedings of the Second NASA Formal Methods Symposium,* (pp. 217-221).

Ottenstein, K. J., & Ottenstein, L. M. (1984). The program dependence graph in a software development environment. *SIGSOFT Software Engineering Notes, 9*(3), 177-184.

Peled, D. (1998). Ten Years of Partial Order Reduction. In: *Proceedings of the 10th International Conference on Computer Aided Verification,* (pp. 17-28): Vol. 1427 of Lecture Notes in Computer Science, Springer.

Pnueli, A. (1977). The temporal logic of programs. In: *Proceedings of the 18th IEEE Symposium on Foundations of Computer Science,* (pp. 46-67), IEEE.

Podgurski, A., & Clarke, L. A. (1990). A Formal Model of Program Dependences and Its Implications for Software Testing, Debugging and Maintenance. *IEEE Transactions on Software Engineering, 16*(9), 965-979.

Quielle, J., & Sifakis, J. (1982). Specification and Verification of Concurrent Systems in CESAR. In: *International Symposium on Programming: 5th Colloquium,* (pp. 337-351): Vol. 137 of Lecture Notes in Computer Science, Springer.

Rakow, A. (2008). Slicing Petri Nets with an Application to Workflow Verification. In: *Proceedings of the 34th Conference on Current Trends in Theorey and Practice of Computer Science (SOFSEM 2008),* (pp. 436-447): Vol. 4910 of Lecture Notes in Computer Science, Springer.

Ranganath, V. P., Amtoft, T., Banerjee, A., Dwyer, M., & Hatcliff, J. (2007). A new foundation for control-dependence and slicing for modern program stuctures. *ACM Transactions on Programming Languages and Systems, 29*(5).

Ranganath, V. P., & Hatcliff, J. (2004). Pruning Interference and Ready Dependence for Slicing Concurrent Java Programs. In: E. Duesterwald (Ed.), *Proceedings of the 13th International Conference on Compiler Construction (CC 2004)*, (pp. 39-56): Vol. 2985 of Lecture Notes in Computer Science, Springer.

Representing Complex Systems (2008). *The Magazine of Engineers Australia, 80*(9), 38.

Reps, T., Horwitz, S., Sagiv, M., & Rosay, G. (1994). Speeding Up Slicing. *ACM SIGSOFT Software Engineering Notes, 19*(5), 11-20.

Sabouri, H., & Sirjani, M. (2010). Slicing-based Reductions for Rebeca. *Electonic Notes in Theoretical Computer Science, 260*, 209-224.

Schaefer, I., & Poetzsch-Heffter, A. (2008). Slicing for Model Reduction in Adaptive Embedded Systems Development. In: *Proceedings of the Workshop on Software Engineering for Adaptive and Self-managing Systems (SEAMS'08)*, (pp. 25-32).

Silva, J. (2011). A Vocabulary of Program Slicing-Based Techniques. *ACM Computing Surveys, To Appear.* (Retrieved from www.dsic.upv.es/~jsilva/papers/Vocabulary.pdf)

Silva, J., Leuschel, M., Tamarit, S., Oliver, J., & Llorens, M. (2008). Static slicing of CSP specifications. In: M. Hanus (Ed.), *Pre-Proceedings of LOPSTR 2008, The 18th International Symposium on Logic-Based Program Synthesis and Transformation, July 2008*, (pp. 141-150).

ter Beek, M. H., Fantechi, A., Gnesi, S., & Mazzanti, F. (2011). A State/Event Based Model Checking Approach for the Analysis of Abstract System Properties. *Science of Computer Programming, 76*, 119-135.

Thrane, C., & Sorensen, U. (2008). Slicing for UPPAAL. In: *Student Paper, 2008 Annual IEEE Conference*, (pp. 1-5), IEEE.

Tip, F. (1995). A survey of program slicing techniques. *Journal of Programming Languages, 3*(3), 121-189.

van Glabbeek, R., Luttik, B., & Trčka, N. (2009a). Branching Bisimilarity with Explicit Divergence. *Fundamenta Informaticae, 93*(4), 371-392.

van Glabbeek, R., Luttik, B., & Trčka, N. (2009b). Computation Tree Logic with Deadlock Detection. *Logical Methods in Computer Science, 5*(4:5), 1-24.

van Glabbeek, R. J., & Weijland, W. P. (1996). Branching Time and Abstraction in Bisimulation Semantics. *Journal of the Association for Computing Machinery, 43*(3), 555-600.

van Langenhove, S., & Hoogewijs, A. (2006). Verifying Sliced Hierarchical Statecharts with SVtL. In: *The 2006 Federated Logic Conference Verify '06: Verification Workshop, Seattle, Washington, USA*, (pp. 42-52).

Vasudevan, S., Emerson, E. A., & Abraham, J. A. (2005). Efficient Model Checking of Hardware Using Conditioned Slicing. *Electonic Notes in Theoretical Computer Science, 128*, 279-294.

Wasserrab, D., Lohner, D., & Snelting, G. (2009). On PDG-based Noninterference and Its Modular Proof. In: *Proceedings of the ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security (PLAS'09)*, (pp. 31-44).

Weiser, M. (1981). Program slicing. In: *Proceedings of the 5th International Conference on Software Engineering (ICSE'81)*, (pp. 439-449), IEEE.

Weiser, M. (1984). Program slicing. *IEEE Transactions on Software Engineering, SE-10*(4), 352-357.

Wen, L., Lin, K., Colvin, R., Seagrott, J., Yatapanage, N., & Dromey, R. G. (2007). "Integrare" - A Collaborative Environment for Behavior-Oriented Design. In: *Proceedings of the 4th International Conference on Cooperative Design, Visualisation and Engineering (CDVE 2007)*, (pp. 122-131): Vol. 4674 of Lecture Notes in Computer Science, Springer.

Wu, F., & Yi, T. (2004). Slicing Z Specifications. *ACM SIGPLAN, 39*(8).

Xu, B., Qian, J., Zhang, X., Wu, Z., & Chen, L. (2005). A brief survey of program slicing. *ACM SIGSOFT Software Engineering Notes, 30*(2), 1-36.

Yatapanage, N., Winter, K., & Zafar, S. (2010). Slicing Behavior Tree models for verification. In: C. S. Calude & V. Sassone (Eds.), *Proceedings of the 6th IFIP International Conference on Theoretical Computer Science (TCS2010)*, (pp. 125-139): Vol. 323 of IFIP Advances in Information and Communication Technology, Springer.

Zafar, S. (2008). *Integration of Access Control Requirements into System Specifications.* PhD Thesis, School of Information and Communication Technology, Griffith University, Queensland, Australia.

Zafar, S., Colvin, R., Winter, K., Yatapanage, N., & Dromey, R. G. (2007). Early Validation and Verification of a Distributed Role-Based Access Control Model. In: *Proceedings of the 14th Asia-Pacific Software Engineering Conference (APSEC 2007)*, (pp. 430-437), IEEE.

Zhao, J. (1999). Slicing concurrent Java programs. In: *Proceedings of the 7th International Workshop on Program Comprehension*, (pp. 126-133), IEEE.