MASTER

Diagnostics for model checking

Linssen, C.A.P.

Award date:
2011

Diagnostics for Model Checking

Charl A.P. Linssen

January 2011

Table of Contents

0 Abstract

Model checking is the process of provably verifying claims about computer processes. These processes are modelled as finite transition systems, while claims about them are expressed in a temporal logic. Several temporal logics are used in practice, such as CTL and the mu-calculus. The modal mu-calculus in particular allows highly complex properties to be formulated, and will be the focus of this work. A model check problem is the combination of a model and a claim about the model, and can be solved using an (automated) model checker, which delivers a simple answer, namely a Boolean constant indicating whether the claim holds over the model or whether it does not.

In practice, models can be very large, with the number of states exceeding astronomical figures. When a model checker delivers an unexpected result, for example that a deadlock exists in the system, it does so by returning the value false. This answer in and of itself does little to help the user in debugging the model (or sometimes the specification). It provides no leads to indicate in which section of the model the fault lies, or how this fault is related to the (potentially complex) claim. Commonly, the user will have to engage in an iterative process to pinpoint the source of the fault, for example by re-running the model check on a limited section of the model or by simplifying the claim. Due to the potentially very large size of the model, this method leads to severe time penalties, as solving a model check problem can easily take hours or days, even on very powerful computers.

To aid the user in tracing the source of an unexpected outcome, various forms of diagnostics may be used. These may be implemented in model checkers, as an improvement over the simple *yes/no* answer. However, some forms of diagnostic are more useful than others. The goal of a diagnostic is usually to provide the (human) user with insight into the source of the fault. Some approaches are more successful at this than others; in particular, the user should not have to be concerned with the internal data structures used by the model checker, but would like a diagnostic in terms of the (familiar) model and mu-calculus claim. This is a central requirement for us, as it is essential to the user-friendliness of a diagnostic.

Several classes of diagnostic will be evaluated for their practicality (e.g. in terms of efficient generation) and most importantly for their contribution to the insight of the user. A number of different techniques exist for solving the model check problem; each gives rise to its own class of diagnostic, but some of these classes are of a more universal nature, such as a path through the transition system that exemplifies the failure.

Because each model check technique considered here verifies claims expressed in the same temporal logic (the mu-calculus) over the same kind of model (transition systems), there is a high degree of correspondence between the classes of diagnostic associated with each method. These correspondences will be explored, and are then used as the basis for a new class of diagnostic developed in this work. This new class will avoid many of the shortcomings of the other diagnostics, and was developed with three requirements in mind: most impor-

tantly, being intelligible by the end user; secondly, being able to explain any fault regardless of the details of the model or claim; and finally being efficiently generable due to the potentially large model size.

Some issues arising on the sidelines are also considered, such as pointing out when the user has formulated a specification that holds trivially or is a tautology, as well as the presentation of a diagnostic. Given the importance of user-friendliness in this work, the presentation and visualisation of a diagnostic is essential in permitting quick interpretation and giving the user the ability to navigate a large diagnostic by targeted exploration or search. We will also introduce the possibility to extract a more simple form of diagnostic, such a failure path, from the more complete (and thus more complex) class of diagnostic developed here—again in the interest of empowering the user with methods for the quick, simple and intuitive analysis of a diagnostic.

1 Labelled Transition Systems

In this work, we shall be concerned with making statements about the behaviour of computational processes. All possible behaviour of a process is captured in a data structure called a *transition system*. A transition system consists of a set of states and a set of transitions, which can be visualised as a directed graph with the states as vertices and the transitions as (labelled) edges. At any point in time, the process is said to be "in" a certain state, where it can perform one of the actions possible in that state, leading to the same or another state. Only finite (nondeterministic) transition systems will be considered here.

Definition 1.1 (Labelled Transition System). *A* labelled transition system *or* LTS *is a tuple* $\mathcal{T} = \langle S,\, s_0,\, \mathcal{L},\, \delta \rangle$*, where* S *is a finite set of states,* $s_0 \in S$ *is the initial state,* $\mathcal{L}$ *is the set of labels and* $\delta \subseteq S \times \mathcal{L} \times S$ *is the transition relation. The notation* $s \xrightarrow{a} s'$ *will be used for* $\langle s,\, a,\, s' \rangle \in \delta$*.* □

A model is defined in the context of a set $\mathcal{Q}$ of atomic propositions, which includes true and false. In each state, a number of propositions may hold. The valuation function $\mathcal{V}$ maps a proposition to the set of states in which it holds.

Definition 1.2 (Valuation Function). *Given a labelled transition system* $\mathcal{T} = \langle S,\, s_0,\, \mathcal{L},\, \delta \rangle$ *and set of atomic propositional variables* $\mathcal{Q}$*, the* valuation function $\mathcal{V} \colon \mathcal{Q} \mapsto \mathcal{P}(S)$ *assigns a set of states to every atomic proposition* $Q \in \mathcal{Q}$*, such that* Q *holds for every state in this set.* $\mathcal{V}(\mathsf{true}) = S$ *and* $\mathcal{V}(\mathsf{false}) = \varnothing$*.* □

The combination of a labelled transition system $\mathcal{T}$, the associated set of atomic propositions $\mathcal{Q}$ and the valuation function $\mathcal{V}$ is termed a *model*, denoted $\mathcal{M}$. A model is deemed to be a complete description of the computational process under investigation.

2 The Modal Mu-Calculus

2.1 Subsections of the Mu-Calculus

The word "modal" refers to modal operators. Modalities were introduced by [Hennessy, Milner 80], and allow one to make statements about the existence or non-existence of certain designated paths, starting from the initial state. The basic existential modality is denoted by a transition label inside a diamond, and its universal dual inside a box. This is followed by the expression which is bound by the operator. The expression $\langle a \rangle \varphi$ thus claims "there exists an a-transition in the current state (leading to, say, state s') such that φ holds in s'." In turn, the formula φ may contain nested modalities. The box operator is equivalent to a universal quantifier, so that $[a]\,\varphi$ means that φ should hold for each outgoing a-transition from the current state. Note that if no a-transitions exist, the box operator expression is true by default, while the diamond defaults to false. The logic thus described is *Hennessy-Milner logic*, abbreviated HML.

Example 2.1 (Hennessy-Milner Logic).

- $\langle a \rangle\, \mathsf{true}$: It is possible to perform an a-action.
- $[a]\, \mathsf{false}$: It is not possible to perform an a-action.
- $\langle a \rangle([b]\, \mathsf{false} \vee [c]\, \mathsf{false})$: It is possible, via an a-action, to arrive in a state in which neither a b or c-action is possible.
- $[a]\, \langle b \rangle\, [c]\, \mathsf{false}$: After every a-action, it is possible to perform a b-action which will leave the system unable to do a c-action.

□

Propositional dynamic logic or PDL is an extension of Hennessy-Milner logic which allows regular expressions over the atomic actions inside the modal operator. A regular expression of this sort is called a *program*, and is inductively defined as an atomic action, or the result of applying one of the following operators to programs: *composition* (e.g. $a \cdot b$ means "do a followed by b"), *nondeterministic choice* (e.g. $a + b$ means "do a or b") and *repetition* (e.g. a^* means "repeat a a nondeterministically chosen number of times").

An alternative enrichment of HML is the inclusion of other modalities. *Computation Tree Logic* or CTL extends HML in this way, by temporal operators such as Until, Next and $\mathsf{Globally}$. Each of these operators is preceded by a universal or existential quantifier, resp. A and E. For example, $\mathsf{A}(\varphi\ \mathsf{U}\ \psi)$ holds if and only if every run of the system has the property $\varphi\ \mathsf{U}\ \psi$. ACTL and ECTL are those sections of CTL restricted to universal and existential quantification, respectively.

CTL in turn has further enhancements and enrichments such as CTL* [Emerson, Lei 86], which allows free mixing of quantifiers and temporal operators, e.g. $\mathsf{A}(P\ \mathsf{U}\ \mathsf{F}Q)$ is not in CTL because the F operator is not directly preceded by a quantifier.

2.2 The Modal Mu-Calculus

The modal mu-calculus subsumes all temporal logics mentioned above, that is, any formula in those logics can also be expressed in the mu-calculus. Following [Bradfield, Stirling 06], $L\mu$ will denote the modal mu-calculus, considered as a logical language. $L\mu$ allows the formulation of assertions about labelled transition systems. Proving these assertions for a specific LTS can then be done using traditional proof methods or algorithmic means.

The defining feature of $L\mu$ is the use of *fixed point* or *fixpoint* operators. We can provide semantics for $L\mu$ by interpreting an $L\mu$-expression as valid in a set of states, i.e. elements from $\mathcal{P}(S)$, telling us in which states the formula holds. These expressions are allowed to contain variables with interpretations also ranging over $\mathcal{P}(S)$. We can thus view the semantics of an $L\mu$-formula $\varphi(X)$ with a free variable X as a function $\varphi\colon \mathcal{P}(S) \to \mathcal{P}(S)$. All functions expressible in $L\mu$ with the given grammar can be shown to be *monotonic*. Due to this property, they are known to have a unique minimal and maximal *fixed point*, i.e. a value for X such that $\varphi(X) = X$.

Fixpoint operators can be most easily understood as *recursion*. Consider the formula $\nu X.\ P \wedge [\mathcal{L}]\, X$, where ν denotes the maximal fixpoint operator (its dual minimal operator is denoted μ). It should be read as "$\nu X \ldots$ is true if $P \wedge [\mathcal{L}]\, X$ is true, which is true if the proposition P holds in this state and X holds wherever we go next." It is the last part that gives rise to recursion: after performing any action in the modality (in this case any action in $\mathcal{L}$), the formula $\nu X \ldots$ has to hold in the subsequent state. This formula thus expresses the CTL equivalent $\mathsf{AG}P$ or "P holds in all states." The difference between the minimal fixpoint operator μ and the maximal operator ν is to be understood as μ enforcing finite recursion, while we may loop through ν forever.

$L\mu$ has strictly more expressive power than CTL* or any of the previously mentioned logics. One of its major strengths is the possibility to *alternate* between fixpoint operators, i.e. mix minimal and maximal fixpoints in a formula with mutual recursion. Again, the expressive power can be seen to strictly increase with the alternation degree (the number of alternations).

2.3 Syntax

The syntax of $L\mu$ is defined in a relatively standard manner, following e.g. [Stirling 96]. Atomic propositions are allowed to hold in sets of states, making use of the valuation function from the previous chapter. We will use σ to range over the fixpoint operators $\{\mu, \nu\}$. A fixpoint operator *binds* a fixpoint variable in the same way as a predicate logic quantifier does. An occurrence of variable X is said to be *free* if it is outside the scope of a binding operator σX.

The given syntax does not include Boolean negation. All expressions with negations can be transformed to *positive form* (i.e. without negations except on atomic propositions), using e.g. DeMorgan's laws and identities on modal operators and fixpoints (refer to [Bradfield, Stirling 06] for an overview). Negations can be "worked inwards" until they occur only on atomic propositions. We

assume that for every atomic proposition $Q \in \mathcal{Q}$, the negation of Q is also a proposition in $\mathcal{Q}$. Excluding negation is then without loss of generality.

Definition 2.2 (Syntax of $L\mu$). *A modal mu-calculus formula is given by the following grammar:*

$$\begin{array}{rcl} \boldsymbol{\Phi} & ::= & \mathsf{true} \mid \mathsf{false} \mid Q \mid \boldsymbol{\Phi} \wedge \ldots \wedge \boldsymbol{\Phi} \mid \boldsymbol{\Phi} \vee \ldots \vee \boldsymbol{\Phi} \mid \\ & & X \mid [a]\,\boldsymbol{\Phi} \mid \langle a \rangle\,\boldsymbol{\Phi} \mid \mu X.\, \boldsymbol{\Phi} \mid \nu X.\, \boldsymbol{\Phi} \end{array}$$

where $Q \in \mathcal{Q}$ is an atomic proposition and $X \in \tilde{\mathcal{X}}$ is a fixpoint variable. □

Modal operators have a higher precedence than Boolean operators, whereas fixpoint operators have the lowest precedence. This makes the fixpoint binding extend as far to the right as possible.

The set of formulas that can be generated with this grammar is denoted $L\mu(\tilde{\mathcal{X}})$, which shows the dependency on the set of variables.

Given a certain *base* (also *root*) formula Φ, there exists a partial order on the bound variables in Φ, so that $X <_{\Phi} Y$ ("X is shallower than Y") if the formula $\sigma Y \ldots$ is within the expression bound by $\sigma' X$. The lowest element in this order is called the *shallowest*.

The following lemma will be useful later on, where we will benefit from inserting "effectless" fixpoint operators, i.e. where the bound variable is not used in the subsequent expression.

Lemma 2.3 (Effectless Fixpoint). *A formula φ can be transformed into a semantically equivalent formula $\sigma X.\, \varphi$ by addition of an effectless fixpoint operator, where X is fresh in φ.* □

The function form retrieves the subformula of the specification that falls under the scope of a given mu-calculus variable. For example, given a formula $\Phi = \mu X.\, \nu Y.\, \varphi \wedge \psi$, then $\mathsf{form}(\Phi, Y) = \nu Y.\, \varphi \wedge \psi$. To make this work, all bound fixpoint variables in the mu-calculus formula should be uniquely named. This is without loss of generality, as variables with the same name that do not occur in each other's scope can be renamed to a fresh identifier. For example, in an expression $(\mu X.\, \varphi) \wedge (\nu X.\, \psi)$ the X variables are distinct in φ and ψ, so one of them can be renamed to e.g. Y, resulting in the semantically equivalent $(\mu X.\, \varphi) \wedge (\nu Y.\, \psi[X := Y])$.[1] If the context is clear, the first argument will be omitted.

Definition 2.4 (Subformula Retrieval). *The partial function* $\mathsf{form} \colon L\mu(\tilde{\mathcal{X}}) \times \tilde{\mathcal{X}} \mapsto L\mu(\tilde{\mathcal{X}})$ *retrieves the mu-calculus expression associated with the given variable. The value of* $\mathsf{form}(\varphi,\, X)$ *is defined only if X is bound in φ.*

[1] Where $\psi[X := Y]$ denotes the syntactic replacement of all occurences of X in ψ with Y.

$$
\begin{array}{rcl}
\mathsf{form}(\sigma Y.\ \varphi,\ X) & = & \begin{cases} \sigma Y.\ \varphi & \textit{if } X = Y \\ \mathsf{form}(\varphi,\ X) & \textit{otherwise} \end{cases} \\
\mathsf{form}(\varphi_1 \wedge \ldots \wedge \varphi_n,\ X) & = & \mathsf{form}(\varphi_i,\ X) \textit{ for } X \textit{ bound in } \varphi_i \\
\mathsf{form}(\varphi_1 \vee \ldots \vee \varphi_n,\ X) & = & \mathsf{form}(\varphi_i,\ X) \textit{ for } X \textit{ bound in } \varphi_i \\
\mathsf{form}([a]\,\varphi,\ X) & = & \mathsf{form}(\varphi,\ X) \\
\mathsf{form}(\langle a\rangle\,\varphi,\ X) & = & \mathsf{form}(\varphi,\ X)
\end{array}
$$

□

2.4 Semantics

The semantics of an $L\mu$ formula φ is given by an *interpretation function*, which gives the set of states of a transition system $\mathcal{T}$ satisfying that formula. In addition, the interpretation function takes a third argument: an *environment* θ (explained below). It is fully written $[\![\varphi]\!]^{\mathcal{T}}_{\theta}$, but may be abbreviated by omitting the LTS and environment.

The satisfaction relation, written $\models$, is defined in terms of the interpretation function. We shall write $s \models \Phi$ if and only if $s \in [\![\Phi]\!]$, and $s \not\models \Phi$ otherwise. This notation extends to transition systems: $\mathcal{T} \models \Phi$ if and only if the initial state s_0 of $\mathcal{T}$ is in the set $[\![\Phi]\!]$.

We have seen that the context for interpretation contains an *environment* in addition to the transition system. An environment is a function $\theta\colon \tilde{\mathcal{X}} \to \mathcal{P}(S)$ which assigns a set of states to free variables in an expression. We shall use the notation $\theta[X := \varsigma]$ to denote the environment that is equivalent to θ for all variables except X, i.e. $\theta(Y) = (\theta[X := \varsigma])(Y)$ for $Y \neq X$ and $\theta[X := \varsigma](X) = \varsigma$ (with $\varsigma \in \mathcal{P}(S)$). This operation has precedence over all other operators.

Definition 2.5 (Semantics of $L\mu$). *The semantics of $L\mu$ is inductively defined in the context of a model $\mathcal{M} = \langle \mathcal{T},\ \mathcal{Q},\ \mathcal{V}\rangle$.*

$$
\begin{array}{lcl}
[\![\mathsf{true}]\!]^{\mathcal{T}}_{\theta} & = & S \\
[\![\mathsf{false}]\!]^{\mathcal{T}}_{\theta} & = & \varnothing \\
[\![Q]\!]^{\mathcal{T}}_{\theta} & = & \mathcal{V}(Q) \\
[\![\neg Q]\!]^{\mathcal{T}}_{\theta} & = & S \setminus \mathcal{V}(Q) \\
[\![\varphi_1 \wedge \ldots \wedge \varphi_n]\!]^{\mathcal{T}}_{\theta} & = & [\![\varphi_1]\!]^{\mathcal{T}}_{\theta} \cap \ldots \cap [\![\varphi_n]\!]^{\mathcal{T}}_{\theta} \\
[\![\varphi_1 \vee \ldots \vee \varphi_n]\!]^{\mathcal{T}}_{\theta} & = & [\![\varphi_1]\!]^{\mathcal{T}}_{\theta} \cup \ldots \cup [\![\varphi_n]\!]^{\mathcal{T}}_{\theta} \\
[\![X]\!]^{\mathcal{T}}_{\theta} & = & \theta(X) \\
[\![[a]\,\varphi]\!]^{\mathcal{T}}_{\theta} & = & \{s \in S \mid \forall_{s' \in S}.\ s \xrightarrow{a} s' \implies s' \in [\![\varphi]\!]^{\mathcal{T}}_{\theta}\} \\
[\![\langle a\rangle\,\varphi]\!]^{\mathcal{T}}_{\theta} & = & \{s \in S \mid \exists_{s' \in S}.\ s \xrightarrow{a} s' \ \wedge\ s' \in [\![\varphi]\!]^{\mathcal{T}}_{\theta}\} \\
[\![\mu X.\varphi]\!]^{\mathcal{T}}_{\theta} & = & \bigcap\{S' \subseteq S \mid S' \supseteq [\![\varphi]\!]^{\mathcal{T}}_{\theta[X:=S']}\} \\
[\![\nu X.\varphi]\!]^{\mathcal{T}}_{\theta} & = & \bigcup\{S' \subseteq S \mid S' \subseteq [\![\varphi]\!]^{\mathcal{T}}_{\theta[X:=S']}\}
\end{array}
$$

□

Example 2.6 (Modal Mu-Calculus). Some examples of $L\mu$ formulas are [Bradfield, Stirling 06]:

- $\nu X.\ Q \vee (P \wedge [a]\,X)$: On every a-path, P holds until Q holds.

- $\mu X.\ Q \vee (P \wedge [a]\, X)$: On every a-path, P holds until Q holds, and Q eventually holds.
- $\mu X.\ [a]\, \mathsf{false} \vee \langle a \rangle\, \langle a \rangle\, X$: There exists a maximal a-path of even length.
- $\mu X.\ \nu Y.\ (P \wedge [a]\, X) \ \vee\ (\neg P \wedge [a]\, Y)$: P is true only finitely often on any a-path.

□

2.5 The Regular Mu-Calculus

In section Section 2.1, the use of regular formulas inside modal operators was shown in the logic PDL. This feature is now formalised for the modal mu-calculus following [Mateescu, Sighireanu 00]. The resulting logic is called the *regular mu-calculus,* and has the same expressive ability as the "vanilla" mu-calculus defined earlier. Modalities are of the general form $[R]\, \varphi$ or $\langle R \rangle\, \varphi$ where R is a regular expression.

Definition 2.7 (Regular Expression Syntax). *An* action formula α *defines a set of actions. A* regular formula R *allows the use of regular expressions over action formulas.*

$$\begin{array}{rcl} \alpha & ::= & \mathcal{L} \ \mid\ a \ \mid\ \overline{\alpha} \ \mid\ \alpha \cup \alpha \ \mid\ \alpha \cap \alpha \\ R & ::= & \alpha \ \mid\ R \cdot R \ \mid\ R + R \ \mid\ R^* \end{array}$$

□

$\overline{\alpha}$ denotes the set complement of α. The operator $\cdot$ stands for sequential composition, $+$ stands for nondeterministic choice and * for repetition (i.e. the Kleene star). The semantics of regular formulas is now made precise.

Definition 2.8 (Regular Expression Semantics). *First, the interpretation of an action formula α, written $[\![\alpha]\!] \subseteq \mathcal{L}$ gives the set of action labels in α:*

$$\begin{array}{lcl} [\![\mathcal{L}]\!] & = & \mathcal{L} \\ [\![a]\!] & = & \{a\} \\ [\![\overline{\alpha}]\!] & = & \mathcal{L} \setminus [\![\alpha]\!] \\ [\![\alpha \cup \alpha']\!] & = & [\![\alpha]\!] \cup [\![\alpha']\!] \\ [\![\alpha \cap \alpha']\!] & = & [\![\alpha]\!] \cap [\![\alpha']\!] \end{array}$$

Second, the interpretation of a regular formula R, written $\|R\| \subseteq S \times S$ gives those state pairs $\langle s, s' \rangle$ for which the label l in the transition $s \xrightarrow{l} s'$ is in α:

$$\begin{array}{lcl} \|\alpha\| & = & \{\langle s, s' \rangle \in S \times S \mid \exists_{a \in [\![\alpha]\!]} .\ s \xrightarrow{a} s' \in \delta\} \\ \|R_1 \cdot R_2\| & = & \|R_1\| \circ \|R_2\| \\ \|R_1 + R_2\| & = & \|R_1\| \cup \|R_2\| \\ \|R^*\| & = & \|R\|^* \end{array}$$

where $\circ$ denotes the composition and * the transitive-reflexive closure of binary relations.

Finally, the intepretation of regular modal operators can be defined:

$$\begin{array}{lcl} [\![[R]\,\varphi]\!]_\theta^{\mathcal{T}} & = & \{s \in S \mid \forall_{s' \in S}.\ \langle s, s' \rangle \in \|R\| \implies s' \in [\![\varphi]\!]_\theta^{\mathcal{T}}\} \\ [\![\langle R \rangle\,\varphi]\!]_\theta^{\mathcal{T}} & = & \{s \in S \mid \exists_{s' \in S}.\ \langle s, s' \rangle \in \|R\| \ \wedge\ s' \in [\![\varphi]\!]_\theta^{\mathcal{T}}\} \end{array}$$

□

Note the slight abuse in notation for the symbol $\mathcal{L}$, which is not only used for the set of all actions but also in the syntax of an action formula to refer to this set. Some authors prefer to use true instead or leave the action formula blank.

Some examples of formulas in the regular mu-calculus are $\langle a^* \rangle\,\mathsf{true}$, which expresses that an a-sequence of any length is possible, $[\mathcal{L}^*]\,\langle \mathcal{L} \rangle\,\mathsf{true}$, which states the absence of deadlock, and $[send]\,\langle \mathcal{L}^* \cdot receive \rangle\,\mathsf{true}$, which says that after a *send*-action, a *receive*-action is attainable.

3 Boolean Equation Systems

3.1 Syntax

Essentially, a Boolean equation system is a finite sequence of fixpoint equations over Boolean variables. Propositional operators ($\wedge, \vee$, etc.) may be used in the right-hand side of each equation.

Definition 3.1 (Grammar of a Boolean Equation System). *A Boolean equation system $\mathcal{E}$ is given by the following grammar:*

$$\begin{array}{rcl} \mathcal{E} & ::= & \epsilon \mid (\mu X = f)\ \mathcal{E} \mid (\nu X = f)\ \mathcal{E} \\ f & ::= & \mathsf{true} \mid \mathsf{false} \mid X \mid f \wedge f \mid f \vee f \end{array}$$

where ϵ is the empty BES and $X \in \mathcal{X}$ is a fixpoint variable. Each $f : \mathbb{B}^k \to \mathbb{B}$ for some $k \in \mathbb{N}$ is a proposition. □

We only consider equation systems that are *well-formed*, i.e. those in which a fixpoint variable occurs at the left-hand side in at most a single equation. In the remainder, $\sigma \in \{\mu, \nu\}$ will be used to refer to an arbitrary fixpoint symbol.

Definition 3.2 (Bound Variables). *For any equation system $\mathcal{E}$, the set of* bound variables $\mathsf{bnd}(\mathcal{E}) \subseteq \mathcal{X}$ *is the set of all variables occurring on the left-hand side of equations in $\mathcal{E}$:*

$$\mathsf{bnd}(\epsilon) = \varnothing$$
$$\mathsf{bnd}((\sigma X = f)\ \mathcal{E}) = \mathsf{bnd}(\mathcal{E}) \cup \{X\}$$

□

An ordering $\lhd$ is defined on bound variables, so that $X_i \lhd X_j$ indicates that the equation for X_i precedes the equation for X_j. The lowest element according to this ordering (i.e. the variable in the leftmost equation) is called the *shallowest* variable. We will colloquially use the terms "shallow" and "deep" to refer to this ordering, indicating variables near the top respectively near the bottom of the equation system.

Definition 3.3 (Occurring Variables). *For any equation system $\mathcal{E}$, the set of* occurring variables $\mathsf{occ}(\mathcal{E}) \subseteq \mathcal{X}$ *is the set of variables occurring on the right-hand side of all equations in $\mathcal{E}$, defined as follows:*

$$\mathsf{occ}(\epsilon) = \varnothing$$
$$\mathsf{occ}((\sigma X = f)\ \mathcal{E}) = \mathsf{occ}(\mathcal{E}) \cup \mathsf{occ}(f)$$

where $\mathsf{occ}(f)$ *is inductively defined as follows:*

$$\begin{aligned}
\mathsf{occ}(\mathsf{true}) &= \varnothing \\
\mathsf{occ}(\mathsf{false}) &= \varnothing \\
\mathsf{occ}(X) &= \{X\} \\
\mathsf{occ}(f \wedge g) &= \mathsf{occ}(f) \cup \mathsf{occ}(g) \\
\mathsf{occ}(f \vee g) &= \mathsf{occ}(f) \cup \mathsf{occ}(g)
\end{aligned}$$

□

Note that the occ function may be used to obtain the occurring variables in the whole BES, or those in a single equation, depending on the context.

A variable is called *bound* if it is in $\mathsf{bnd}(\mathcal{E})$, and is called *free* if it is not bound by any fixpoint operator in the equation system: $\mathsf{free}(\mathcal{E}) = \mathsf{occ}(\mathcal{E}) \setminus \mathsf{bnd}(\mathcal{E})$. An equation system $\mathcal{E}$ is said to be *closed* when there are no free variables, i.e. $\mathsf{free}(\mathcal{E}) = \varnothing$.

An equation system may be divided into *blocks*. A block is a sequence of consecutive equations having the same fixpoint operator. The number of alternations between these blocks is useful as a measure of the complexity of the equation system, and indeed occurs as a measure of the computational complexity of some of the algorithms for solving equation systems [Keiren 09].

An ordinal called the *rank* is assigned to each bound variable to identify in which block its defining equation occurs. The rank of a variable will be useful for solving the equation system using parity games (more about this later in Section 5.6, see also *ibid.*). Counting proceeds in reverse order, so that the deepest variable (the greatest according to $\lhd$) will have the lowest rank, equal to 0 if its fixpoint operator is maximal and 1 if it is minimal. The rank of a variable is odd if and only if its fixpoint operator is minimal.

Definition 3.4 (Variable Rank). *Given a Boolean equation system* $\mathcal{E}$*, the* rank *of a variable* $X \in \mathsf{bnd}(\mathcal{E})$*, notation* $\mathsf{rank}_{\mathcal{E}}(X)$*, is defined as follows:*

$$\mathsf{rank}_{(\sigma Y = f)\mathcal{E}}(X) = \begin{cases} \mathsf{rank}_{\mathcal{E}}(X) & \textit{if } X \neq Y \\ \mathsf{block}_{\sigma}(\mathcal{E}) & \textit{otherwise} \end{cases}$$

where $\mathsf{block}_{\sigma}(\mathcal{E})$ *is defined as:*

$$\mathsf{block}_{\sigma}(\epsilon) = \begin{cases} 0 & \textit{if } \sigma = \nu \\ 1 & \textit{otherwise} \end{cases} \qquad \mathsf{block}_{\sigma}((\sigma' Y = f)\mathcal{E}) = \begin{cases} \mathsf{block}_{\sigma}(\mathcal{E}) & \textit{if } \sigma = \sigma' \\ 1 + \mathsf{block}_{\sigma'}(\mathcal{E}) & \textit{if } \sigma \neq \sigma' \end{cases}$$

□

3.2 Semantics

The Boolean expressions in an equation system are evaluated in the context of an *environment*, denoted η. An environment is a function $\eta \colon \mathcal{X} \to \mathbb{B}$ which assigns

a Boolean constant to free variables in an expression. The result of applying a Boolean function f to an environment η, denoted $f(\eta)$, is the value of the function f after substituting each free variable X in f by $\eta(X)$. Environments are not necessarily complete, i.e. they may not assign a value to each free variable in a function.

We shall use the notation $\eta[X := b]$ to denote the environment that is equivalent to η for all variables except X, i.e. $\eta(Y) = (\eta[X := b])(Y)$ for $Y \neq X$ and $\eta[X := b](X) = b$. Equivalent notations used in the literature are $\eta[b/X]$ (common), and, confusingly, $\eta[X/b]$ (less common). This operation has precedence over all other operators.

For readability, we shall not syntactically distinguish between a semantic Boolean value and its representation.

Definition 3.5 (Interpretation). *Let $\eta : \mathcal{X} \to \mathbb{B}$ be an environment. The* interpretation $[\![f]\!]\,\eta$ *maps a propositional formula to* true *or* false*:*

$$
\begin{aligned}
[\![X]\!]\,\eta &= \eta(X) & & \\
[\![\mathsf{true}]\!]\,\eta &= \mathsf{true} & [\![\varphi \wedge \psi]\!]\,\eta &= [\![\varphi]\!]\,\eta \wedge [\![\psi]\!]\,\eta \\
[\![\mathsf{false}]\!]\,\eta &= \mathsf{false} & [\![\varphi \vee \psi]\!]\,\eta &= [\![\varphi]\!]\,\eta \vee [\![\psi]\!]\,\eta
\end{aligned}
$$

□

If an equation system $\mathcal{E}$ is closed, its solution is invariant with respect to the environment, i.e. $[\![\mathcal{E}]\!]\,\eta = [\![\mathcal{E}]\!]\,\eta'$ for all η, η'. We will thus omit the environment when dealing with closed equation systems, and simply write $[\![\mathcal{E}]\!]$.

The following lemma relates the semantics for open equation systems to that of closed equation systems. Following the notation for environments, we will write $\mathcal{E}[X := b]$ where $X \notin \mathsf{bnd}(\mathcal{E})$ and $b \in \mathbb{B}$ for the equation system in which every syntactic occurrence of X has been replaced by b.

Lemma 3.6 (Relation Between Closed and Open Equation Systems). *Let $\mathcal{E}$ be an equation system, and let η be an arbitrary environment. Assume $X \notin \mathsf{bnd}(\mathcal{E})$ is a propositional variable, and let $b \in \mathbb{B}$ be such that $\eta(X) = b$. Then $[\![\mathcal{E}]\!]\,\eta = [\![\mathcal{E}[X := b]]\!]\,\eta$.* □

Various techniques have been developed for solving a BES. Intuitively, a solution is a valuation for all left-hand side variables, such that each equation is satisfied, and furthermore that the minimality and maximality conditions dictated by the fixpoint operators are satisfied. Note that the fixpoint signs of shallow equations dominate those that follow. This phenomenon is a result of the nested recursion for evaluating the right-hand side equation of the shallowest variable. As a consequence, the solution is order-sensitive: the solution to $(\mu X = Y)(\nu Y = X)$, yielding all false, is different from the solution to $(\nu Y = X)(\mu X = Y)$, yielding all true. The complexity of a solution arises from this recursive definition.

Definition 3.7 (Characterisation of Solution). *The solution to a Boolean equation system is characterised by the following inductive definition. Given an environment η,*

$$\begin{aligned} [\![\epsilon]\!]\,\eta &= \eta \\ [\![(\sigma X = f)\ \mathcal{E}]\!]\,\eta &= \begin{cases} [\![\mathcal{E}]\!]\,(\eta[X := [\![f]\!]\,([\![\mathcal{E}]\!]\,\eta[X := \mathsf{false}])]) & \text{if } \sigma = \mu \\ [\![\mathcal{E}]\!]\,(\eta[X := [\![f]\!]\,([\![\mathcal{E}]\!]\,\eta[X := \mathsf{true}])]) & \text{if } \sigma = \nu \end{cases} \end{aligned}$$

□

A *global solver* computes the valuation of all bound variables in the equation system. On the other hand, when a *local* solver is asked to solve for a single variable (most likely the shallowest variable), it may not compute the valuation for all variables. Although this does not agree with the previous definition, where η was said to be a complete function, we prefer to leave this as is to reduce unnecessary complication of many subsequent references to this function. Where the distinction between global and local solvers is relevant, it will be discussed as a "special case," i.e. a trichotomy between true, false and "not evaluated."

3.3 Restricted Forms

An equation system is said to be in *simple form* when its right-hand side equations do not contain free variables or nested formulas (as in $(\varphi_1 \wedge \varphi_2) \vee \varphi_3$). The *standard form* further restricts the size of the right-hand side formulas to two. This is formalised below. Finally, *recursive* form prohibits the use of constants on the right-hand side of equations.

Definition 3.8 (Standard Form). *A Boolean equation system $\mathcal{E}$ is in* standard form *if each right-hand side expression consists of a conjunction $X_i \wedge X_j$, a disjunction $X_i \vee X_j$, a single variable X_i or a constant* true *or* false*, with $X_i, X_j \in \mathsf{bnd}(\mathcal{E})$.* □

Definition 3.9 (Simple Form). *A Boolean equation system $\mathcal{E}$ is in* simple form *if each right-hand side expression consists of a finite series of conjunctions $X_i \wedge \cdots \wedge X_j$, a finite series of disjunctions $X_i \vee \cdots \vee X_j$, a single variable X_i or a constant* true *or* false*, with $X_i, X_j \in \mathsf{bnd}(\mathcal{E})$.* □

Definition 3.10 (Recursive Form). *A Boolean equation system is in* recursive form *if none of its right-hand side formulas contain constants.* □

If an equation is both in standard and recursive form it will be said to be in *standard recursive form.* Note that constants can be eliminated in linear time (in the size of the equation system) such that the solution is preserved. The same goes for free variables when a suitable environment is given.

Any equation system can be transformed into standard form by creating additional equations in the equation system to represent nested formulas. For example, $\mathcal{E} = (\mu X = (a \vee b) \wedge c)$ can be transformed to the standard form $\mathcal{E}' = (\mu X = X' \wedge c)(\mu X' = a \vee b)$. The number of additional variables is linear in the size of the right-hand side expressions in $\mathcal{E}$.

Theorem 3.11 (Conversion to Standard Form). *Any equation system $\mathcal{E}$ can be rewritten to an equation system $\mathcal{E}'$ in standard form, preserving the solution. This is accomplished by applying the following identity until standard form is obtained:*

$$\begin{aligned}&(\sigma X = \varphi_1 \oplus \cdots \oplus \varphi_n)\ \mathcal{E} \ = \\ &\qquad (\sigma X = \varphi_1 \oplus \cdots \oplus \varphi_{n-1} \oplus X')\,(\sigma' X' = \varphi_n)\,\mathcal{E}\end{aligned}$$

where $\oplus \in \{\wedge, \vee\}$, X' is a fresh variable (i.e. $X' \notin \mathsf{occ}(\mathcal{E})$) and σ' can be either μ or ν without changing the solution.

Proof. Using straightforward application of the definition of the semantics. It is assumed that X' does not occur on the right-hand side of $\mathcal{E}$ or in φ_i for any i.

$$\begin{aligned}
&[\![(\sigma X = \varphi_1 \oplus \cdots \oplus \varphi_{n-1} \oplus X')\ (\sigma' X' = \varphi_n)\ \mathcal{E}]\!]\,\eta \\
&\quad = [\![(\sigma' X' = \varphi_n)\ \mathcal{E}]\!]\,(\eta[X := \\
&\qquad\qquad [\![\varphi_1 \oplus \cdots \oplus \varphi_{n-1} \oplus X']\!]\,(\,[\![(\sigma' X' = \varphi_n)\ \mathcal{E}]\!]\,\eta[X := f(\sigma)]\,)]) && (1)\\
&\quad = [\![(\sigma' X' = \varphi_n)\ \mathcal{E}]\!]\,(\eta[X := \\
&\qquad\qquad [\![\varphi_1]\!]\,(\,[\![(\sigma' X' = \varphi_n)\ \mathcal{E}]\!]\,\eta[X := f(\sigma)]\,) \\
&\qquad\qquad \oplus \cdots \oplus \\
&\qquad\qquad [\![X']\!]\,(\,[\![(\sigma' X' = \varphi_n)\ \mathcal{E}]\!]\,\eta[X := f(\sigma)]\,)]) && (2)\\
&\quad = [\![(\sigma' X' = \varphi_n)\ \mathcal{E}]\!]\,(\eta[X := \\
&\qquad\qquad [\![\varphi_1]\!]\,(\,[\![\mathcal{E}]\!]\,(\eta[X := f(\sigma); X' := \ [\![\varphi_n]\!]\,(\,[\![\mathcal{E}]\!]\,\eta[X := f(\sigma); X' := f(\sigma')]\,)]\,) \\
&\qquad\qquad \oplus \cdots \oplus \\
&\qquad\qquad [\![X']\!]\,(\,[\![\mathcal{E}]\!]\,(\eta[X := f(\sigma); X' := \ [\![\varphi_n]\!]\,(\,[\![\mathcal{E}]\!]\,\eta[X := f(\sigma); \\
&\qquad\qquad\qquad\qquad X' := f(\sigma')]\,)]\,)\,)\,)] && (3)\\
&\quad = [\![(\sigma' X' = \varphi_n)\ \mathcal{E}]\!]\,(\eta[X := \\
&\qquad\qquad [\![\varphi_1]\!]\,(\,[\![\mathcal{E}]\!]\,(\eta[X := f(\sigma)]\,)) \\
&\qquad\qquad \oplus \cdots \oplus \\
&\qquad\qquad [\![\varphi_n]\!]\,(\,[\![\mathcal{E}]\!]\,(\eta[X := f(\sigma)]\,))]) && (4)\\
&\quad = [\![(\sigma' X' = \varphi_n)\ \mathcal{E}]\!]\,(\eta[X := \ [\![\varphi_1 \oplus \cdots \oplus \varphi_n]\!]\,(\,[\![\mathcal{E}]\!]\,\eta[X := f(\sigma)]\,)]) && (5)\\
&\quad = [\![\mathcal{E}]\!]\,(\eta[X := \ [\![\varphi_1 \oplus \cdots \oplus \varphi_n]\!]\,(\,[\![\mathcal{E}]\!]\,\eta[X := f(\sigma); X' := \ldots]\,)]) && (6)\\
&\quad = [\![\mathcal{E}]\!]\,(\eta[X := \ [\![\varphi_1 \oplus \cdots \oplus \varphi_n]\!]\,(\,[\![\mathcal{E}]\!]\,\eta[X := f(\sigma)]\,)]) && (7)\\
&\quad = [\![(\sigma X = \varphi_1 \oplus \cdots \oplus \varphi_n)\ \mathcal{E}]\!]\,\eta && (8)
\end{aligned}$$

$$\text{where } f(\sigma) = \begin{cases} \mathsf{true} \text{ if } \sigma \ = \ \nu \\ \mathsf{false} \text{ if } \sigma \ = \ \mu \end{cases}$$

Q.E.D. □

3.4 Dependency Graphs for Simple Form

A useful auxilary data structure when working with a Boolean equation system is its *dependency graph*. It shows the interdependencies and nesting structure of

variables in the equation system. The vertices in this graph represent the bound variables in the BES. An edge $X_i \to X_j$ means that X_j occurs in the right-hand side of the equation for X_i, indicating that X_i depends upon X_j. We begin by defining these graphs on equation systems in simple form.

Definition 3.12 (Dependency Graph). *Let $\mathcal{E}$ be a Boolean equation system in simple form. The* dependency graph *of $\mathcal{E}$ is a tuple $\mathcal{G}_\mathcal{E} = \langle V, \Delta \rangle$ where*

- $V = \mathsf{bnd}(\mathcal{E}) \cup \mathbb{B}$ *is a set of vertices;*
- $\Delta \subseteq V \times V$ *is the set of edges such that for all* $X_i \in \mathsf{bnd}(\mathcal{E})$:
  - $\langle X_i, X_j \rangle \in \Delta$ *for all* $X_j \in \mathsf{occ}(f_i)$*;*
  - $\langle X_i, \mathsf{true} \rangle \in \Delta$ *if* $f_i = \mathsf{true}$*;*
  - $\langle X_i, \mathsf{false} \rangle \in \Delta$ *if* $f_i = \mathsf{false}$.

□

We will now define some properties of dependency graphs.

Definition 3.13 (Paths in Dependency Graphs). *A* path *of length n in a dependency graph $\mathcal{G}_\mathcal{E} = \langle V, \Delta \rangle$ is a sequence $[v_0, v_1, \ldots, v_{n-1}]$ such that $v_i \in V$ for all $0 \leq i < n$ and $\langle v_i, v_{i+1} \rangle \in \Delta$ for all $0 \leq i < n-1$. The path is said to begin from v_0 and end at v_{n-1}.* □

Definition 3.14 (Reachability). *A vertex v_j is* reachable *from v_i in a dependency graph $\mathcal{G}_\mathcal{E}$ if there is a path in $\mathcal{G}_\mathcal{E}$ from v_i to v_j.* □

Based on the definition of reachability, we say that a variable X_i in the equation system *depends on* variable X_j if the vertex X_j is reachable from X_i in the dependency graph. Variables X_i and X_j are said to be *mututally dependent* if X_i depends on X_j and vice versa.

Definition 3.15 (Cycles). *A path $[v_0, \ldots, v_n]$ in a dependency graph $\mathcal{G}_\mathcal{E}$ is a* cycle *if $v_0 = v_n$.* □

Definition 3.16 (Lasso). *A path $[v_0, \ldots, v_n]$ in a dependency graph $\mathcal{G}_\mathcal{E}$ is a* lasso *if $v_n = v_j$ for some $0 < j \leq n$.* □

3.5 Structure Graphs

Problems with a Trivial Implementation

We would now like to extend the concept of a dependency graph to Boolean equation systems in general, i.e. without the simple form restriction. Lifting this restriction means that we should be able to deal unambiguously with nested formulas like $(\varphi_1 \wedge \varphi_2) \vee \varphi_3$ but also $(\varphi_1 \vee \varphi_2) \vee \varphi_3$ and $(\varphi \vee \varphi) \vee \varphi$. A trivial set of rules might, in the last example, create an extra vertex for the subformula $(\varphi \vee \varphi)$, which is then given an outgoing edge to φ.

This issue is illustrated in the figure below, where the semantically equivalent formulas $X \wedge (Y \wedge Z)$ and $(X \wedge Y) \wedge Z$ produce two dissimilar dependency graphs:

$$\begin{tikzcd} X \wedge (Y \wedge Z) \arrow[r] \arrow[d] & Y \wedge Z \arrow[r] \arrow[d] & Z \\ X & Y & \end{tikzcd} \qquad \begin{tikzcd} (Y \wedge X) \wedge Z \arrow[r] \arrow[d] & Z \\ Y \wedge X \arrow[r] \arrow[d] & X \\ Y & \end{tikzcd}$$

This is not an ideal situation, and we would prefer the semantic equivalence of the two formulas to be reflected as two equivalent dependency graphs. We are thus looking for a set of derivation rules that forego the precise syntactic structure of the formula, in return for the standard logical equivalences to hold.

Definition of Structure Graph

An elegant solution for the ambiguity problem, due to [Keiren, Reniers, Willemse 10], uses Structural Operational Semantics to generate what is called a *structure graph*. This is equivalent to the notion of a dependency graph defined earlier, but extends it with a number of decorations, such as the rank of bound fixpoint variables. These decorations are for instance useful when *parity games* are used to solve a Boolean equation system. These "games" work by traversing the dependency graph, and will be explored in more depth in Section 5.6. In addition to decorations, another important extension introduced by structure graphs is the concept that formulas can also be vertices in the graph, instead of just fixpoint variables.

We will now give the definition of structure graphs, and use this term to refer to dependency graphs in general from now on. We will then give a set of Structural Operational Semantics (SOS) deduction rules, which will be demonstrated to meet the criteria outlined in the previous section.

Definition 3.17 (Structure Graph). *Let $\mathcal{E}$ be a BES, and let $\mathcal{X}$ be its set of variables, that is, $\mathcal{X} = \mathsf{occ}(\mathcal{E}) \cup \mathsf{bnd}(\mathcal{E})$. A* structure graph *over $\mathcal{X}$ is a vertex-labelled graph $\mathcal{G}_{\mathcal{E}} = \langle V, v_0, \Delta, d, r, \nearrow \rangle$, where:*

- *V is a finite set of vertices;*
- *$v_0 \in V$ is the inital vertex;*
- *$\Delta \subseteq V \times V$ is a dependency relation;*
- *$d : V \mapsto \{\blacktriangle, \blacktriangledown, \mathsf{true}, \mathsf{false}\}$ is a vertex decoration mapping;*
- *$r : V \mapsto \mathbb{N}$ is a vertex ranking mapping;*
- *$\nearrow : V \mapsto \mathcal{X}$ is a free variable mapping.*

□

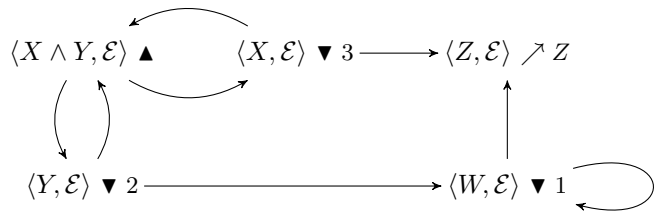
Intuitively, the decoration mapping d reflects whether the top symbol of a propositional formula is *true*, *false*, a conjunction or a disjunction; represented respectively by true, false, $\blacktriangle$ and $\blacktriangledown$. The vertex ranking mapping r indicates the rank of a vertex. The free variable mapping indicates whether a vertex represents a free variable. Note that each vertex can have at most one rank, at most one decoration and at most one free variable.

For readability, we introduce some shorthand notations. The predicate $v \nearrow X$ represents $\nearrow (v) = X$. The predicate $v \pitchfork n$ represents $r(v) = n$. For $\star \in \{\blacktriangle, \blacktriangledown, \mathsf{true}, \mathsf{false}\}$, $v\star$ represents $d(v) = \star$. The notation $v \not\pitchfork$ represents $\neg(v \pitchfork n)$ for all $n \in \mathbb{N}$, i.e. the vertex v is not ranked.

To get a feeling for the nature of structure graphs, an example is shown below, where a structure graph is derived from the BES on the left. Observe that the term $X \wedge Y$ is shared by the equations for X and Y, and appears only once in the structure graph as an unranked vertex. There is no equation for Z; this is represented by the term Z, decorated only by the label $\nearrow Z$. The subterm $Z \vee W$ in the equation for W does not appear as a separate vertex in the structure graph, since the disjunctive subterm occurs within the scope of another disjunction, and is thus "flattened" by the derivation rules.

Example 3.18. A BES (left) and its structure graph (right).

$$\begin{aligned} \mu X &= (X \wedge Y) \vee Z \\ \nu Y &= W \vee (X \wedge Y) \\ \mu W &= Z \vee (Z \vee W) \end{aligned}$$



□

Derivation Rules

Plotkin-style Structural Operational Semantics rules [Plotkin 04] are given in Figure 3.1 which allow the structure graph to be derived from a Boolean equation system. These rules are taken to define what relationships hold—namely those we can establish from the rules. The format of these rules is in the traditional $\frac{\textit{premiss}}{\textit{conclusion}}$ style, where the premiss is a set of formulas and the conclusion is a positive formula. Each formula is of the form $\langle \varphi, \mathcal{E}\rangle$ where $\mathcal{E}$ can be seen as the background store or context.

Some rules include negative premises. A negative premiss is true when its positive form cannot be derived, or equivalently, when one of its positive *complements* can be derived. For example, the premiss $\neg\langle f, \mathcal{E}\rangle\blacktriangle$, used below, holds when the vertex f is labelled with an element in the complement of the set $\{\blacktriangle\}$, namely $\{\blacktriangledown, \mathsf{true}, \mathsf{false}\}$. For a review of negative premises see [Mousavi, Reniers, Groote 07].

As discussed earlier, a big advantage of using these deduction rules is that the associativity, commutativity and (a restriced form of) idempotence properties of propositional operators are implicitly covered by the rules.

Finally, a structure graph can be *normalised* so that each vertex that has successors will be ranked. A normalised structure graph induces a BES in simple form; note that a BES in simple form also yields a normalised structure graph. The normalisation process can take place by applying a number of SOS deduction rules (for details refer to [Keiren, Reniers, Willemse 10]).

Vertex decoration. Straightforward decoration axioms.

$$(1)\frac{}{\langle \mathsf{true}, \mathcal{E}\rangle \mathsf{true}} \qquad (2)\frac{}{\langle \mathsf{false}, \mathcal{E}\rangle \mathsf{false}} \qquad (3)\frac{}{\langle f \wedge f', \mathcal{E}\rangle \blacktriangle} \qquad (4)\frac{}{\langle f \vee f', \mathcal{E}\rangle \blacktriangledown}$$

Free variable labelling. Free variables are labelled as such.

$$(5)\frac{X \notin \mathsf{bnd}(\mathcal{E})}{\langle X, \mathcal{E}\rangle \nearrow_X}$$

Bound variable ranking. Vertices representing bound variables are labelled with a natural number representing the rank of the variable in the equation system.

$$(6)\frac{X \in \mathsf{bnd}(\mathcal{E})}{\langle X, \mathcal{E}\rangle \pitchfork \mathsf{rank}_{\mathcal{E}}(X)}$$

Flattening. If a subformula has the same top-level operator and is not ranked, dependencies of that subformula apply to the whole formula.

$$(7)\frac{\langle f, \mathcal{E}\rangle \blacktriangle \quad \langle f, \mathcal{E}\rangle \not\pitchfork \quad \langle f, \mathcal{E}\rangle \to \langle g, \mathcal{E}\rangle}{\langle f \wedge f', \mathcal{E}\rangle \to \langle g, \mathcal{E}\rangle} \qquad (8)\frac{\langle f', \mathcal{E}\rangle \blacktriangle \quad \langle f', \mathcal{E}\rangle \not\pitchfork \quad \langle f', \mathcal{E}\rangle \to \langle g', \mathcal{E}\rangle}{\langle f \wedge f', \mathcal{E}\rangle \to \langle g', \mathcal{E}\rangle}$$

$$(9)\frac{\langle f, \mathcal{E}\rangle \blacktriangledown \quad \langle f, \mathcal{E}\rangle \not\pitchfork \quad \langle f, \mathcal{E}\rangle \to \langle g, \mathcal{E}\rangle}{\langle f \vee f', \mathcal{E}\rangle \to \langle g, \mathcal{E}\rangle} \qquad (10)\frac{\langle f', \mathcal{E}\rangle \blacktriangledown \quad \langle f, \mathcal{E}\rangle \not\pitchfork \quad \langle f', \mathcal{E}\rangle \to \langle g', \mathcal{E}\rangle}{\langle f \vee f', \mathcal{E}\rangle \to \langle g', \mathcal{E}\rangle}$$

Change of operator. If a subformula has a different top-level operator, that subformula gets its own vertex.

$$(11)\frac{\neg\langle f, \mathcal{E}\rangle \blacktriangle}{\langle f \wedge f', \mathcal{E}\rangle \to \langle f, \mathcal{E}\rangle} \qquad (12)\frac{\neg\langle f', \mathcal{E}\rangle \blacktriangle}{\langle f \wedge f', \mathcal{E}\rangle \to \langle f', \mathcal{E}\rangle}$$

$$(13)\frac{\neg\langle f, \mathcal{E}\rangle \blacktriangledown}{\langle f \vee f', \mathcal{E}\rangle \to \langle f, \mathcal{E}\rangle} \qquad (14)\frac{\neg\langle f', \mathcal{E}\rangle \blacktriangledown}{\langle f \vee f', \mathcal{E}\rangle \to \langle f', \mathcal{E}\rangle}$$

Subterm is a bound variable. If a bound variable occurs in a formula, there is a dependency on it.

$$(15)\frac{\langle f, \mathcal{E}\rangle \pitchfork n}{\langle f \wedge f', \mathcal{E}\rangle \to \langle f, \mathcal{E}\rangle} \qquad (16)\frac{\langle f', \mathcal{E}\rangle \pitchfork n}{\langle f \wedge f', \mathcal{E}\rangle \to \langle f', \mathcal{E}\rangle}$$

$$(17)\ \frac{\langle f, \mathcal{E}\rangle \pitchfork n}{\langle f \vee f', \mathcal{E}\rangle \to \langle f, \mathcal{E}\rangle} \qquad (18)\ \frac{\langle f', \mathcal{E}\rangle \pitchfork n}{\langle f \vee f', \mathcal{E}\rangle \to \langle f', \mathcal{E}\rangle}$$

Introduction of fixpoint variables. Allows vertices to be derived and appropriately labeled for bound fixpoint variables in the BES.

$$(19)\ \frac{\sigma X = f \in \mathcal{E} \quad \langle f, \mathcal{E}\rangle\blacktriangle \quad \langle f, \mathcal{E}\rangle \not\pitchfork}{\langle X, \mathcal{E}\rangle\blacktriangle} \qquad (20)\ \frac{\sigma X = f \in \mathcal{E} \quad \langle f, \mathcal{E}\rangle\blacktriangledown \quad \langle f, \mathcal{E}\rangle \not\pitchfork}{\langle X, \mathcal{E}\rangle\blacktriangledown}$$

Dependency on constant or free variable. Introduce dependency on true or false.

$$(21)\ \frac{\sigma X = f \in \mathcal{E} \quad \neg\langle f, \mathcal{E}\rangle\blacktriangledown \quad \neg\langle f, \mathcal{E}\rangle\blacktriangle \quad \langle f, \mathcal{E}\rangle \not\pitchfork}{\langle X, \mathcal{E}\rangle \to \langle f, \mathcal{E}\rangle}$$

Direct dependency on bound variable. This rule covers formulas of the form $\sigma X = Y$.

$$(22)\ \frac{\sigma X = f \in \mathcal{E} \quad \langle f, \mathcal{E}\rangle \pitchfork n}{\langle X, \mathcal{E}\rangle \to \langle f, \mathcal{E}\rangle}$$

Top-level dependencies. If the right-hand side formula f of an equation $\in \mathcal{E}$ has a dependency, then the fixpoint variable X associated with this formula has that dependency.

$$(23)\ \frac{\sigma X = f \in \mathcal{E} \quad \langle f, \mathcal{E}\rangle \to \langle g, \mathcal{E}\rangle \quad \langle f, \mathcal{E}\rangle\blacktriangle \quad \langle f, \mathcal{E}\rangle \not\pitchfork}{\langle X, \mathcal{E}\rangle \to \langle g, \mathcal{E}\rangle}$$

$$(24)\ \frac{\sigma X = f \in \mathcal{E} \quad \langle f, \mathcal{E}\rangle \to \langle g, \mathcal{E}\rangle \quad \langle f, \mathcal{E}\rangle\blacktriangledown \quad \langle f, \mathcal{E}\rangle \not\pitchfork}{\langle X, \mathcal{E}\rangle \to \langle g, \mathcal{E}\rangle}$$

Fig. 3.1. *SOS deduction rules for deriving the structure graph from a Boolean equation system.*

4 Model Checking

The primary strength of formal modelling of computer systems is the possibility to provably verify claims about these systems. Formalisation of a system generally begins with making a minimal model that captures all salient behavioural aspects of the system. The type of model used here is that introduced in Section 1, based on labelled transition systems. A system model may be very complex, as in the case when multiple subprocesses are operating concurrently. For this reason, modelling is often carried out in a higher language, such as mCRL2 [Groote 08]. This language allows the use of many constructs which make the life of the modeller easier. For example, concurrent subprocesses can be entered separately, and afterwards composited into a single whole (a single LTS). Interaction between the subprocesses can be made explicit using dedicated operators. mCRL2 also supports the use of typed parameters, making it possible to express not just to express the action $receive$, but the action $receive(data)$ where $data\colon DataType$.

After the system under consideration has been entered in a higher language, a finite labelled transition system is automatically generated by the toolchain. The use of LTSes (or, equivalently, Kripke structures) as low-level system models is a *de facto* standard. The details of how a high-level model is translated into an LTS are beyond the scope of this text, but an important thing to note is that parallel processes will cause an exponential growth in the number of states in the transition system, with respect to the number of processes. This gives rise to the so-called *state-space explosion problem*: the composition of n processes of size k yields k^n states. We will come back to this issue later.

Now that we have a formal model in the form of a labelled transition system, we can proceed to make claims about this model. These claims say things like "the process cannot deadlock" or "every $receive$ action is eventually followed by a $send$ action." Statements such as these will be referred to as *(formal) specifications*, and we have already seen in Section 2 how they can be formulated in the mu-calculus. Many useful properties can also be expressed in other temporal logics, such as HML and CTL, but recall that the expressiveness of the mu-calculus is greater than that of all these logics. In the following chapters, work by other authors is explored which is sometimes restricted to less expressive logics, but in the rest of this work only the full mu-calculus shall be considered.

Model checking is the process of validating or verifying that a given temporal logic specification holds for a given model. The process by which this is carried out can take on many forms. A number of these will be briefly described in Chapter 5. We will see later that despite the use of various model check algorithms, the use of mu-calculus in combination with labelled transition systems will give rise to a similar type of diagnostic. The main focus here is the use of Boolean equation systems to perform the actual model check operation. The final result of any model check is a single Boolean value, true or false, which says whether the given specification holds for the model (or equivalently whether the given model satisfies the specification).

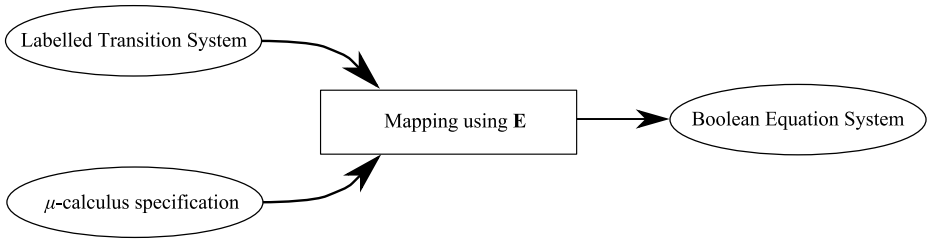


Fig. 4.1. *Model checking using Boolean equation systems.*

4.1 Model Checking using Boolean Equation Systems

A popular method of verifying whether a specification holds over a model is to encode this problem in the form of a Boolean equation system. The model check problem is encoded in the form of an equation system by use of the function **E**. This function maps subexpressions of the $L\mu$ specification onto states of the LTS (see Figure 4.1). The function actually consists of two parts: a linearisation function **E**, and a set of functions $\mathbf{E}_\mathrm{i}$ which are related to states s_i in the LTS (see Figure 4.2). The former is responsible for assigning each fixpoint in the specification to each state in the LTS. For example, given a formula $\mu X \ldots (\nu Y \ldots)$, the output of the **E** function will be a set of equations $\mu X_i = \ldots$ and $\nu Y_i = \ldots$ for each state $0 \leq i < |S|$. Each right-hand side expression is given by the $\mathbf{E}_\mathrm{i}$ function, which evaluates the given subexpression in state s_i.

Formally, the mapping function $\mathbf{E}(\Phi,\, \mathcal{M})$ maps a model and a specification to a Boolean equation system, but the second parameter will be omitted for readability. The number of equations in the resulting BES is bound by $\mathcal{O}(|\Phi| \cdot |\mathcal{M}|)$.

After the BES has been generated, it is solved to arrive at an answer for the model check problem. The equivalence of solving the model check problem and solving a Boolean equation system is formalised in the following theorem: a state in the model satisfies a property if and only if the corresponding variable in the derived Boolean equation system is true (this essential result is due to [Mader 96, Theorem 5.1]).

Theorem 4.1 (Transformation to BES Preserves Semantics). *Let $\Phi = \sigma X.\ \varphi$ be a mu-calculus specification, $\mathcal{M} = \langle \mathcal{T},\, \mathcal{Q},\, \mathcal{V} \rangle$ be a model and s_i a state of $\mathcal{T}$. Then for any environment $\eta_\mathcal{V}$:*

$$s_i \models^{\mathcal{M}} \Phi \quad \Longleftrightarrow \quad (\, [\![\mathbf{E}(\Phi,\, \mathcal{M})]\!]\, \eta_\mathcal{V})(X_i) = \mathsf{true}$$

□

The full mu-calculus specification will consistently be denoted with a capital letter (Φ) and subformulas of it with lower case letters (φ, ψ).

$$
\begin{array}{rcl}
\mathbf{E}(\mathrm{Q}) & = & \epsilon \\
\mathbf{E}(X) & = & \epsilon \\
\mathbf{E}(\varphi \wedge \ldots \wedge \psi) & = & \mathbf{E}(\varphi) \ldots \mathbf{E}(\psi) \\
\mathbf{E}(\varphi \vee \ldots \vee \psi) & = & \mathbf{E}(\varphi) \ldots \mathbf{E}(\psi) \\
\mathbf{E}([a]\,\varphi) & = & \mathbf{E}(\varphi) \\
\mathbf{E}(\langle a \rangle\,\varphi) & = & \mathbf{E}(\varphi) \\
\mathbf{E}(\sigma X.\varphi) & = & (\sigma X_0 = \mathbf{E}_0(\varphi)) \cdots (\sigma X_{n-1} = \mathbf{E}_{\text{n-1}}(\varphi))\ \mathbf{E}(\varphi) \\
\\
\mathbf{E}_{\mathrm{i}}(Q) & = & \begin{cases} \mathsf{true} & \text{if } s_i \in \mathcal{V}(Q) \\ \mathsf{false} & \text{otherwise} \end{cases} \\
\mathbf{E}_{\mathrm{i}}(X) & = & X_i \\
\mathbf{E}_{\mathrm{i}}(\varphi \wedge \ldots \wedge \psi) & = & \mathbf{E}_{\mathrm{i}}(\varphi) \wedge \ldots \wedge \mathbf{E}_{\mathrm{i}}(\psi) \\
\mathbf{E}_{\mathrm{i}}(\varphi \vee \ldots \vee \psi) & = & \mathbf{E}_{\mathrm{i}}(\varphi) \vee \ldots \vee \mathbf{E}_{\mathrm{i}}(\psi) \\
\mathbf{E}_{\mathrm{i}}([a]\,\varphi) & = & \begin{cases} \mathsf{true} & \text{if } \nexists_j.\ s_i \xrightarrow{a} s_j \\ \bigwedge\limits_{s_i \xrightarrow{a} s_j} \mathbf{E}_j(\varphi) & \text{otherwise} \end{cases} \\
\mathbf{E}_{\mathrm{i}}(\langle a \rangle\,\varphi) & = & \begin{cases} \mathsf{false} & \text{if } \nexists_j.\ s_i \xrightarrow{a} s_j \\ \bigvee\limits_{s_i \xrightarrow{a} s_j} \mathbf{E}_j(\varphi) & \text{otherwise} \end{cases} \\
\mathbf{E}_{\mathrm{i}}(\sigma X.\varphi) & = & X_i
\end{array}
$$

Fig. 4.2. *The mapping function* $\mathbf{E}$.

4.2 The Model Check Design Cycle

Model checking is most commonly employed before or during the design of the product. Using model checking post-hoc, on a completed design, is also possible, but the additional effort is less likely to pay off, as changes to the product become more costly later on in the design process. The cost of changes during development can be up to 6× as high as during design, while changes after release can be up to 100× as costly [Pressman 97]. Formal methods often have high start-up costs, mostly due to the specialised knowledge their usage requires. However, the potential benefits in complex and critical systems (e.g. aeronautics, medical) are unsurpassed by any software engineering method.

Professional software engineering tends to follow a certain process model that describes the tasks that should be carried out, in which order, to arrive at the end product. The end product is rarely a piece of software in isolation; documentation and some sort of quality assurance are also part of the product. There have been many process models developed, some generic, others for certain niches, one more

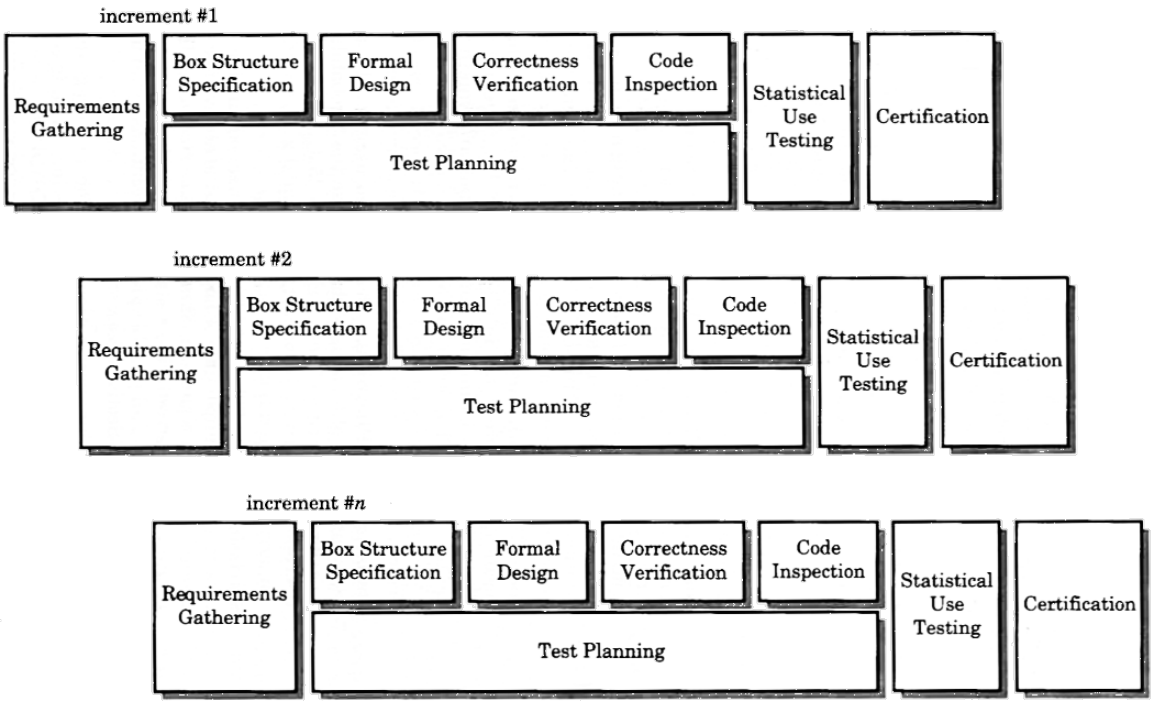


Fig. 4.3. *The "cleanroom" software engineering process flow.*

complex than the other. Most processes begin with requirements elicitation and end with a product release. The most simple *linear sequential* model iterates this route once, but e.g. the *spiral* and *prototyping* process models can iterate any number of times.

Some process models lend themselves better to the use of formal methods than others. It is beyond the scope of this work to perform an in-depth analysis, but we can make some general statements about the use of formal methods, and specifically diagnostics during software engineering. The *cleanroom* process model in particular is well suited for this. The philosophy behind this is to write incremental features right the first time, instead of relying on defect removal late in the process. Formal methods can be easily integrated due to the modularity of this process: each increment can be verified relatively independently before being added to the complete system (see Figure 4.3).

Diagnostics have an important role within the formal design and verification stages in each increment. A failure in a model check can be due to any of the following:

- A mistake in the specification;
- A mistake during modelling of the system under investigation, e.g. inaccurate modelling or incorrect use of abstraction (see below);

- An actual mistake in the system under investigation.

Given the generally accepted difficulty of applying formal methods, the first two points are common occurrences during the formal modelling and verification process. The model is often a strong simplification of the actual system, predominantly due to the state space explosion problem. The model will therefore have to be modified repeatedly, irrespective of whether the specification holds over the actual system. In the cleanroom process flow, this gives rise to a feedback loop between the "Formal Design" and "Correctness Verification" boxes. Only the third point is the real merit of formal modelling, when an actual fault is discovered in the design. How faults like these are handled depends on the process flow that is used. In the cleanroom model, minimal effort is wasted because only the abstract specification has to be modified, followed by a re-modelling and re-verification. In other process flows, e.g. where the model has been distilled from code, changes further back in the process are required, so that more effort will be spent re-doing the later steps.

Diagnostics have been called the most valuable product of model checking. A diagnostic produced by the model checker is much more easily understood than the corresponding error occurring in a detailed simulation trace. These errors are much more difficult to detect and diagnose either in simulation or in testing the actual product [McMillan 94]. Without diagnostics, mistakes in the product are very difficult to localise; after all, the only output of the model checker is a single Boolean value. This issue continuously grows in importance, as modern computers with more processing power allow larger and larger models to be verified. Lacking diagnostics, users have to rely on techniques like abstraction to cull the model and re-verify it, thereby learning whether the error was in the abstracted part or the remainder. This back-and-forth process can be quite costly in terms of time, even in simple cases where the model check takes no longer than, say, 10 minutes.

Instead, a good diagnostic allows the user to immediately pinpoint the location and nature of the error. This is no guarantee that an actual error (point 3 above) is easy to solve, but should make mistakes in the specification (point 1) much more obvious, and a decreased reliance on abstraction and other model simplification techniques will also reduce the incidence of mistakes during modelling (point 2). What makes a diagnostic "good" will be explicated in subsequent chapters.

Abstraction Refinement

In industrial-size model checking cases, the state-space explosion problem is a major difficulty. When the state space becomes unmanagably large, a potential solution is the use of *abstraction*. Abstraction amounts to omitting or simplifying sections of the model that are not relevant to verifying the property under consideration. The result of this is a (possibly very large) reduction in size of the state space, with obvious benefits.

The difficulty with abstraction is the decision which sections of the model can be safely pruned, i.e. without changing the model check result. There are several abstraction methods. *Over-approximation* progressively releases constraints, leading to a larger state space, while *under-approximation* removes behaviour from the original model. These may be used in conjunction with each other and other abstraction methods.

Abstraction is often a manual process which requires significant insight into the system model. An automated version of this process is called *counterexample-guided abstraction refinement* [Clarke et al. 03]. This particular method begins with a skeleton model, and computes increasingly accurate approximations of the full model. It accomplishes this by extracting information from false negatives, which are a result of the over-approximation. In case of a false negative, the information contained in this so-called *spurious* counterexample is used to refine the abstractions made.

Counterexample-guided abstraction refinement is an example where information from the diagnostic is used to modify not the model or the specification, but the level of abstraction. This is a valuable merit for any diagnostic, because of the necessity of using abstraction for real-world models. Although the role of diagnostics for abstraction refinement will not be discussed in detail here, it should be clear that certain types of diagnostic can be more useful in this regard than others. In the next chapter, we will explore several classes of diagnostics, some of which have a limited applicability in this area (such as linear paths) while others carry specific information about the subsection of the model in which the failure lies. Information from the diagnostic can then be used to refine the abstraction, for example by expanding upon the failing state or section.

5 Classes of Diagnostics

5.1 Basic Properties of Diagnostics

We have seen that the goal of model checking is to determine whether a formula encoding a property holds for the initial state of a transition system. Because we are using Boolean equation systems, the result of this operation is encoded as a Boolean fixpoint variable. A positive answer to the model check question results in this variable getting the value true, and a negative answer will result in false. To formalise this, say that we are verifying a mu-calculus specification $\Phi = \sigma X.\ \varphi$ over an LTS with initial state s_0. Call the first BES variable resulting from this mapping X_0. The model check theorem (Theorem 4.1) says that the result of the model check is equivalent to the truth value of this variable: $s_0 \models \Phi \iff X_0 = \mathsf{true}$.

When the automated model check tool returns its answer (true or false), its job is essentially done. However, this result in and of itself is not very insightful for the user. Especially given the LTS size of complex models, a single Boolean outcome usually does little to help the user with development. What is needed is a *diagnostic*, explaining *how* the tool arrived at the particular outcome. The diagnostic can take on a variety of forms, depending on the requirements and the methods used.

The end user is not concerned with the methods used to arrive at the *yes* or *no* result. All the user needs to be aware of is the specification (expressed in the mu-calculus or some other temporal logic) and the model (in the form of a labelled transition system). Although Boolean equation systems are used as an intermediate data structure by the tool, this need not and should not be relevant to the user. Therefore, a diagnostic is preferred that is entirely in terms of the mu-calculus specification and the LTS.

Counterexamples and Witnesses

Diagnostics come in two forms. In case the result of the model check operation was positive (i.e. the property holds in the initial state of the LTS), this result can be backed by a *witness*. In case the result was negative (i.e. the property does not hold), it can be proven by a *counterexample*. Both forms are needed so that the verification tool can fully motivate the outcome of the operation. We will see that each class of diagnostic discussed here has this duality, so that the same type of diagnostic can be used regardless of the model check outcome (this holds as well for the type of diagnostic introduced in Chapter 7: The Diagnostic Graph).

Explaining why a property Φ holds (using a witness) is equivalent to explaining why property $\neg\Phi$ fails to hold (using a counterexample). This is thanks to the fact that closed formulas in the mu-calculus are closed under negation.

Theorem 5.1 (Satisfiability of Negation). *Given a model $\mathcal{M}$ and a mu-calculus formula Φ, it holds that $\mathcal{M} \not\models \Phi \iff \mathcal{M} \models \neg\Phi$.* □

For example, take the CTL formula $\mathsf{EF}\varphi$, "there exists a transition sequence leading to a φ-state." If this property holds over an LTS, it can be demonstrated by a witness, namely a particular transition sequence leading to a φ-state. As a second example, take the formula $\mathsf{AF}\varphi$, "all transition sequences lead to a φ-state." If this property fails over the LTS, it can be demonstrated by a counterexample, namely a transition sequence leading to deadlock or to a loop without reaching a φ-state.

Now, take again the example $\mathsf{AF}\varphi$, but suppose this time the property holds. What should be the witness in this case? The witness should convey that there are no paths that do not lead to a φ-state. Trivially, we can return the subpart of the transition system leading up to φ-states, but this subpart is potentially very large and with a high degree of branching, so it will do little to further the understanding of the user. The same goes for the existential example: if there exist no paths, the counterexample is the entire transition system. This issue arises in the mu-calculus as it does for CTL, although in the former there is no such clear-cut distinction between universal and existential quantification.

Composition and Reduction

Usually, we are looking for a *reduced* or *minimal* diagnostic: one that is as concise as possible in explaining the failure. On the other hand, we want the diagnostic to be as *complete* as possible: it should fully explain the point(s) of failure in the model. These conflicting goals are compounded by the fact that forcing either may impact understandability. Furthermore, [Clarke et al. 95] show that for CTL formulas, finding a minimal linear path satisfying a set of constraints is NP-hard, so computing a minimal diagnostic may be computationally intractable.

If a *disjunctive* formula fails, each of its disjuncts necessarily fails, so we are usually looking for an explanation for the failure of each disjunct. These individual diagnostics combined can then be presented as a unified whole. In case a *conjunctive* formula fails, we have the option of finding diagnostics for all its failing conjuncts, or selecting a single one from them to further investigate. When the emphasis is on minimal size, the latter is obviously preferred (this is indeed the choice taken in Section 7.8: Reduced LTS Extraction). However, the decision can also be made on the basis of user requirements, technical implementation of the model checker and duration of the model check. For example, assume that we want some property to hold each time a "request" action occurs. Because this action occurs in several places in the transition system, a potential witness has to show that the property holds for each occurrence in order to be complete. However, this may be costly to compute, and the user may already be convinced after seeing a diagnostic for one of the occurrences (for example because of insight into the model; like knowing that all sections following a "request" are homologous).

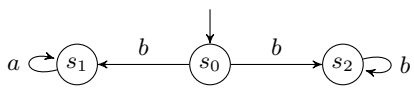
5.2 Linear Paths

Typically, a diagnostic is given in terms of states and transitions of the labelled transistion system. In its simplest form, these are given as a linear (i.e. non-branching) sequence[1]: a finite or infinite path through the LTS. In the case of an infinite path, a finite representation is given.

A path π in an LTS $\mathcal{T} = \langle S, s_0, \mathcal{L}, \delta \rangle$ is of the form $[s_{(0)} \xrightarrow{l_0} s_{(1)} \xrightarrow{l_1} \cdots \xrightarrow{l_{n-1}} s_{(n-1)}]$ such that $s_{(i)} \in S$, $l_i \in \mathcal{L}$ and $\langle s_{(i)}, l_i, s_{(i+1)} \rangle \in \delta$ for all $0 \leq i < n$. (The subscript brackets are used to distinguish between states in S and states in the sequence: s_0 does not necessarily correspond to $s_{(0)}$). Often, transition labels are omitted, in which case the path can be written $[s_{(0)}, s_{(1)}, \ldots, s_{(n-1)}]$. The notation $\pi[i]$ is used to extract the $(i+1)$-th element from the sequence, so that the first element $s_{(0)}$ has index 0.

The path π induces a labelled transition system $\mathcal{T}_\pi$, which forms a subpart of the original LTS $\mathcal{T}$, that is: those states, transitions and labels in $\mathcal{T}$ that also occur in π are in $\mathcal{T}_\pi$. In the ideal case, the path itself is sufficient to disprove the property, i.e. all information needed to disprove that $\mathcal{T} \not\models \Phi$ is contained in π, so that $\mathcal{T} \not\models \Phi \iff \mathcal{T}_\pi \not\models \Phi$. In this case, the path is known as a *counterpath*.

Example 5.2 (Counterpath). Let $\mathcal{T}$ be the LTS given below:



The property we wish to verify is $\Phi = \mu X.\ [\mathcal{L}]\, X \vee (\nu Y.\ \langle a \rangle\, \mathsf{true} \wedge [\mathcal{L}]\, Y)$, which expresses that "it will eventually be possible to always do an a-action."

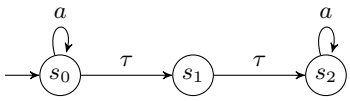
It is clear that the property fails over this LTS. This can be demonstrated by the path $\pi = [s_0, s_2, s_2, \ldots]$, because an a-action cannot be done for every state $\pi[i]$ for $i \geq 0$. The path π is by itself sufficient to disprove Φ: we do not have to consider states or transitions outside $\mathcal{T}_\pi$ to show that $\mathcal{T} \not\models \Phi$. Thus π is a counterpath. □

The modal mu-calculus, but also CTL variants allow expression of comprehensive, nested formulas, for which linear diagnostics are inadequate. This is illustrated by the following two examples.

Example 5.3 (Complex CTL Diagnostic). Consider the CTL formula $\mathsf{AFAX}\varphi$, which expresses that in all paths, we will eventually meet a state whose immediate successors all satisfy φ. A counterexample for this formula has to show the existence of an infinite path such that every state in the path has at least one immediate successor for which $\neg\varphi$ holds. Thus, the path itself is not sufficient: in addition, the diagnostic has to include the fact that each state in this path has at least one $\neg\varphi$ successor. □

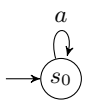
[1] Some equivalent terms used in the literature are *sequence*, *trace* and (linear) *path*.

Example 5.4 (Insufficiency of Linear Path Diagnostic). This example is due to *ibid.* Given the following labelled transition system:



Consider again the mu-calculus specification $\mu X.\ [\mathcal{L}]\, X \vee (\nu Y.\ \langle a \rangle\, \mathsf{true} \wedge [\mathcal{L}]\, Y)$. This property fails over the given transition system, because it is possible to continuously loop in s_0, i.e. $\pi = [s_0 \xrightarrow{a} s_0 \xrightarrow{a} \ldots]$ This violates the least fixpoint, causing the failure.

However, π by itself is not a complete counterexample. Consider the LTS $\mathcal{T}_\pi$ induced by π:



The specification does *not* fail over this structure. To fully demonstrate the failure, we also need to show that from each $\pi[i]$ (i.e. s_0), it is possible to go to a state (namely, s_1) in which an a-transition is not possible. We will return to this example later on.

□

The fragment of the mu-calculus that is guaranteed to admit counterpaths is very weak. Linear temporal logics, such as LTL and the linear subset of the mu-calculus (known as the *Linear Time Mu-calculus* LμTL) are known to admit counterpaths. This is the case because linear formulas are model checked along single paths. Counterpaths for these logics will in general be in the shape of a lasso, i.e. of the form $\varphi\psi^\omega$ [Clarke, Draghicescu 89]. For higher logics such as ACTL, it becomes NP-hard to decide whether a given model check problem admits a linear diagnostic and PSPACE-hard to recognise whether an arbitrary formula admits a linear diagnostic [Buccafurri et al. 01]. This leads *ibid.* to investigate the use of (ACTL) *formula templates*, instances of which are known to induce counterpaths. Because ACTL $\subset$ Lμ, it is clear that template methods are inadequate as a general diagnostic because of their lacking coverage of the full mu-calculus.

A path is easy to interpret, so users have come to rely on this form of diagnostic as a quick pointer to the general location of a problem. However, for the reasons discussed, they are often insufficient to unambiguously identify the exact source of the problem: the diagnostic path returned by a model checker is often not a counterpath. Despite their limitations, the simplicty of linear path diagnostics has led to relatively widespread implementation and continuing popularity, for example in the SMV and FDR model checkers [McMillan 93]. They can often be extracted from more complex diagnostics (as we will see in Section 7.7).

5.3 Branching Paths

The limitations of linear paths lead us to their natural extension, namely to augment them with branching structure. The previous examples showed that it can be necessary to consider further paths, starting from states in π, to show why the specification does not hold in that state. The resulting tree is called a *multi-path*, and has the original path π as its back-bone. This back-bone has branches at appropriate points, which in turn can branch out further. Multi-paths model nested paths in a labelled transition system.

Definition 5.5 (Multi-Path). *Let $\mathcal{T} = \langle S, s_0, \mathcal{L}, \delta \rangle$ be a labelled transition system.*

- *For every state $s \in S$, $\Pi = [s]$ is a finite multi-path in $\mathcal{T}$ with origin $\mathsf{or}(\Pi) = s$;*
- *If $\Pi_0, \Pi_1, \ldots$ are countably infinitely many multi-paths in $\mathcal{T}$ and $\langle \mathsf{or}(\Pi_i), l_i, \mathsf{or}(\Pi_{i+1}) \rangle \in \delta$ then $\Pi = [\Pi_0 \xrightarrow{l_0} \Pi_1 \xrightarrow{l_1} \ldots]$ is a multi-path in S with origin $\mathsf{or}(\Pi) = \mathsf{or}(\Pi_0)$.*

□

As with paths, the notation $\Pi[i]$ is used to extract the $(i+1)$-th element from the sequence, so that the first element has index 0. Also, labels are sometimes omitted in the multi-path, so that it is written $\Pi = [\Pi_0, \Pi_1, \ldots]$

Definition 5.6 (Main Path). *For any multi-path Π, the* main path *of $\Pi = \Pi_0 \xrightarrow{l_0} \Pi_1 \xrightarrow{l_1} \ldots$, denoted $\mathsf{main}(\Pi)$, is:*

- *s if $\Pi = s$;*
- *The path $[\mathsf{or}(\Pi[0]) \xrightarrow{l_0} \mathsf{or}(\Pi[1]) \xrightarrow{l_1} \ldots]$ otherwise.*

□

Note that the main path of a multi-path corresponds to the linear paths we reviewed in the previous section.

Example 5.7 (Multi-path). Assuming a suitable model, the diagram in Figure 5.1 represents a multi-path with main sequence $\pi_0 = [s_0 \xrightarrow{a} s_2 \xrightarrow{b} s_0 \xrightarrow{a} s_2 \xrightarrow{b} \ldots]$ on the right side in bold, and paths $\pi_1 = [s_0 \xrightarrow{c} s_1 \xrightarrow{d} s_1 \xrightarrow{d} \ldots]$ branching off at every even index. The complete multi-path $\Pi = [\pi_1 \xrightarrow{a} s_2 \xrightarrow{b} \pi_1 \xrightarrow{a} s_2 \ldots]$ □

Multi-paths are in principle sufficiently expressive to serve as a diagnostic for the entire modal mu-calculus. To see this, note that Lμ has the *tree model property*: if a formula has a model, it has a (potentially infinite) model that is in the shape of a tree[1]. We have also seen that, if a model does not satisfy a property

[1] Just unroll the original model (see below for an explanation of unrolling), thereby preserving bisimulation [Bradfield, Stirling 06].

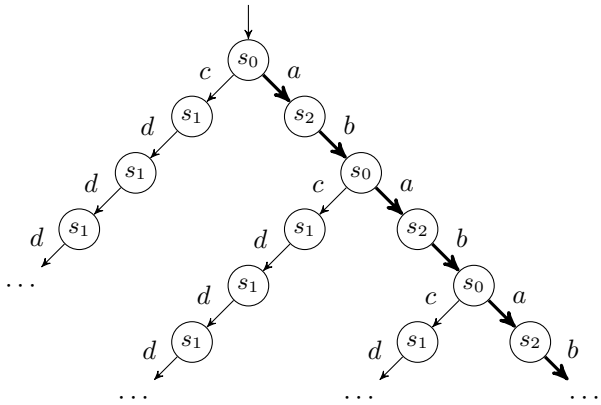


Fig. 5.1. *Multi-path for Example 5.7. The main path is indicated in bold.*

Φ, it satisfies its negation (Theorem 5.1). Let us assume that our model fails the specification, i.e. $\mathcal{T} \not\models \Phi$. We are looking for a counterexample, in this setting a multi-path, which demonstrates that $\mathcal{T} \models \neg\Phi$. The tree model property says that $\neg\Phi$ will be satisfied by a model in the form of a tree $\mathcal{T}'$ which is a subset of the original model. This tree can be directly interpreted as a multi-path.

Example 5.8 (Multi-Path as Counterexample). Reconsider the LTS and specification from Example 5.4. A multi-path demonstrating the failure is $\Pi = [[s_0, s_1], [s_0, s_1], \ldots]$ The branches $[s_0, s_1]$ demonstrate the failure of the subformula $\nu Y \ldots$ in state s_1. The main sequence shows that the least fixpoint is violated. Thus, the multi-path Π is a complete counterexample. □

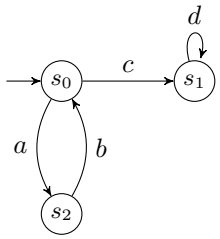
For ACTL, a symbolic, inductive definition of multi-path counterexamples is given by [Buccafurri et al. 01]. However, this method is only applicable to a subset of Lμ $\cap$ ACTL according to a set of templates. It is not clear how this can be extended to the entire mu-calculus, because not all formulas will exhibit the "neat" nesting structure of ACTL.

Furthermore, a difficulty with multi-path diagnostics is their interpretation. As we have seen earlier, linear path diagnostics are generally easy to understand when they are applicable. Multi-paths form an extension of linear paths where these are not sufficient. This means that the main sequence in the multi-path can only be understood as part of the whole multi-path, and not independently as with a linear path. The multi-paths in examples 5.7 and 5.8 give pause for thought, but if the branches of the main sequence have a nested branching structure, any intuitive understanding that the user may obtain from the diagnostic is lost.

5.4 Labelled Transition Systems

The idea to return a labelled transition system as a diagnostic is closely related to that of branching trees. In fact, an infinite multi-path as defined in the previous section can be straightforwardly and efficiently generated from an LTS using an "unrolling" process. Colloquially, this works as follows: suppose one begins in state s_0. The possible transitions from this state to other states $s_i, \ldots, s_j$ are added to a tree as branches $s_0 \to s_i, \ldots, s_0 \to s_j$. The process is then repeated for the newly added states. A single state may thus be repeated multiple times in the tree, so a self-loop in the LTS results in an infinite branch in the tree $s \to s \to \ldots$

Example 5.9 (Unfolding). The multi-path in Example 5.7 can be created by unfolding the following LTS:



□

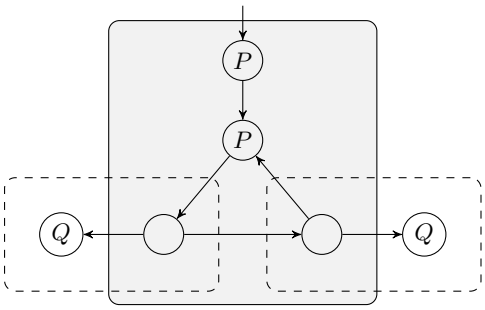
The sufficiency of this form of diagnostic is demonstrated by observing that the original model can always serve as a diagnostic—albeit not a very useful one. Ideally, the diagnostic would be a minimised version of the original LTS that violates the specification: the only requirement is that failure or success of the minimised LTS implies, respectively, failure or success of the original.

Definition 5.10 (Validity of Reduced LTS). *Let Φ be a temporal logic specification, and let $\mathcal{T}_r$ be the reduced or minimised counterpart to $\mathcal{T}$ with respect to the specification Φ. Then $\mathcal{T} \models \Phi \iff \mathcal{T}_r \models \Phi$.* □

[Clarke et al. 02] give a symbolic algorithm for generating ACTL counterexamples and ECTL witnesses. This algorithm returns what they call "tree-like" diagnostics: those labelled transition systems that can be obtained by glueing finite cycles to leaves of a tree and glueing finite trees to vertices in the cycles. The fact that this form of counterexamples is complete for ACTL is due to the "neat" nesting of temporal operators.

Colloquially, the method works as follows. Assume that the model check failed. By virtue of Theorem 5.1, it is known that the model satisfies the negated formula, so this negated formula can be used as a proof for the outcome of the model check. Therefore, the specification is negated and the negation is worked inwards using equalities (e.g. $\neg\mathsf{AG}x = \mathsf{EF}\neg x$). The authors then show that models for these formulas are of a tree-like form.

Example 5.11 (Tree-Like ACTL Counterexample). Assume that the ACTL specification $\Phi = \mathsf{AF}(\neg P \wedge \mathsf{AX}\neg Q)$ fails on a model $\mathcal{M}$. Then a counterexample for this model check problem is a labelled transition system that is a model of $\neg\Phi = \mathsf{EG}(P \vee \mathsf{EX}Q)$, an outline of which is shown below. The grey box indicates the section pertaining to the outer EG formula and the dashed boxes pertain to the inner EX subformula. Note that this LTS is "tree-like."



□

The computation of a counterexample for a formula Φ may be reduced to that of violated subformulas of Φ. This is demonstrated for an extension of ACTL, and an algorithm for computing the counterexample is given in *ibid.* This algorithm is not guaranteed to deliver minimal diagnostics, and in general, minimising the size of these counterexamples is NP-hard [Clarke et al. 95].

As with the multi-path algorithm discussed in the previous section, it is not immediately clear how the method here can be extended to cover the entire mu-calculus. The mu-calculus does not exhibit the neat nesting structure of ACTL, for example in alternating fixpoint formulas. However, this does not imply that this class of diagnostic is not viable for the mu-calculus, only that the resulting reduced transistion systems will not necessarily be tree-like.

The use of labelled transition systems as diagnostic, reduced in some way to focus on the source of the failure, are an improvement over the branching paths introduced in the previous section, mostly because they are more concise and therefore easier to interpret. Although a symbolic algorithm for Lμ is not available, this class of diagnostic is provided as a feature of the diagnostic method developed in Chapter 7 (see Section 7.8).

5.5 Tableaux Proofs

A requirement of any diagnostic is that it serves as a mathematically rigorous proof of failure. In the previous forms of diagnostic we have studied, the proof is only given as an end result: as a path or structure upon which the specification can be seen to fail. To see that the specification does indeed fail, we essentially need to repeat the model checking process on the returned diagnostic. The diagnostic is thus not as much a "proof" as it is an exemplar of failure in the model.

Proof-based diagnostics, on the other hand, use the semantics of the mu-calculus to arrive at a detailed, step-by-step proof structure. A proof of this sort is generated by applying derivational rules in the form of tableaux to the negation of the specification $\neg\Phi$. Because the verification of the specification failed, its negation holds, and furthermore this is justified because every step of the derivation is explicit.

Below, we reproduce a set of rules for $L\mu$, modified for simplicity from [Stirling, Walker 89]. (Note that a variety of rule-sets exist, e.g. for CTL [Gurfinkel, Chechik 03] and the negation-free [Dong, Ramakrishnan, Smolka 03] and full $L\mu$ [Cleaveland 90].) The proof rules operate on *sequents* of the form $s \models^{\mathcal{M}}_{\Delta} \varphi$, which are proof-theoretic analogues of $s \in [\![\varphi]\!]^{T}_{\mathcal{V}}$. The satisfaction relation is defined relative to a model $\mathcal{M}$ and an environment Δ (not related to a Boolean equation system environment). The function of Δ is to keep track of fixpoint expansion, so that fixpoint operators are not infinitely recursed resulting in an infinite tableau. This will be further explained below. The superscript $^{\mathcal{M}}$ may be omitted for readability.

Tableaux-based derivation proceeds top-down, and accordingly, the rules are written with conclusions appearing above premises, and are of the form:

$$(name)\frac{s \models^{\mathcal{M}}_{\Delta} \Phi}{s_1 \models^{\mathcal{M}}_{\Delta} \Phi_1 \quad \cdots \quad s_n \models^{\mathcal{M}}_{\Delta} \Phi_n}\mathcal{C}$$

where $n \geq 1$ and $\mathcal{C}$ is a Boolean condition. The rule is only applicable if the condition holds.

Axioms

$$(prop)\frac{s \models_{\Delta} P}{}\, s \in \mathcal{V}(P) \qquad (nprop)\frac{s \models_{\Delta} \neg P}{}\, s \notin \mathcal{V}(P)$$

$$(infrecurs)\frac{s \models_{\Delta} \nu X.\Phi}{}\, \langle s, \nu X.\, \Phi\rangle \in \Delta$$

Basic Logic

$$(doubleneg)\frac{s \models_{\Delta} \neg\neg\Phi}{s \models_{\Delta} \Phi} \qquad (and)\frac{s \models_{\Delta} \Phi \wedge \Psi}{s \models_{\Delta} \Phi \quad s \models_{\Delta} \Psi}$$

$$(nand1)\frac{s \models_{\Delta} \neg(\Phi \wedge \Psi)}{s \models_{\Delta} \neg\Phi} \qquad (nand2)\frac{s \models_{\Delta} \neg(\Phi \wedge \Psi)}{s \models_{\Delta} \neg\Psi}$$

Modal Operators

$$(box)\frac{s \models_{\Delta} [a]\,\Phi}{s_1 \models_{\Delta} \Phi \quad s_2 \models_{\Delta} \Phi \quad \ldots}\forall s \xrightarrow{a} s_i \qquad (diamond)\frac{s \models_{\Delta} \langle a\rangle\,\Phi}{s' \models_{\Delta} \neg\Phi}\exists s \xrightarrow{a} s'$$

Fixpoint Operators

$$(maxunroll)\frac{s \models_{\Delta} \nu X.\Phi}{s \models_{\Delta'} \Phi[\nu X.\,\Phi/X]}\langle s, \nu X.\,\varphi\rangle \notin \Delta \qquad \text{with } \Delta' = \nabla \cup \langle s,\ \nu X.\Phi\rangle$$

$$(minunroll)\frac{s \models_{\Delta} \mu X.\Phi}{s \models_{\Delta'} \Phi[\mu X.\,\Phi/X]}\langle s, \mu X.\,\varphi\rangle \notin \Delta \qquad \text{with } \Delta' = \nabla \cup \langle s,\ \mu X.\Phi\rangle$$

where $\nabla = \Delta - \{\langle s',\ \varphi\rangle \mid \sigma X.\,\Phi \prec \varphi\}$

Fig. 5.2. *Tableau rules for $L\mu$.*

The axioms (*prop*), (*nprop*) say that a state satisfies an atomic proposition if and only if it occurs in the valuation of P. Rule ($infrecurs$) says that a maximal fixpoint is satisfied when it is evaluated for a second time in the same state, i.e. it is possible to recurse forever. Note that a fourth "axiom" is rule (*box*) when no applicable outgoing transitions exist.

The logic rules are self-explanatory (negations can be worked inwards onto atomic propositions $\mathcal{Q}$). Rules (*box*) and (*diamond*) describe the modal operators. Finally, (*maxunroll*) and (*minunroll*) describe the expansion of fixpoint operators, which say that $\nu X.\varphi$ is satisfied in state s if φ is satsfied in that state, where all occurrences of X in φ are prepended by a fixpoint operator. This method of syntactic expansion (also *unrolling* or *unfolding*) is a simple yet effective method to iteratively describe the semantics of fixpoint operators. Note that $\neg\nu X.\varphi(X) = \mu X.\neg\varphi(\neg X)$, so that a rule for dealing with minimal fixpoints is redundant and can be derived from the others.

We are now in a position to detail the environment Δ. The environment is a set of state-formula tuples $\langle s,\ \sigma X.\,\varphi\rangle$ which encode the fact that fixpoint X has been unrolled in state s. Recursion through a fixpoint in the same state is a stopping condition, per rules ($infrecurs$), (*maxunroll*) and (*minunroll*). Note the use of the set ∇, which removes those tuples from the set where the current formula (in the premiss) occurs as a subformula. This is important to capture that recursion in some inner fixpoint is interrupted by recursion of the outer. (An example of this can be seen later, in Section 5.9). Removing formulas from the environment is called *discharging*.

A proof tree is called a *tableau* when it is maximal, that is, when no rules apply to sequents in leafs of the tree. A tableau is *successful* if the last rule applied in all leafs is an axiom. Every proof tree is of finite depth due the fact that fixpoint expansion is kept track of in the environment Δ, so that it can be used as a stopping condition.

Theorem 5.12 (Soundness and Completeness). *$s \models^{\mathcal{M}} \Phi$ has a successful tableau if and only if $s \in [\![\Phi]\!]^{\mathcal{T}}_{\mathcal{V}}$.* □

Theorem 5.13 (Finite Proof). *Every proof tree for $s \models^{\mathcal{M}} \Phi$ is finite.* □

Subformulas are explictly named in every sequent of the tableau. The derivation rules follow closely the semantics of the mu-calculus, and are are in themselves very easy to understand. Combined with step-by-step derivation, this makes tableaux-based diagnostics insightful to the user and in principle allow a fault to be traced quickly and accurately to its origin.

The downside to the tableaux method is its limited applicability to model checking using Boolean equation systems. Generating a proof tree is a model checking operation in itself. It is difficult to re-use the results from the model check using Boolean equation systems. Although the annotations that we will introduce later on the BES should make this a little easier, the differences in method between tableaux-based and BES-based model checking stand in the way of somehow unifying the two. However, we will see later that the two methods do share a common structure, and that it is possible to reproduce the advantages of the tableaux-based method.

5.6 Interactive Parity Games

We have seen that the model checking process can be carried out using a variety of methods. Parity games form one such method, and can be seen to offer some advantages over the use of Boolean equation systems. The size of an equation system can blow up exponentially in the number of variables involved during solving when using Gauß Elimination (see [Mader 96, section 6.4.2]). In addition, BES semantics can be hard to understand. Parity games use a graph structure which allows more insight into the problem and usage of additional algorithms. The complexity of solving the model checking problem is still exponential (using algorithms known at the time of writing).

We will first briefly review the way in which parity games allow us to solve the model checking problem. A parity game is played by two players on a directed graph. The players are often known as Odd and Even, and each player owns a set of vertices in the graph. A player does a step in the game if a token is on a vertex owned by that player. A *play*, denoted π, is a finite or infinite sequence of steps. Finally, a priority function assigns a natural number to each vertex.

Definition 5.14 (Parity Game). *A* parity game *is a four-tuple $\Gamma = \langle V, E, p, \langle V_{\mathsf{Even}}, V_{\mathsf{Odd}} \rangle\rangle$ where $\langle V, E\rangle$ is a directed graph with vertices V and total edge relation E, $p : V \to \mathbb{N}$ is a priority function, and $\langle V_{\mathsf{Even}}, V_{\mathsf{Odd}}\rangle$ is a partitioning of V.* □

A play is *won* by player Even if the highest[1] priority occuring infinitely often in a play π is even, and dually for player Odd. A *strategy* for a player is a partial function $\psi_{\mathsf{Player}} : V^* V_{\mathsf{Player}} \mapsto V$ that decides the vertex the token is played to based on the history of the vertices that has been visited[2]. A strategy is *winning* for a player from set $W \subseteq V$ if every play starting from a vertex in W, given Player's strategy, is winning for that player. It is well-known that for winning strategies, it suffices to look at *history-free* strategies, i.e. strategies $\psi_{\mathsf{Player}} \colon V_{\mathsf{Player}} \mapsto V$ in which a vertex always gets the same successor, independent of the path by which it was reached.

Parity games correspond to Boolean equation system in simple recursive form (see Definition 3.9 and 3.10). We can construct a parity game from a Boolean equation system by constructing its dependency graph: the game graph is the dependency graph for the equation system with a number of extensions.

Definition 5.15 (Parity Game from BES). *Let $\mathcal{E}$ be a Boolean equation system in simple recursive form and let $\langle V, \Delta \rangle$ be its dependency graph.*
The corresponding parity game graph $\Gamma_{\mathcal{E}} = \langle V, E, p, \langle V_{\mathsf{Even}}, V_{\mathsf{Odd}} \rangle \rangle$ is given by:

- $E = \Delta$;
- $p(X_i) = \mathsf{rank}_{\mathcal{E}}(X_i)$ *for all bound variables in* $\mathcal{E}$;
- $V_{\mathsf{Odd}} = \{X_i \mid f_i = \varphi \wedge \ldots \wedge \psi\}$, *so all conjuctive equations are assigned to player* Odd;
- $V_{\mathsf{Even}} = V \setminus V_{\mathsf{Odd}}$, *so all other equations are assigned to player* Even.

□

Parity games are equivalent to Boolean equation systems, in the sense that both methods can be used to solve the model check problem. The following theorem formally establishes this equivalence [Mader 96, Theorem 8.7]; as a result, we can conclude that $\mathcal{M} \models \Phi$ if and only if player Even has a winning strategy for the corresponding game.

Theorem 5.16 (Equivalence Between BES and Game). *Player* Even *has a winning strategy for the game on $\Gamma_{\mathcal{E}}$ with initial vertex X_0 if and only if* $(\llbracket \mathcal{E} \rrbracket\, \eta)(X_0) = \mathsf{true}$. □

To illustrate the equivalence between an (alternating) Boolean equation system and a parity game, Figure 5.3 shows a BES and its game graph side-by-side.

Parity games have been introduced as a method for solving the model check problem. The gamelike nature of this method gives rise to an interactive form of diagnostic. Observe that the model check tool always has a winning strategy, regardless of whether the outcome of the model check is positive or negative: in the latter case, it will have a strategy for player Odd, otherwise for Even. This

[1] This is the typical definition for max-parity games, for min-parity games replace *highest* with *lowest* [Keiren 09].
[2] Where * is the Kleene star.

$$
\begin{aligned}
\mu X &= X \wedge X' \\
\mu X' &= Y \vee Z \\
\nu Y &= W \vee Y' \\
\nu Y' &= X \wedge Y \\
\mu Z &= Z' \\
\mu Z' &= Z' \\
\mu W &= Z \vee (Z \vee W)
\end{aligned}
$$

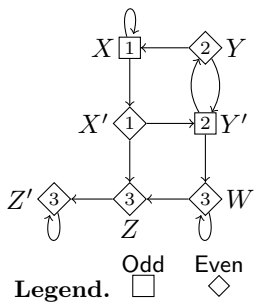


Fig. 5.3. *An example BES and its corresponding parity game.*

follows from Theorem 5.16 and Theorem 4.1: $\mathcal{M} \models \Phi \Leftrightarrow$ *Even has a winning strategy.* The player playing against the tool is thus destined to lose, but reaching a configuration that is winning for the tool may take any number of moves (see Figure 5.5: Winning Conditions).

A diagnostic of this form is thus a play π between the computer (the model check tool) and the user. The user was expecting the opposite outcome from the tool, and is thus convinced to be able to beat the tool. But of course, every time the user makes a move, the existence of the winning strategy for the tool ensures that it can make an appropriate countermove.

Each vertex in the play corresponds to a bound variable in the Boolean equation system. Therefore, each vertex corresponds to a state in the model and a subformula of the specification. (We shall have more to say about these correspondences in Section 5.9.) The play generally starts in the initial vertex X_0, associated with the initial LTS state s_0 and specification Φ. For readability, vertices will be denoted by their corresponding state and subformula.

The interactive nature of the diagnostic allows the user to steer the process by following a certain path in the transition system. This allows the user to apply domain knowledge of the model to limit the size and complexity of the diagnostic. Also, in combination with a suitable tool, an (initial) section of the game may be fully automated, releasing control of the losing player to the user when a section of interest is reached.

Depending on the outcome of the model check operation, the tool chooses whether it will play for player Even (the model check was successful) or player Odd (the model check failed). The user may make the moves for the other player. Suppose the token is currently on vertex v_i corresponding to $\langle s_i, \varphi_i \rangle$. Depending on the form of φ_i, a fixpoint may need to be unwound, which is when an expression of the form $\sigma X.\psi$ is unwound to ψ. In this case, there are no choices, and it is said that the "referee" moves (by unwinding the fixpoint). Recall that conjunctive equations are assigned to player Odd. When a token is on a vertex owned by Odd, we know that the corresponding formula was $\varphi \wedge \psi$. Odd is thus

- If $\varphi_i = \psi_1 \wedge \psi_2$ then Odd chooses φ_{i+1} to be either ψ_1 or ψ_2, and $s_{(i+1)}$ is $s_{(i)}$;
- If $\varphi_i = \psi_1 \vee \psi_2$ then Even chooses φ_{i+1} to be either ψ_1 or ψ_2, and $s_{(i+1)}$ is $s_{(i)}$;
- If $\varphi_i = [R]\,\psi$ then Odd chooses a transition $s_{(i)} \xrightarrow{a} s_{(i+1)}$ with $a \in R$ and φ_{i+1} is ψ;
- If $\varphi_i = \langle R\rangle\,\psi$ then Even chooses a transition $s_{(i)} \xrightarrow{a} s_{(i+1)}$ with $a \in R$ and φ_{i+1} is ψ;
- If $\varphi_i = \sigma X.\ \psi$ then φ_{i+1} is X and $s_{(i+1)}$ is $s_{(i)}$;
- If $\varphi_i = X$ and X is bound by $\sigma X.\ \psi$ then φ_{i+1} is ψ and $s_{(i+1)}$ is $s_{(i)}$.

Fig. 5.4. *Rules for the next move.*

Odd wins.

- The play is $\langle s_{(0)},\ \varphi_0\rangle \cdots \langle s_{(n)},\ \varphi_n\rangle$ and $\varphi_n = \mathsf{false}$;
- The play is $\langle s_{(0)},\ \varphi_0\rangle \cdots \langle s_{(n)},\ \varphi_n\rangle$ and $\varphi_n = \langle R\rangle\,\psi$ and $\neg\exists_s :\ s_{(n)} \xrightarrow{a} s$ for $a \in R$;
- The play $\langle s_{(0)},\ \varphi_0\rangle \cdots \langle s_{(n)},\ \varphi_n\rangle \cdots$ has infinite length and the unique variable X which occurs infinitely often identifies a least fixed point subformula $\mu X.\ \psi$.

Even wins.

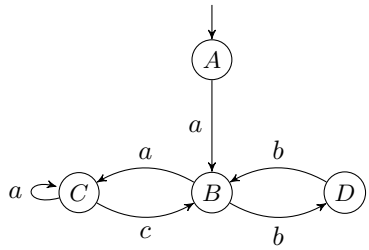
- The play is $\langle s_{(0)},\ \varphi_0\rangle \cdots \langle s_{(n)},\ \varphi_n\rangle$ and $\varphi_n = \mathsf{true}$;
- The play is $\langle s_{(0)},\ \varphi_0\rangle \cdots \langle s_{(n)},\ \varphi_n\rangle$ and $\varphi_n = [R]\,\psi$ and $\neg\exists_s :\ s_{(n)} \xrightarrow{a} s$ for $a \in R$;
- The play $\langle s_{(0)},\ \varphi_0\rangle \cdots \langle s_{(n)},\ \varphi_n\rangle \cdots$ has infinite length and the unique variable X which occurs infinitely often identifies a greatest fixed point subformula $\nu X.\ \psi$.

Fig. 5.5. *Winning conditions.*

offered the choice between moving to the vertex corresponding to φ or the vertex for ψ. The same goes for a formula of the form $[a]\,\varphi$. Complementary to this, player Even gets to choose between disjunctive terms and transitions for $\langle a\rangle\,\varphi$. The complete set of rules is given in Figure 5.4.

Interactive game diagnostics have been implemented in the *Edinburgh Concurrency Workbench (ECW)* [Stevens, Stirling 98]. The process proceeds as described above, with the addition of a terminating condition when the same configuration is entered for a second time. This prevents the user from having to evaluate infinite paths. For illustration, we repeat an example offered by *ibid.*

Example 5.17 (Interactive Game-Based Diagnostic). The following is an abbreviated dialogue between a user and the Edinburgh Concurrency Workbench. The labelled transition system is given below, and the property we wish to verify is $\nu X.\mu Y.(P \wedge \langle\mathcal{L}\rangle\, X) \vee \langle\mathcal{L}\rangle\, Y$, i.e. "there is some path on which P holds infinitely often," where $P = \langle a\rangle\,\mathsf{true} \wedge [\overline{a}]\,\mathsf{false}$, i.e. "$a$ is the only action possible."



The user may have made an error in the model, where a transition $D \xrightarrow{b} A$ was intended instead of $D \xrightarrow{b} B$. In addition, the user likely has a path in mind, for instance always following a b transition when possible, and expecting P to be true only at A. The following game against the tool may ensue (this is verbatim, with the exception of markup and omitted pause messages at each referee move):

```
Would you like to play (and lose!) a game against the CWB? (y or n) y
The CWB will choose Odd's moves. You can choose Even's.

1. Current position: ⟨A, X⟩
   The referee unwinds the fixpoint.

2. Current position: ⟨A, Y⟩
   The referee unwinds the fixpoint.

3. Current position: ⟨A, (⟨a⟩ true ∧ [ā] false ∧ ⟨ℒ⟩ X) ∨ ⟨ℒ⟩ Y⟩
   Your turn (playing Even)
   1. ⟨A, ⟨a⟩ true ∧ [ā] false ∧ ⟨ℒ⟩ X⟩
   2. ⟨A, ⟨ℒ⟩ Y⟩
   Which move? 1

4. Current position: ⟨A, ⟨a⟩ true ∧ [ā] false ∧ ⟨ℒ⟩ X⟩
   CWB's turn (playing Odd)

5. Current position: ⟨A, ⟨ℒ⟩ X⟩
   Your turn (playing Even)
   1. ⟨B, X⟩
   Which move? 1

6. Current position: ⟨B, X⟩
   The referee unwinds the fixpoint.

7. Current position: ⟨B, Y⟩
   The referee unwinds the fixpoint.

8. Current position: ⟨B, (⟨a⟩ true ∧ [ā] false ∧ ⟨ℒ⟩ X) ∨ ⟨ℒ⟩ Y⟩
   Your turn (playing Even)
   1. ⟨B, ⟨a⟩ true ∧ [ā] false ∧ ⟨ℒ⟩ X⟩
   2. ⟨B, ⟨ℒ⟩ Y⟩
```

```
    Which move? 2

 9. Current position: ⟨B, ⟨ℒ⟩ Y⟩
    Your turn (playing Even)
    1. ⟨D, Y⟩
    2. ⟨C, Y⟩
    Which move? 1

10. Current position: ⟨D, Y⟩
    The referee unwinds the fixpoint.

11. Current position: ⟨D, (⟨a⟩ true ∧ [ā] false ∧ ⟨ℒ⟩ X) ∨ ⟨ℒ⟩ Y⟩
    Your turn (playing Even)
    1. ⟨D, ⟨a⟩ true ∧ [ā] false ∧ ⟨ℒ⟩ X⟩
    2. ⟨D, ⟨ℒ⟩ Y⟩
    Which move? 2

12. Current position: ⟨D, ⟨ℒ⟩ Y⟩
    Your turn (playing Even)
    1. ⟨B, Y⟩
    Which move? 1

13. Current position: ⟨B, Y⟩
    The CWB (playing Odd) won, because of a repeat
    Another game? (y or n) n
```

□

5.7 Extended Boolean Graphs

The use of Boolean equation systems can be taken as central to the generation of a diagnostic. We have already seen how the dependencies between variables in an equation system provide a basis for the justification of the model check outcome. The present diagnostic method is also based on revealing these interdependencies, by removing superfluous information from the BES. The result is a dependency graph with as few edges as possible, which captures the minimum set of dependent variables that are "responsible" for the outcome.

In case of a diagnostic for a variable X, we are looking for a subset $\mathcal{E}'$ of the Boolean equation system $\mathcal{E}$ under the condition that by solving $\mathcal{E}'$ we obtain the same value for X as by solving $\mathcal{E}$. Note that $\mathcal{E}'$ may be open (i.e. containing free variables) even if $\mathcal{E}$ was closed. To make sure that the value for X is consistent, it should not depend on the environment of $\mathcal{E}'$ (where one such environment is that imposed by $\mathcal{E}$, which assigns truth values to variables that are free in $\mathcal{E}'$ but were bound in $\mathcal{E}$). A Boolean equation system that meets this condition is called *solution-closed*.

[Mateescu 00] extends the notion of a dependency graph with a conjuctive/disjunctive vertex labelling and a *frontier*, which is the set of vertices to which new edges may be added when the graph is embedded in another graph. The frontier corresponds to free variables in $\mathcal{E}$'.

Definition 5.18 (Extended Dependency Graph). *Let $\mathcal{E}$ be a Boolean equation system in simple form. The* extended dependency graph *of $\mathcal{E}$ is a tuple $\mathcal{G}_\mathcal{E} = \langle V, E, L, F \rangle$ where V is the set of vertices, $E \subseteq V \times V$ is the set of edges, $L\colon V \mapsto \{\wedge, \vee\}$ is the vertex labelling and $F \subseteq V$ is the frontier. V corresponds to variables in the equation system and E to dependencies between them as in a normal dependency graph (see Definition 3.12). $L(\mathsf{true}) = \wedge$ and $L(\mathsf{false}) = \vee$.* □

Note that *ibid.* foregoes labelling the vertices with their fixpoint operator $\{\mu, \nu\}$ because his exposition only considers alternation-free equation systems. *The following therefore applies only to alternation-free equation systems.* For simplicity, BES are assumed to be in simple form.

The extended dependency graph of a solution-closed BES will also be called solution-closed. We can use the graph to determine whether its corresponding BES is solution-closed, by looking at vertices in its frontier:

Theorem 5.19 (Solution-Closed Extended Dependency Graph). *An Extended Dependency Graph is solution-closed if and only if (1) all conjuctive vertices in its frontier are false, and (2) all disjunctive vertices in its frontier are true.* □

An extended dependency graph $\mathcal{G}_\mathcal{E}$ induces an LTS $T_\mathcal{E}$ such that each bound variable is a state, the transition relation is the dependency relation between variables, and transitions are not explicitly labelled. We can use this LTS to characterise the solution of the Boolean equation system using the following formulas:

Definition 5.20 (Example and Counterexample formulas). *The following formulas are called, respectively,* example formula *and* counterexample formula:

$$\begin{aligned} \mathrm{Ex} &= \mu X.(P_\vee \wedge \langle \mathcal{L} \rangle\, X) \vee (P_\wedge \wedge [\mathcal{L}]\, X) \\ \mathrm{Cx} &= \nu X.(P_\vee \wedge [\mathcal{L}]\, X) \vee (P_\wedge \wedge \langle \mathcal{L} \rangle\, X) \end{aligned}$$

where $s \models P_{\mathrm{op}} \iff L(s) = \mathrm{op}$ □

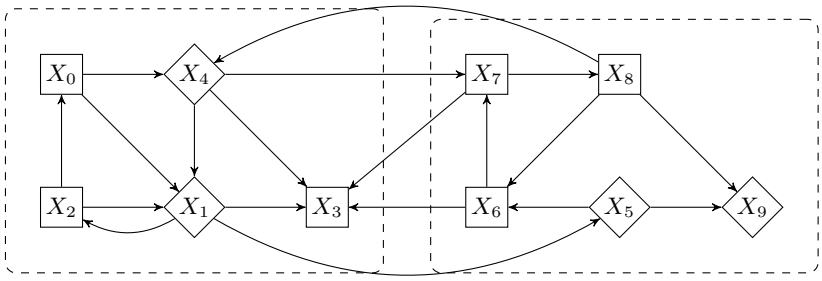
The satisfiability of these formulas on the induced LTS $\mathcal{T}_\mathcal{E}$ is equivalent to the outcome of the model check operation:

Theorem 5.21 (Characterisation of BES Solution). *Let $\mathcal{E}$ be a closed Boolean equation system and let $\mathcal{T}_\mathcal{E}$ be its associated LTS. Then $[\![\mathcal{E}]\!](X_i) = \mathsf{true}$ $\iff X_i \models^{\mathcal{T}_\mathcal{E}} \mathrm{Ex}$ for all left-hand side variables (or states) X_i.* □

We can now use Extended dependency graphs to reason about diagnostics, instead of the Boolean equation systems they are derived from.

Example 5.22 (Extended Dependency Graph Diagnostic). Consider the following closed Boolean equation system and its Extended dependency graph:

$$\begin{aligned}
\mu X_0 &= X_1 \wedge X_4 & \mu X_5 &= X_6 \vee X_9 \\
\mu X_1 &= X_2 \vee X_3 \vee X_5 & \mu X_6 &= X_3 \wedge X_7 \\
\mu X_2 &= X_0 \wedge X_1 & \mu X_7 &= X_3 \wedge X_8 \\
\mu X_3 &= \mathsf{true} & \mu X_8 &= X_4 \wedge X_6 \wedge X_9 \\
\mu X_4 &= X_1 \vee X_3 \vee X_7 & \mu X_9 &= \mathsf{false}
\end{aligned}$$



Legend. Variables in the left dashed box are true while those in the right box are false.

Conj. □ Disj. ◇

A potential diagnostic showing why X_0 is true (i.e. a witness for X_0) is the subsystem in the left dashed box, consisting of the vertices $\{X_0, \ldots, X_4\}$. Similarly, a counterexample for X_5 showing why it is false could be the subsystem in the right box, formed by the vertices $\{X_5, \ldots, X_9\}$. These two subsystems can be solved independently and the truth value obtained for X_0 and X_5 would be the same. Another potential diagnostic for X_0 is the subsystem $\{X_0, \ldots, X_4, X_6, X_7, X_8\}$. Note that all mentioned subsystems meet the conditions for solution-closedness. □

From the previous example, it is clear that minimal diagnostics are preferred. Minimality is defined with respect to a subgraph relation, and thus (via the number of vertices) with respect to the number of bound variables in the corresponding equation system. This notion is formalised below for witnesses; the dual for counterexamples (i.e. in case $[\![\mathcal{E}]\!](X) = \mathsf{false}$) is obtained by replacing disjunction with conjunction.

Definition 5.23 (Minimal Witness Diagnostic). *Let $\mathcal{E}$ be a Boolean equation system and let $\mathcal{G}_\mathcal{E} = \langle V, E, L, F \rangle$ be a diagnostic for $X \in V$. Then $\mathcal{G}_\mathcal{E}$ is* minimal *if and only if the following conditions hold:*

- $[\![\mathcal{E}]\!](X) = \mathsf{true}$ *(i.e. the diagnostic is a witness);*
- *All disjunctive vertices in V have exactly 1 outgoing edge;*

- *All vertices in V are reachable from X;*
- *The frontier F contains only disjunctive vertices.*

□

The definition of a minimal diagnostic in this context is similar to a parity game diagnostic. The last requirement in particular is important. Recall that disjunctive vertices in the dependency graph are assigned to player Even. In the given case (where the diagnostic is a witness), player Even is controlled by the model checker, and tasked with preventing the token from reaching a false vertex. This is easy, because as per the second requirement above, there is always precisely one outgoing edge, which will lead to a true vertex. It is in the interest of player Odd to try to reach a false vertex. The only way Odd is able to do this, is by choosing an edge that will lead outside of V. But all vertices in V with outgoing edges to false vertices are in the frontier. If we require that no conjunctive vertices may be in the frontier, we take away any possibility for Odd to foul the play. The definition of minimality above thus implies the existence of a winning strategy for the model checker on the "game graph" $\mathcal{G}_{\mathcal{E}}$ (playing Even in case of a witness, and Odd in case of a counterexample).

Extended dependency graphs form a compact and easy to interpret class of diagnostic. However, as outlined in the requirements earlier, the user is only aware of the transition system and specification, and need not be concerned with the use by the tool of Boolean equation systems. This does not mean that Extended dependency graphs cannot be used as the underlying formalism for diagnostics, but an additional presentational layer should output the diagnostic in a way that is more familiar to the user. In the CADP toolset, where this type of diagnostic is implemented, diagnostics can be returned in the form of an Extended boolean (sub)graph or the labelled transition system $\mathcal{T}_{\mathcal{E}}$ it induces [Mateescu 06]. However, this LTS turns out not to be always valid, i.e. Definition 5.10 may be violated, depending on the search strategy used. This seems to indicate a bug in this version of the toolset (`bcg_open` v1.5).

More importantly, the core theorems above are only applicable to the alternation-free segment of the mu-calculus. The use of alternation is essential to the powerful expressive power of $L\mu$, so providing diagnostics for the entire calculus is a hard requirement, placing this class of diagnostic at a serious disadvantage.

5.8 Support Sets

Support sets were introduced by [Tan, Cleaveland 02] and are tailored to model checking using Boolean equation systems. The purpose of a support set is to function as a type of generic evidence-collecting data structure that may be used by model checkers of various types. This evidence may then be used post-hoc as a basis for a user-presentable diagnostic, or to verify (certify) the model checking result.

The basic idea is to define a *support set* for one or more left-hand side variables in the BES. A support set contains information that can be used to support the truth value the model checker found for that variable. In particular, the set supports the outcome of that variable in terms of other variables. For example, a support set for fixpoint variable X contains an environment $\Xi(X)$, such that the interpretation of X under this environment is equal to the value which resulted from solving the BES, i.e. $[\![X]\!](\Xi(X)) = [\![\mathcal{E}]\!](X)$.

Definition 5.24 (Support Set). *Let $\mathcal{E}$ be a closed Boolean equation system with $\mathcal{X} = lhs(\mathcal{E})$, $X \in \mathcal{X}$ and $r \in \{0, 1\}$ is the truth value of X (resp.* $\mathsf{false}, \mathsf{true}$*). Then a support set for r and X is a tuple $\Gamma = \langle r, X, \Xi \rangle$ where $\Xi : \mathcal{X} \mapsto \mathcal{P}(\mathcal{X})$ is a partial function such that $\Xi(X)$ is defined. Furthermore, for each X_i where $\Xi(X_i)$ is defined, the following properties hold:*

- Direct Inference*: $[\![f_i]\!](\Xi(X_i)) = r$, i.e. under the interpretation $\Xi(X_i)$, f_i evaluates to r, the Boolean result of the support set;*
- Inclusion*: If $(\Xi(X_i))(X_j) = r$ then $\Xi(X_j)$ is defined, i.e. all variables on which X_i depends are in the domain of Ξ. The notation $X_i \xrightarrow{\Gamma} X_j$ means $(\Xi(X_i))(X_j) = r$;*
- Circularity Restriction*: If there exists a circular dependency $\rho = X_i \xrightarrow{\Gamma} \cdots \xrightarrow{\Gamma} X_i$ and X_j is the shallowest variable on ρ, then $r = 1 \Rightarrow \sigma_j = \nu$ and $r = 0 \Rightarrow \sigma_j = \mu$.*

□

A support set may be *reduced* or *minimised*, so that it holds no extraneous information. A minimised support set is similar to a minimised Extended dependency graph (but recall that EDGs are defined only for alternation-free equation systems in simple form). Assume a support set for X_i. The reachability condition for EBGs is translated to the requirement that all variables in the environment Ξ must affect X_i. In case the model check result was positive, all disjunctive vertices in a minimised EGB have only one outgoing edge (dually for conjunctive vertices in case the result was negative). This is translated to a requirement on the size of the environment, namely that $|\Xi(X_i)| = 1$ for disjunctive f_i (dually conjunctive).

The final requirement is on the frontier of the EBG. Recall that the frontier contains those vertices with free variables, including variables that have become free due to the fact that the minimisation process eliminated their binding equations. This requirement is not explicitly formulated for support sets, but follows from the definition. For EBGs, the requirement was that all vertices in the frontier are disjunctive for a positive model check result. This translates to the following. Assume a positive model check result ($r = 1$). Then for every variable X_i in the domain of Ξ that has free right-hand side variables, its equation f_i is disjunctive. If the converse were true, the variable X_i would be conjunctive, meaning that all its conjuncts also have $r = 1$ (otherwise it would itself not be true). This implies that the environment Ξ would then also be defined for all

conjuncts (per the first and second point in the definition), so that X_i is not in the frontier.

In the next section, we will see that the use of a graph as a common data structure lies closer to the various individual methods. The use of an "abstract proof structure" like support sets is an alternative, but more limited and somewhat contrived manner of performing the same function.

5.9 Correspondences between Classes

In this section, we will highlight the correspondences between different classes of diagnostics studied in the sections above. To formalise these correspondences, we will need the concept of a mu-calculus dependency, which will be introduced first.

Mu-Calculus Dependencies

Mu-calculus dependencies describe the nesting structure of mu-calculus formulas. As an introduction, consider propositional logic, and the equation $\varphi = \psi_1 \vee \ldots \vee \psi_n$. The value for φ can be seen to depend upon the values for $\psi_1 \ldots \psi_n$. This simple notion of dependency can be formalised as follows.

Definition 5.25 (Propositional Logic Dependency). *The* dependency set *of a propositional logic formula φ is the minimal set $\rightsquigarrow \subseteq Lp \times Lp$ (where Lp indicates propositional logic) that fulfills the following conditions. The notation $\varphi \rightsquigarrow \psi$ will be used for $\langle \varphi, \psi \rangle \in \rightsquigarrow$. The dependency set of φ is given by $\varphi \rightsquigarrow$*

- *If $\varphi = \psi_1 \wedge \ldots \wedge \psi_n$ then $\varphi \rightsquigarrow \psi_i$ for all $1 \leq i \leq n$;*
- *If $\varphi = \psi_1 \vee \ldots \vee \psi_n$ then $\varphi \rightsquigarrow \psi_i$ for all $1 \leq i \leq n$.* □

The transitive closure of the dependency relation is denoted $\rightsquigarrow^+$, and is the set union of all transitive dependencies of a formula φ. For example, if the dependency set of φ is $\{\langle \varphi, \psi \rangle\}$ and the dependency set of ψ is $\{\langle \psi, \xi \rangle\}$ then $\varphi \rightsquigarrow^+ = \{\langle \varphi, \psi \rangle, \langle \psi, \xi \rangle\}$.

The definition of dependencies can be extended to the mu-calculus in much the same way. A complication here is the recursion that is inherent to the mu-calculus. Recursion to a fixpoint results in a cyclic dependency, but there are a number of different syntactic methods to evaluate these cycles.

Most simply, when a term of the form X is encountered, it is taken to refer to or stand for $\mathsf{form}(\Phi, X)$, in other words $\sigma X.\ \varphi$, the expression in which X was bound. This is called *unwinding* or *unfolding* of a fixpoint. However, there are some variations on this. For example, in the model checking approach described in [Stevens, Stirling 98], formulas of the form $\sigma X.\ \varphi$ are first evaluated to X and next to φ.

Another variation is the so-called *unrolling* of fixpoints. This is a technique used by e.g. tableau-based model checking. If a formula of the form $\sigma X.\ \varphi$ is encountered, all occurrences of X in φ are replaced with the full expression $\sigma X.\ \varphi$, i.e. $\sigma X.\ \varphi \longrightarrow \varphi[X := \sigma X.\ \varphi]$. The expression $\varphi[X := \sigma X.\ \varphi]$ is said to be the unrolling of $\sigma X.\ \varphi$. This essentially replicates the entire formula at all positions where the bound variable occurs.

These variations in how formulas are evaluated during model checking are not essential. They are merely different syntactic methods to deal with recursion. We will see in a later chapter how mu-calculus dependencies appear in derivation trees and equation systems used for model checking. As described above, different model check methods will evaluate mu-calculus formulas in slightly different ways. These differences are reflected in the dependencies that can be found in the proof trees and equation systems.

Instead of artificially fitting the dependencies found "in the field" to a rigid definition of mu-calculus dependency, we acknowledge the variations that inevitably arise when different model check methods are used, by introducing variations in the definition of mu-calculus dependency. It is convenient to use a type of mu-calculus dependency that lies closest to what we will encounter while model checking. The fact that the various model check methods are essentially equivalent (in the sense that they obey the semantics of the mu-calculus) allows us this luxury; the types of mu-calculus dependency are also essentially equivalent. They vary only in how fixpoints are dealt with: by unrolling or unwinding, in one or two steps, etc.

Because Boolean equation systems are the central topic of this work, a notion of mu-calculus dependency that uses unwinding instead of unrolling is central. Two other variations are formalised in the interest of capturing the correspondences between different types of diagnostics in a later chapter. Which type of dependency is actually used depends on the context and application.

For simplicity, we only consider closed formulas. A subscript Φ is now added to the dependency relation $\rightsquigarrow$ to indicate the *base formula*. This is necessary for unwinding of fixpoints: a fixpoint variable X can only refer to a full formula $\sigma X \ldots$ in the context of a formula Φ that contains $\sigma X \ldots$ The subscript will be dropped when the base formula is clear from context.

Definition 5.26 (Mu-Calculus Dependency—Full Unwinding Type). *The* dependency set *of a mu-calculus formula φ is the minimal set $\rightsquigarrow_\Phi \subseteq L\mu \times L\mu$ that fulfills the following conditions and the propositional logic conditions in Definition 5.25.*

- *If $\varphi = [a]\,\psi$ then $\varphi \rightsquigarrow \psi$;*
- *If $\varphi = \langle a \rangle\,\psi$ then $\varphi \rightsquigarrow \psi$;*
- *If $\varphi = \sigma X.\ \psi$ then $\varphi \rightsquigarrow X$;*
- *If $\varphi = X$ then $X \rightsquigarrow \psi$ where $\mathsf{form}(\Phi,\ X) = \sigma X.\ \psi$.* □

Definition 5.27 (Mu-Calculus Dependency—Tableau/Unrolling Type). *The* dependency set *of a mu-calculus formula φ is the minimal set $\rightsquigarrow_{\Phi} \subseteq L\mu \times L\mu$ that fulfills the following conditions and the propositional logic conditions in Definition 5.25.*

- *If $\varphi = [a]\,\psi$ then $\varphi \rightsquigarrow \psi$;*
- *If $\varphi = \langle a \rangle\,\psi$ then $\varphi \rightsquigarrow \psi$;*
- *If $\varphi = \sigma X.\ \psi$ then $\varphi \rightsquigarrow \psi[X := \varphi]$.*

□

Definition 5.28 (Mu-Calculus Dependency—BES Type). *The* dependency set *of a mu-calculus formula φ is the minimal set $\rightsquigarrow_{\Phi} \subseteq L\mu \times L\mu$ that fulfills the following conditions.*

- *If $\varphi = \psi_1 \wedge \ldots \wedge \psi_n$ then for all $1 \leq i \leq n$: $\varphi \rightsquigarrow \begin{cases} \mathsf{form}(\Phi,\ X) & \text{if } \psi_i = X \\ \psi_i & \text{otherwise} \end{cases}$*
- *If $\varphi = \psi_1 \vee \ldots \vee \psi_n$ then (similar).*
- *If $\varphi = [a]\,\psi$ then $\varphi \rightsquigarrow \begin{cases} \mathsf{form}(\Phi,\ X) & \text{if } \psi = X \\ \psi & \text{otherwise} \end{cases}$*
- *If $\varphi = \langle a \rangle\,\psi$ then (similar).*
- *If $\varphi = \sigma X.\ \psi$ then*
  - *If $\psi = Y$, then $\varphi \rightsquigarrow \mathsf{form}(\Phi,\ Y)$;*
  - *If $\psi = \sigma' Y \ldots$ or $|\Xi| = 0$, then $\varphi \rightsquigarrow \psi$;*
  - *Otherwise, $\varphi \rightsquigarrow \xi_i$ for all $\xi_i \in \Xi$.*

  where Ξ is the dependency set of ψ, i.e. $\Xi = \psi \rightsquigarrow$

□

It is natural to express dependencies in graph form, so that each subexpression corresponds to a vertex and an edge from vertex φ to ψ means that the value for φ depends on the value for ψ.

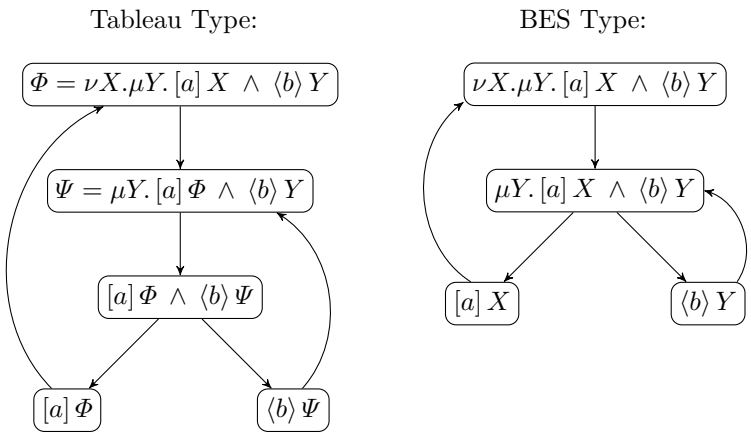
Definition 5.29 (Formula Dependency Graph). *The* Formula dependency graph *of a formula Φ is a directed graph $\mathcal{G}_{\Phi} = \langle V, E \rangle$ where V is a set of vertices and E is a set of edges $\subseteq V \times V$ that contains all reachable dependencies starting from Φ.*

- *$V = \{\Phi\} \cup \{\psi \mid \Phi \rightsquigarrow^{+}_{\Phi} \psi\}$;*
- *$\varphi \rightarrow \psi \in E$ if and only if $\varphi \rightsquigarrow_{\Phi} \psi$.*

□

The set of vertices in the graph $\mathcal{G}_\Phi$ is called the *closure* of Φ, denoted $\mathsf{cl}(\Phi) \subseteq \mathrm{L}\mu$. The closure of a closed formula is finite, so the formula graph will be finite as well (even in the case of unrolling [Fischer, Ladner 79]).

Example 5.30 (Formula Dependency Graph). The Formula dependency graph of $\nu X.\mu Y.\,[a]\,X \;\wedge\; \langle b\rangle\,Y$ can take on different forms, determined by the type of dependency used.



□

A Common Diagnostic Structure

The classes of diagnostics introduced in the previous sections appear rather different at first sight, in particular Extended boolean graphs, interactive (parity game) diagnostics and tableaux. However, these types of diagnostic can in fact be expressed in the same way, namely in the form of a graph. This graph is formalised in this section as a generic structure, so that it is independent not only of the model check method, but also of the particular diagnostic type (e.g. one of the three just mentioned). Instead, the use of a different diagnostic type or model check method can reveal itself by differences in the exact form of the graph. For example, as we shall see below, the use of tableaux will result in more detail than vanilla Boolean equation systems.

This type of generic diagnostic will be referred to as the *Generic diagnostic graph*. This graph will be defined precisely below. However, because this (hypothetical) data structure is intended to capture the *common basics* of any model check diagnostic, the specification leaves some room for variation. Any graph of this form can be interpreted in a specific and clear-cut manner, regardless of the level of detail it contains.

The pervasiveness of this generic form is of course no accident. All of the mentioned model check methods considered verify mu-calculus formulas over labelled transition systems. The semantics of the mu-calculus causes formulas to

be evaluated in the same way during the mapping to the LTS, regardless of the process by which this is carried out. For example, when using tableaux, box and diamond modalities are evaluated as the derivation proceeds: in case a sequent of the form $s \models [a]\,\varphi$ is encountered, the next step is to inspect the outgoing transitions of state s. In case this state has no outgoing a-transitions, an axiom can be applied, terminating the branch.

Conversely, when using Boolean equation systems, modal operators have already been evaluated during the mapping operation, courtesy of the mapping function **E**. Consider again the case that the state has no outgoing a-transitions. Then the BES that was the output of **E** would simply not include any equations for a-transitions in that state. The evaluation of the box modality has already taken place while evaluating the **E** function, and needs not be calculated "on the fly" as in tableaux. A resulting difference between the two discussed model check methods in this case will be that the BES diagnostic lacks these branch "stubs" that the tableau will have.

One and the same model checking problem may thus give rise to a variety of Generic diagnostic graphs. A key way to characterise the correspondences between them is dependency analysis. Dependency analysis has already been introduced for Boolean equation systems, giving rise to the dependency graph (Section 3.4), and for the mu-calculus, giving rise to the Formula dependency graph (Section 5.9). The Generic dependency graph can be seen as an extension of the BES dependency graph, with a subjugate role for the Formula dependency graph.

Consider a normal Boolean equation system dependency graph. In a dependency graph, each vertex corresponds to a fixpoint variable in the BES. Some extra information is now added to each vertex: a tuple $\langle s, \varphi \rangle$ where $s \in S$ is a state in the labelled transition system and $\varphi \in \mathsf{cl}(\Phi)$ is a subexpression of the mu-calculus specification. This tuple captures the semantics of a Boolean equation system variable, as stated earlier in Theorem 4.1, which says the following. Suppose the specification was of the form $\Phi = \sigma X.\ \varphi$. The fixpoint variable X is mapped to all states $s_0, \ldots, s_{|S|-1}$ of the transition system. A resulting BES variable X_i can then be said to encode the validity of φ in state s_i. In other words, if X_i is true, then $s_i \models \varphi$ whereas $s_i \not\models \varphi$ in case X_i is false. The truth value of the BES variable thus says whether a subexpression of the mu-calculus specification holds in a specific state. The state for variable X_i is simply s_i, and the subexpression is given by the subpart of the specification that is bound by σX (how to derive these annotations from a BES will be formalised in a later chapter; see Section 7.3).

Other model checking methods can also yield a graph where vertices are labelled with state-formula tuples, and where edges encode causal dependencies. This is a result of the model check process. Model checking the modal mu-calculus with respect to a labelled transition system generally occurs by mapping the formula, or subsections of it, to states in the LTS. Every model checker (of interest here) will use some variation of this method. The result of the mapping

is a set of tuples that is a subset of $S \times \mathsf{cl}(\Phi)$. Tuples in this set will correspond to vertices in the graph.

Above, we saw how mu-calculus dependencies express how the truth value of an expression depends on the truth values of one or more of its subexpressions. This is the same causal relation as is represented by edges in the Generic diagnostic graph between the state-formula tuples. It should come as no surprise that, like the state-formula tuples for vertices, the edge relation is essentially the same for various model check methods.

Definition 5.31 (Generic Diagnostic Graph). *Given a mu-calculus specification Φ and a labelled transition system, a* Generic diagnostic graph *is any graph $\mathcal{G} = \langle V, E \rangle$ for which the following hold:*

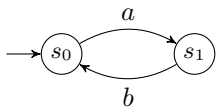
- *Vertices have a* state annotation *s and* formula annotation *φ, so that to each vertex corresponds a state-formula tuple $\langle s, \varphi \rangle$ with $s \in S$ and $\varphi \in \mathsf{cl}(\Phi)$;*
- *There is an* initial vertex *$\langle s_0, \Phi \rangle \in V$;*
- *There are special vertices* true *and* $\mathsf{false} \in V$ *which do not have annotations;*
- *An edge $\langle s, \varphi \rangle \rightarrow \langle s', \psi \rangle \in E$ implies both*
  - *$s = s'$ or $\langle s, s' \rangle \in \delta$; and*
  - *A mu-calculus dependency $\varphi \rightsquigarrow^{+}_{\Phi} \psi$ exists.*
- *A solution function assigns a truth value to each vertex, so that $s \models \varphi$ if and only if the vertex $\langle s, \varphi \rangle$ has the value* true.

□

As outlined above, the vertices in the dependency graph of a Boolean equation system can be augmented with a state-formula annotation, yielding a Generic dependency graph. This graph encodes some generic basics of the model checking problem, which apply to any model check method: each, after all, will need to evaluate the dependent clauses in a mu-calculus expression to arrive at a truth value for that expression. This follows from the semantics of the mu-calculus. By using Boolean equation systems, these tuples are encoded as bound variables, so that these variables appear on the vertices of the dependency graph. The Generic diagnostic graph only requires the presence of state-formula tuples, which can also be found if other model check methods are used (other than BES solving). This motivates the "generic" nomer.

To show how the same model checking problem can give rise to variations in the Generic diagnostic graph when different model check methods are used, the following problem is solved using both tableaux and Boolean equation systems. Diagnostic graphs will then be constructed for each method, after which the correspondences and differences between the two will be pointed out.

Example 5.32 (Generic Diagnostic Graph). Consider the mu-calculus formula $\Phi = \nu X.\ \mu Y.\ [a]\, X \wedge [b]\, Y$ and the following labelled transition system (this example is due to [Cleaveland 90]):



Using tableaux.
The abbreviation $\Psi = \mu Y.\ [a]\,\Phi \wedge [b]\,Y$ is used for readability; this is the first unrolling of Φ. The environments are abbreviated as follows (note the discharge of $\langle s_0, \Psi\rangle$ in Δ_3):

$$\begin{aligned}
\Delta_1 &= \{\langle s_0, \Phi\rangle\}\\
\Delta_2 &= \Delta_1 \cup \{\langle s_0, \Psi\rangle\}\\
\Delta_3 &= \Delta_1 \cup \{\langle s_1, \Phi\rangle\}\\
\Delta_4 &= \Delta_3 \cup \{\langle s_1, \Psi\rangle\}\\
\Delta_5 &= \Delta_4 \cup \{\langle s_0, \Psi\rangle\}
\end{aligned}$$

The tableau is:

$$\begin{array}{c}
s_0 \models \Phi \\ \hline
s_0 \models_{\Delta_1} \Psi \\ \hline
s_0 \models_{\Delta_2} [a]\,\Phi \wedge [b]\,\Psi \\ \hline
s_0 \models_{\Delta_2} [a]\,\Phi \qquad \dfrac{s_0 \models_{\Delta_2} [b]\,\Psi}{}\,(box) \\ \hline
s_1 \models_{\Delta_2} \Phi \\ \hline
s_1 \models_{\Delta_3} \Psi \\ \hline
s_1 \models_{\Delta_4} [a]\,\Phi \wedge [b]\,\Psi \\ \hline
\dfrac{s_1 \models_{\Delta_4} [a]\,\Phi}{}\,(box) \qquad s_1 \models_{\Delta_4} [b]\,\Psi \\ \hline
s_0 \models_{\Delta_4} \Psi \\ \hline
s_0 \models_{\Delta_5} [a]\,\Phi \wedge [b]\,\Psi \\ \hline
\dfrac{\dfrac{s_0 \models_{\Delta_5} [a]\,\Phi}{}}{s_1 \models_{\Delta_5} \Phi}\,(infrecurs) \qquad \dfrac{s_0 \models_{\Delta_5} [b]\,\Psi}{}\,(box)
\end{array}$$

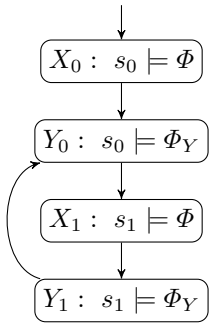
Turning the tableau into a Generic diagnostic graph is slightly nontrivial due to the environment Δ. In principle, the tableau tree can be converted one-to-one to a Generic diagnostic graph, because each sequent includes a tuple $\langle s, \varphi\rangle$, an (implicit) valuation (all sequents are true in this example), and because horizontal lines correspond to dependencies. The set of vertices in the Generic diagnostic graph is thus the set of sequents, while the edges go from each premiss to its conclusions directly below the horizontal line.

In a tableau, fixpoints are unfolded, and the environment Δ is used to detect recursion. The success due to infinite recursion of the outer fixpoint is represented as a stopping condition that becomes asserted. This occurs in the leaf labelled with $(infrecurs)$. Application of this axiom corresponds to the existence of a cycle both in the LTS and in the mu-calculus dependency chain. In the Generic diagnostic graph, this can be represented as a cycle. Application of other axioms, such as (box), should be represented as a dependency on the true-vertex.

Note the existence of a duplicate vertex in the Generic diagnostic graph: the tuple $\langle s_0, \Psi \rangle$ occurs twice under different environments Δ_1 and Δ_4. This does not warrant application of a recursion axiom due to the discharge mentioned earlier. Although this should not in principle impair tracing the reason for the model check result, it may be slightly confusing to the user. However, because the main focus of this work is Boolean equation systems, working out these details is beyond the scope. Suffice it to say that suitable presentation, perhaps made possible by tableaux-specific extensions to the Generic diagnostic graph, should bring satisfactory insight to the end user.

Using Boolean equation systems.
The Boolean equation system encoding the same model check problem is given below, as is its dependency graph. The abbreviation Φ_Y will be used for the subformula $\mu Y.\, [a]\, X \ \wedge\ [b]\, Y$ (note that we cannot use Ψ from the tableaux example because of the unrolling of νX).

$$\begin{aligned} \nu X_0 &= Y_0 \\ \nu X_1 &= Y_1 \\ \mu Y_0 &= X_1 \\ \mu Y_1 &= Y_0 \end{aligned}$$



The dependency graph has been annotated with the relevant state-formula tuples, so that it is already in the form of a Generic diagnostic graph.

Clearly, this graph contains much less detail than the one derived from the tableau. However, in spite of these differences, both graphs convey the same essential information. Each vertex in the BES GDG is also in the tableau GDG. Both contain a cycle with Φ as the most shallow formula, so that, due to the lack of a blantantly true or false configuration, the conclusion may be drawn in both cases that the success of the model check operation is due to infinite recursion of the maximal fixpoint. □

The procedure for drawing conclusions like the above from a Generic diagnostic graph will be elaborated in Section 7. However, at this point the value of even a raw Generic diagnostic graph should be clear: it allows tracing of the causality of the model check outcome in terms of the labelled transition system as well as the mu-calculus formula.

6 Supporting Methods

6.1 Generation of Diagnostics

The diagnostic may be generated independently from the model check operation, but if possible, re-use of (intermediate) data structures generated by the model checker during the model check process is preferred. However, in doing so, the form of the diagnostic may become constrained by the algorithm or specifics of the implementation. This may hamper the comprehension of the diagnostic, because it requires the user to know a great deal about the internals of the model checker and toolchain. This was discussed earlier for Boolean equation systems (Section 5.7).

The size of a diagnostic is in general polynomial in the size of the model, but this size can easily reach millions of states due to the state-space explosion problem, i.e. the fact that the number of states in a model is exponential in the size of the system description. Generating a diagnostic of considerable size independently of the model checker will therefore likely result in an unacceptable delay in the design cycle. Apart from time spent on generation, the diagnostic (or at least the data structure from which the diagnostic is computed) may be of such size that it exceeds the internal computer memory capacity, so any operations carried out on this data structure must be carefully considered.

6.2 Vacuity Detection

As we have seen, a great advantage of model checking is the ability of the tool to support a negative outcome by a counterexample, namely some structure that demonstrates the failure. On the other hand, when the outcome is positive, the result is rarely supported by a witness. Since a positive outcome means that the spefication holds over the model, this seems reasonable. However, it is not a guarantee that there are no mistakes in either. For example, a specification may be a tautology, making it trivially true over any model. Thus, also in case of a positive model checking outcome, the specification and model should be "suspect" [Kupferman, Vardi 99].

A classical example of this (due to [Beatty, Bryant 94]) is verifying a system with respect to the CTL specification $\mathsf{AG}(req \rightarrow \mathsf{AF}ack)$ ("every request is eventually followed by an acknowledge"). This implication can suffer from *antecedent failure* in case requests never occur in the model. The formula then holds trivially because the pre-condition (antecedent) of the formula is not satisfyable in the model. The model is said to *vacuously* satisfy the specification.

According to [Beer et al. 97], working in the IBM Haifa Research Laboratory, "Several years of experience [...] have shown us that during the first formal verification runs of a new hardware design, typically 20% of formulas are found to be trivially valid, and that trivial validity always points to a real problem in either the design of its specification or environment." Vacuity suggests an unexpected property of the system, namely the absence of a behaviour which was expected to have occurred. Satisfaction due to vacuity should thus be indicated

to the user. If it is not, the usefulness of formal verification is compromised, since a trivially valid formula is not intentionally part of a specification, and therefore indicates an error in the model or specification.

Of course, vacuity may be due to causes other than antecedent failure. We would like to capture the same problem in the semantically equivalent formula where $P \to Q$ is replaced by $\neg P \vee Q$. We are dealing with temporal logic, so we would also like the notion of vacuity to include a temporal aspect. Consider the following examples by *ibid.*: $\mathsf{AG}(P \to \mathsf{AX}(Q \to \mathsf{AX}\varphi))$. If P never occurs, then the formula holds trivially due to antecedent failure. Even if P does occur, the formula may be vacuous due to the fact that Q never occurs in a state directly following P. Another example is $\mathsf{AG}(\mathit{request} \to \mathsf{A}(\neg \mathit{data\_valid}\ \mathsf{U}\ \mathit{write\_enable}))$ holds vacuously in case there are no states in which $\neg \mathit{data\_valid}$ holds, i.e. when a *request* is accompanied by a *write_enable*. To cover all these cases, vacuity is defined as follows (in two steps):

Definition 6.1 (Affect). *A subformula φ of formula Φ* affects *Φ in model $\mathcal{M}$ if there is a formula ψ such that the truth values of Φ and $\Phi[\varphi := \psi]$ are different in $\mathcal{M}$.* □

Definition 6.2 (Vacuity). *Formula Φ is* vacuous *in model $\mathcal{M}$ if there is a subformula φ of Φ such that φ does not affect Φ in $\mathcal{M}$.* □

Ibid. proceeds to show how vacuity detection can be accomplished for logics like CTL* with complexity $\mathcal{O}(|\Phi| \cdot C_M(|\Phi|))$ where $C_M(n)$ is the complexity of checking a formula of size n in model $\mathcal{M}$. Note that this complexity stems from the definition, which requires checking all subformulas φ of Φ (it turns out that we need only consider replacement of φ by constants true and false, so we do not need to search for suitable replacement formulas ψ [Kupferman, Vardi 99]). The complexity can be decreased to $\mathcal{O}(C_M(|\Phi|))$ by restricting specification formulas to a subset of ACTL.

The user can often discover quickly that a specification holds vacuously by inspecting the witness. The user is of course under no obgligation to perform this inspection or may fail to observe the source of the failure in a complex specification. Vacuity detection should therefore preferably be automated and thus implies a third output option in addition to generating a counterexample and generating a witness: in case a formula is vacuously true, this should be indicated as such to the user as a special case.

The type of diagnostic developed in Chapter 7 supports quick insight into potential vacuous satisfaction. This is thanks to "formula annotations" in the graph, which show (in a user-friendly manner as described in the next section) that the antecedent failure of Q is responsible for satisfaction.

6.3 Presentation and Visualisation

From the previous expositions, we have seen that the user-friendly presentation or visualisation of a diagnostic can be as important as details about its underlying

formalism. In Section 4.2, we discussed the merits of diagnostics in the design cycle. Now that we know more about the nature of diagnostics and how a specific notion of diagnostic constrains the feedback given to the user, we can evaluate in more detail what this feedback should *look* like.

We do this by reviewing a tool called the *Evidence Explorer*, which was developed to assist the user in *interpreting*, *navigating* and *manipulating* the diagnostic generated by a model checker. The purpose of the tool is to improve the usability of model checking diagnostics in production environments. The tool accepts a diagnostic, in this case a directed graph of tuples $\langle s, \varphi \rangle$, which represent the assertion that s satisfies φ. Edges in the graph define dependencies between these assertions. This is of course the general form that could be derived from many of the diagnostic classes reviewed in the previous chapter, and was formalised in Section 5.9 as the Generic diagnostic graph.

An important property of models—and thereby model check diagnostics—is their sheer size. This makes single-step traversal of the graph prohibitively expensive. To assist the user as described in the previous paragraph, Evidence Explorer has a number of features:

- The diagnostic can be observed from various viewpoints called *views*, which will be described below;
- The scope of the exploration can be reduced according to some criterion of interest;
- Since a specification is usually small with respect to the model, formula-specific patterns may repeat themselves throughout the tree. The tool can visualise these homologies;
- The user may "jump" to any vertex in the tree by using an index;
- The user can search for paths containing certain actions and states with certain properties;
- Finally, a trace can be extracted from the selected portion, yielding a more succint type of diagnostic.

Views allow the user to observe the same diagnostic structure from different viewpoints, making it easy to locate interesting sections. The primary view is a simple treelike display of the vertices in the graph, i.e. with the root vertex at the top and dependencies branching out toward the bottom. The currently inspected vertex is called the *focal vertex* and is shown highlighted at all times, along with its immediate neighbourhood. The focal node is the main indicator of the position in the diagnostic. In addition to this view, a secondary window provides a strongly "zoomed out" view of the same graph, allowing the user to observe the global structure in the wider neighbourhood of the focal vertex. The scope of exploration may be restricted according to some criterion of interest.

Two more views are provided by Evidence Explorer, namely the *source code view* and the *state view*. The former highlights the line of code in the source file (e.g. Java) that the current state in the model corresponds to. Depending on the model check toolset used, this view may not always be available, since it requires the toolchain to keep track of which state in the LTS corresponds to which line

of code. The state view lists properties of the focal state, such as the value of source code variables in that state.

The user can use three operations to restrict the navigation of the diagnostic:

- *Manual selection* of a subset of vertices. Those outside the selection cannot be reached in the evidence exploration. This allows focusing on an interesting or representative section.
- *Projection* allows the user to focus on vertices with certain properties. For example, if the user is interested in a subformula of the specification, the subformula can be selected. This causes all vertices annotated with this subformula to be highlighted, and is known as a *formula projection*.
  A second type is a *state projection*, where vertices are highlighted based on their state properties. In addition to selection based on propositions that hold in a state, Evidence Explorer also allows vertices to be selected based on their position in the graph. This allows the user to select those states lying on the shortest path from root to focal node.
- *Grouping* partitions the graph. This allows the user to apply insight into the model by structuring the diagnostic. The diagnostic can afterwards be navigated by jumping between groups instead of individual vertices. Grouping can be done manually, or through referencing vertex attributes as in projections. Grouping can be nested, creating a hierarchy.

The mu-calculus specification is shown in its own window. What is actually shown is the Formula dependency graph of the specification (see Definition 5.29). For easy visual identification, extensive styling is used:

- The top-level operator of a subformula determines the shape of a node, e.g. a square for the box operator, a circle-plus for disjunctions, and a double circle for fixpoints;
- The binding scope of fixpoint operators divide the vertices into groups, which are coloured differently;
- The truth value of a formula vertex determines the border colour of a node;
- When a formula does not occur in a diagnostic, the colour of the corresponding vertex in the formula graph is set to transparent, so that the user immediately sees that the formula is vacuous (see Section 6.2. Vacuity Detection).

Use of the tool is exemplified by two screenshots. These screenshots show a stage in the model check process of (an abstract model of) a Java Virtual Machine component. Two specifications are checked separately, the first `deadlock` (Figure 6.1) and the second `livemon` (Figure 6.2). The former property holds, as there is no deadlock in the system. The model checker thus has to visit every state in the system, leading to a diagnostic of maximal size (note the branching structure in the graph). The latter property is violated, so a counterexample exists. This counterexample is mostly linear with cycles at regular points due to nesting of fixpoints in the property (note the linear structure of the graph).

Evidence Explorer demonstrates the advantages of a powerful tool for the visualisation and interpretation of any model checking diagnostic that conforms to the form of the Generic diagnostic graph. The tool was designed to be extensible, making it easy to define new views and methods for manipulation. For example, a new view might be developed to colour states in a transition system according to which subformulas hold. Unfortunately, at the time of writing, the source nor any executable of Evidence Explorer is publically available [Chechik, Garfunkel 05].

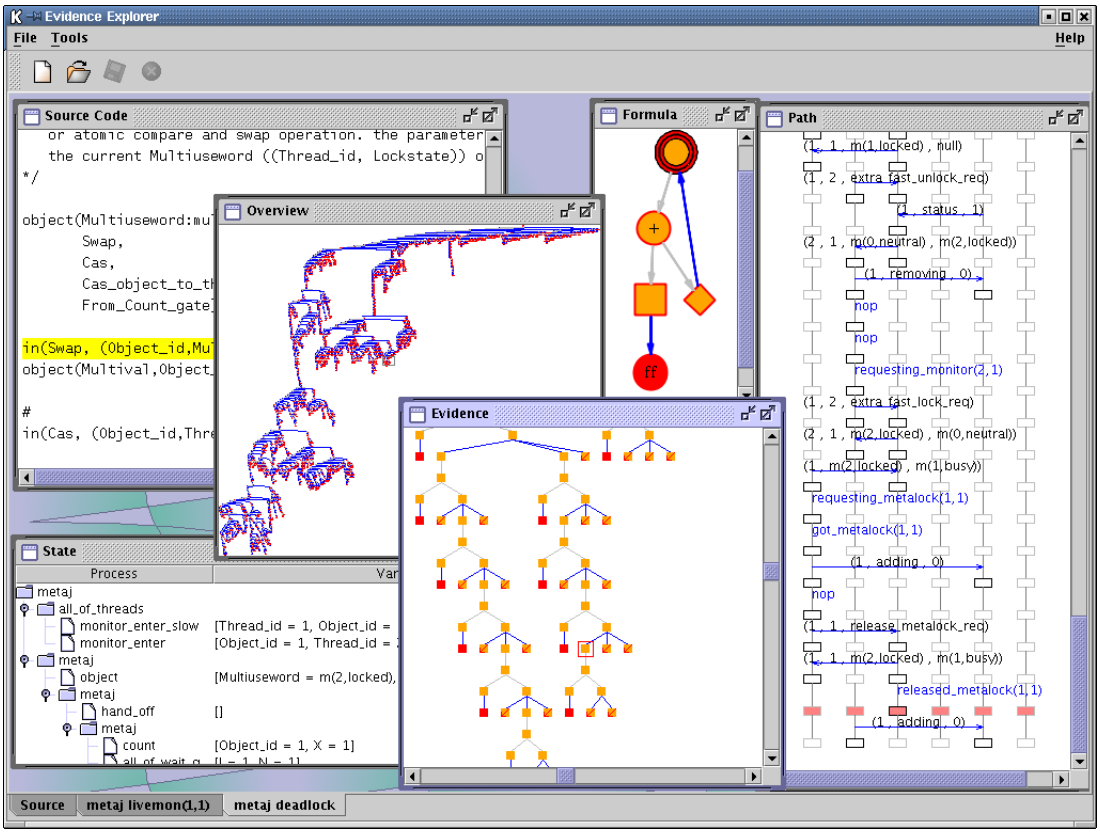


Fig. 6.1. *A counterexample for the* `deadlock` *property.*

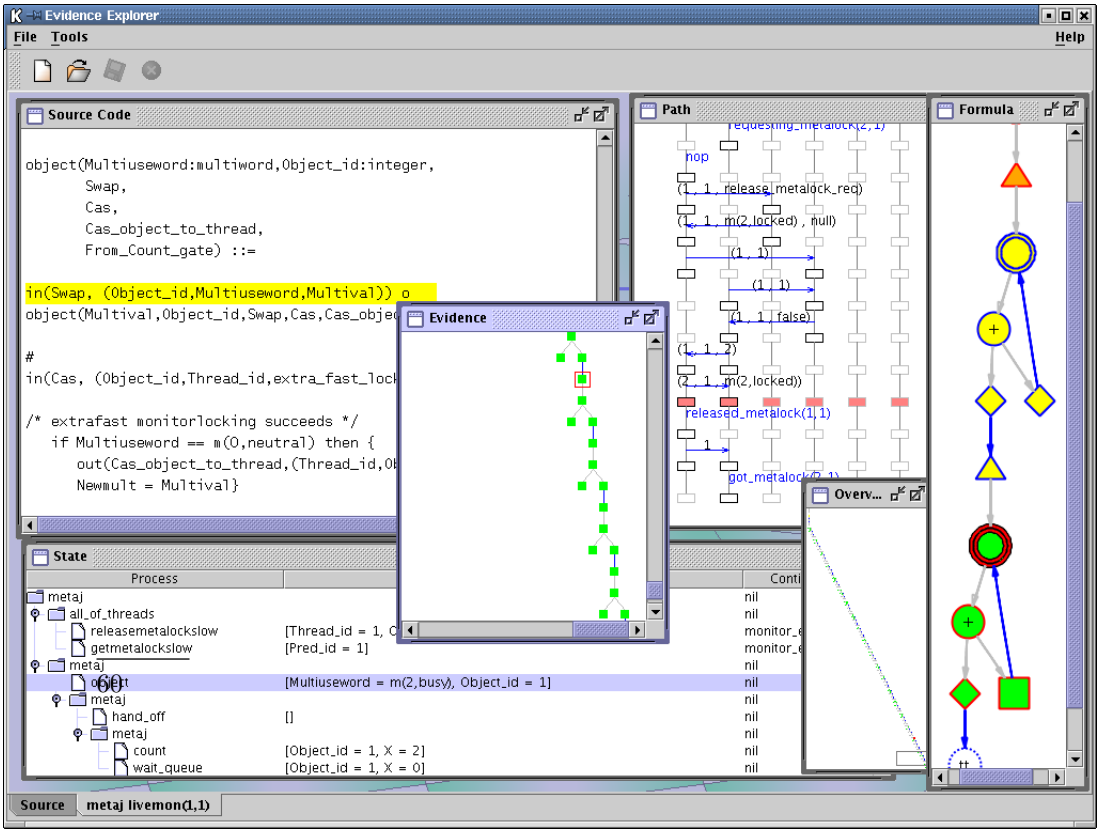


Fig. 6.2. *A counterexample for the* `livemon` *property.*

7 The Diagnostic Graph

7.1 Requirements

In Chapter 4, we reviewed the model check process. The transition system under investigation is combined with a specification, yielding a Boolean equation system which encodes the model checking problem. The BES is then solved, resulting in a *yes* or *no* answer to whether the specification holds for the model. We now explain how the model check process can be augmented to deliver not only a Boolean result, but also a valuable diagnostic. If a diagnostic is to be useful, it has to meet three main criteria [Clarke et al. 02]:

- *Completeness*: The class of diagnostic should be complete for the modal mu-calculus, i.e. each violation of a specification is witnessed by a suitable diagnostic. Vacuous satisfaction should be indicated as a special case (see Section 6.2: Vacuity Detection).
- *Intelligibility*: The diagnostic structure should be simple and specific enough to be analysed by human engineers, possibly with the aid of automated tools and suitable annotations (see Section 6.3: Presentation of Diagnostics).
- *Efficiency*: There should exist efficient (possibly symbolic) algorithms for generating and manipulating diagnostics of this class.

In an earlier chapter, we reviewed different classes of diagnostic. All classes satisfied some of these requirements, but could also be seen to suffer from various drawbacks, such as their lacking coverage of $L\mu$ or the fact that they do not focus on user-friendly presentation. The type of diagnosic developed in this chapter, the Diagnostic graph, takes elements from the studied classes to fulfill all requirements in full. The correspondences with the relevant classes in Chapter 5 will be pointed out as the Diagnostic graph is introduced.

The fundamental element in our definition of a diagnostic is the structure graph of a Boolean equation system. As discussed previously, the structure graph shows the interdependencies and nesting structure of an equation system. Evaluating the dependencies of a variable in the equation system is a natural way to find the chain of causality that led to the result for that variable. However, in a previous example of the Generic diagnostic graph (Example 5.32) we saw the drawbacks this method can have: due to the way in which mu-calculus formulas are mapped onto the LTS, the information that can be extracted from a Boolean equation system is coarse-grain. Details, such as which clause provided the path to success or failure, may not be available. This is in contrast to a tableau, in which each atomic step is meticulously represented in the proof tree. In this chapter, we will see how a model check process that uses Boolean equation system can be augmented so as to provide this type of fine-grain information to the user, permitting a maximal degree of understanding and analysis.

Thanks to the increased level of detail of the diagnostic, imposed by these augmentations, it can no longer be called generic. It is specifically tailored to

the use of Boolean equation systems, but more importantly, the extra information allows us to draw conclusions beyond those that we could from a Generic diagnostic graph. For these reasons, we drop the "generic" nomer, so that this type of diagnostic is designated *Diagnostic graph*. The new conlusions we can draw from it give rise to other types of diagnostic. Several classes that we saw in Chapter 5, such as linear paths and reduced labelled transition systems, can be readily extracted from a Diagnostic graph. These are the subject of sections 7.7 and 7.8, respectively.

The final sections of this chapter, including 7.7 and 7.8, demonstrate how the information contained in a Diagnostic graph can be presented to the end user. The most rudimentary way of inspecting the diagnostic is to browse the raw Diagnostic graph. For large, real-world problems, this is best carried out using a visualisation tool such as Evidence Explorer (see Section 6.3). In principle, diagnostics for smaller cases with a limited size can also be explored manually. The intelligibility of the diagnostic in this "low-level," vertex-by-vertex exploration is an important metric for its intelligibility in general (the second requirement above). The other half of the coin is how well this understanding scales to diagnostics of much greater size.

Even on this low level, inspection of the Diagnostic graph takes place only via a dedicated presentational layer. This layer sits between the data and the user, formatting the data in a user-friendly manner and translating user input and directions to operations on the data. The presentational layer, in combination with the other classes that can be extracted from the Diagnostic graph, forms the user-facing part of the diagnostic.

7.2 Incorporation in the Model Check Design Cycle

Ordinarily, when encoding the model checking problem into a BES, some information about the original formula and transition system is abstracted from. This information is not relevant to solving the equation system (or, therefore, to the model check result). However, if we wish to present the diagnostic in terms with which the user is familiar (i.e. the $L\mu$ formula and the transition system), we need to keep track of some of this information that is normally discarded.

Generally speaking, there are two ways to do this. One option is to retrieve this information post-hoc, i.e. after the model check is complete. This involves relating sections of the Boolean equation system to sections of the specification and LTS. However, this is not necessarily possible for arbitrary BES. We shall see below that doing this requires some constraints on the structure of a Boolean equation system, but is without loss of generality in terms of the model or the mu-calculus specification. After the model check is complete, the diagnostic can be constructed by combining information from the data structures that make up the model check process. This is illustrated by the squiggly arrows in Figure 7.1.

A second option for constructing the diagnostic is by storing the required extra information in a suitably augmented Boolean equation system. In this case, some elements of the BES are annotated, indicating to which parts of

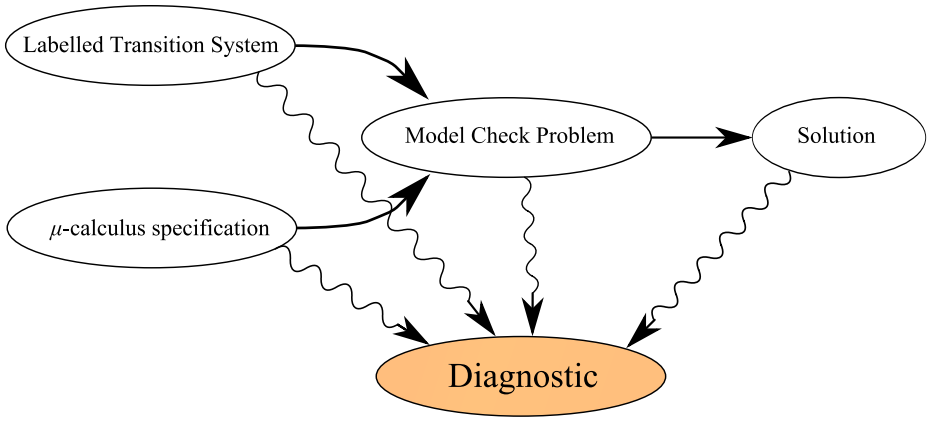


Fig. 7.1. *Various elements in the model check process can contribute to the diagnostic.*

the specification and model they relate. This allows the diagnostic to be built using only information contained in the BES (in addition to its solution). Normally, the equation system is the output of a mapping function **E** that combines the transition system and specification into one (this function was defined in Section 4.1). The BES can be derived directly and straightforwardly from the specification and model using this function. But in the equation system, these origins are not explicit: we can tell very little, given a BES, about the structure of the specification or the shape of the transition system it was derived from. Of course, it is exactly these in terms of which we want to present any user-facing diagnostic. To enable us to relate a BES back to these ingredients, we augment the mapping function to output a suitably annotated BES. An annotation in this context means simply that elements of the equation system will carry extra "labels." In an object-oriented programming environment, an annotation might be implemented as an extra data field in a class.

The extended mapping function that produces a BES with the necessary annotations is named **F**, as pictured in Figure 7.2, where the presence of anno-

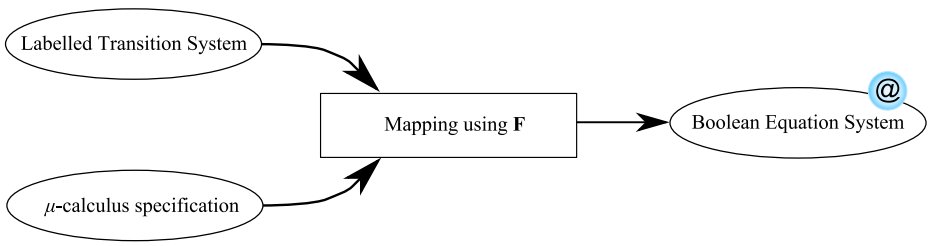


Fig. 7.2. *Annotating the Boolean equation system.*

tations is indicated by the @-symbol. This type of Boolean equation system is an extension of plain BES, so it can be solved by an unmodified model checker, which simply ignores the annotations. Note that if the model check is carried out using parity games, the equation system will first be converted into a game graph. As we shall see below, the Diagnostic graph is very similar to a game (structure) graph, so much of the data structure can be re-used. Doing so will mean that the annotations from the BES will have to be carried over to the structure graph; this can be formalised by using a modified set of SOS rules (recall the use of SOS rules to generate the game graph from a BES from Section 3.5).

The processes depicted and described above are of course simplified; in particular, they omit any simplification or rewriting steps of the BES. The game graph will commonly be *normalised* before play begins, potentially introducing more vertices (and with that, bound variables). Simplification strategies may do the opposite and remove vertices and edges. Such rewriting techniques are seen here as computational optimisations, not affecting the Diagnostic graph. However, the Diagnostic graph relies on finding "chains of causality" that led to the outcome for the variable of interest. Where simplifications omit sections of the BES, they cripple the power of the diagnostic. It is therefore crucial that the solution of a simplified (or otherwise optimised) equation system can be related back to a solution of the original. As an extreme example, if the solver only returns the truth value of the requested fixpoint variable (say X_i; usually the most shallow variable X_0), there is far too little information for a meaningful diagnostic. To analyse the dependencies of X_i adequately, many if not all of their truth values have to be known. Missing information will result in an incomplete diagnostic.

Note that this does not conflict with the use of local solvers. Recall that local solvers do not in general compute a truth value for each bound variable, but stop as soon as the value for the requested variable is known. For example, given an equation $\mu X_i = \varphi \wedge \psi$, a local solver will not evaluate ψ if it has already determined that $\varphi = \mathsf{false}$. Although the value for all dependent variables may not be known, there will always be at least one chain of causality that led to the result for X_i; in this case via φ. The value of ψ is not known, so even if it would have contributed to the diagnostic, i.e. when it had also been false, that part of the diagnostic is rendered inexplorable by the user. In case this is of concern, the value of these vertices should be described as a special "not evaluated" case which comes in addition to $\mathsf{true}, \mathsf{false}$, and should be indicated as such to the user.

At this point, we have our Diagnostic graph and a solution for the BES. Now that the diagnostic data is known, we would like to return it to the user. However, the Diagnostic graph is intimately related to the BES. Earlier we pointed out that the fact that our model checking tool uses Boolean equation systems internally should be concealed from the user. This apparent conflict is resolved by separating the concept of a diagnostic into a static, underlying data structure, and user-facing diagnostics that are generated based on this data. The Diagnos-

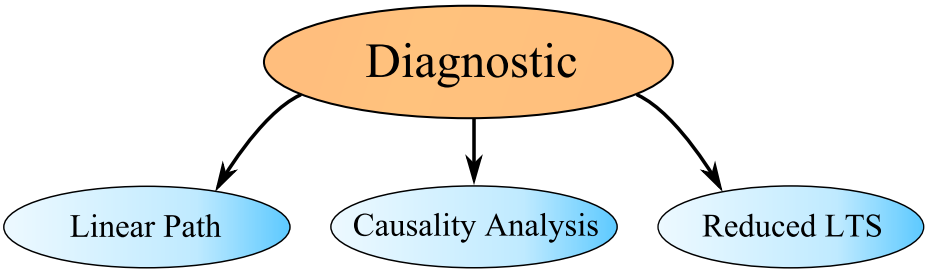


Fig. 7.3. *The master diagnostic data structure can give rise to various types of* user-facing *diagnostic.*

tic graph data structure is available at the end of the model check operation, but is not returned to the user as such. Instead, it remains the underlying structure for future exploration and generation of user-facing diagnostics. We already read about a presentational layer that formats the graph in a user-friendly, intuitive manner, while also allowing operations such as "zooming in" on a specific section of the graph (see Section 6.3: Presentation and Visualisation). A layer of this sort enables a dynamic exploration process that allows the user to interactively investigate the causal chains that led to the model check outcome. In addition, different classes of diagnostic such as linear paths and reduced labelled transition systems can be generated based on the Diagnostic graph using appropriate algorithms. These are the subject of sections 7.7 and 7.8, respectively. In this way, a single Diagnostic graph can actually give rise to a multitude of different ultimate, user-facing diagnostics (depicted in Figure 7.3).

7.3 The Diagnostic Graph

State-Formula Annotations

We saw in Section 5.9 that the Generic diagnostic graph can be generated by performing dependency analysis on a Boolean equation system. The Generic diagnostic graph is essentially the structure graph of the BES, extended with a state-formula tuple on each vertex. In a structure graph, vertices correspond to bound variables in the BES, which in turn are related to the model and specification. This relation is made explicit by adding a tuple $\langle s, \varphi \rangle$ to each vertex, where $s \in S$ is a state in the labelled transition system and $\varphi \in \mathsf{cl}(\Phi)$ is a subexpression of the mu-calculus specification. The tuple captures the semantics of a BES variable, as stated earlier in Theorem 4.1; briefly, a BES variable X_i can be said to encode the validity of φ in state s_i, so that $s_i \models \sigma X.\ \varphi \Leftrightarrow X_i = \mathsf{true}$, where φ is the subexpression of the specification that is bound by σX. We will now formalise how the formula annotation can be obtained for each vertex.

The function form was defined earlier for mu-calculus variables (see Definition 2.4). It is first extended to accept bound BES variables $\mathcal{X}$ in addition to mu-calculus variables $\tilde{\mathcal{X}}$. The function name is "overloaded" to prevent clutter;

it will be clear from the variable type of the parameter which implementation is intended (compare e.g. $\mathsf{form}(X)$ and $\mathsf{form}(X_{42})$). The following definition makes use of the fact that mu-calculus variables retain their name when mapped to BES variables. The BES variable name is resolved to that of the original mu-calculus variable by simply stripping the underscore. If this similarity in naming is not available, the BES mapping can be straightforwardly modified so as to retain a relation between each BES variable and the mu-calculus variable it was derived from.

Definition 7.1 (Subformula Retrieval (BES variable)). *The function* $\mathsf{form}: L\mu(\tilde{\mathcal{X}}) \times \mathcal{X} \mapsto L\mu(\tilde{\mathcal{X}})$ *retrieves the mu-calculus expression that falls under the scope of the fixpoint operator that binds the mu-calculus variable associated with the given bound BES variable.*

$$\mathsf{form}(\varphi,\ X_i) \quad = \quad \mathsf{form}(\varphi,\ X)$$

□

In a BES mapping, the granularity of formula annotations is limited by the number of fixpoints. This is a consequence of how the mapping function **E** works: it generates a BES with variables $X_0, \ldots, X_{|S|-1}$ for each fixpoint variable X in the mu-calculus specification. Assume for a moment that the expression bound by σX gives rise to an equation system in simple form. As defined above, the formula annotation associated with vertex X_i is $\mathsf{form}(X_i)$. This variable may depend on right-hand side constants and other variables. If the user tries to trace the origin of a failure, at some point these dependencies may need to be inspected. Given a dependency $X_i \to Y_i$, it is not immedately clear what the source of this relation was. It may have arisen because, in the formula for X, there was a recursion to Y; because of a nested fixpoint $\sigma' Y$ or because of a modal operator followed by Y (and in case of multiple candidate modal operators: which?) Likewise, for a dependency on a Boolean constant, the source could have been any atomic property or some other expression. If the assumption on simple form is lifted, even more possibilities arise.

Motivating the Model Check Outcome

The purpose of the Diagnostic graph is to analyse the causal relationship between state-formula tuples. The Boolean outcome of the model check corresponds to the tuple $\langle s_0,\ \Phi \rangle$. The motivation for this outcome is given in terms of other tuples by analysing their interdependencies: colloquially speaking "$s \not\models \varphi$ because its dependency $s' \not\models \psi$." Clearly, a motivation of this sort can only explain a false outcome in terms of dependencies that are also false; a true dependency cannot contribute to a false outcome. The converse is also true: a true outcome can only be motivated in terms of true dependencies. This is thanks to the lack of negation in our mu-calculus grammar, and implies that the only vertices of interest in the

Diagnostic graph are those that have the same truth value as the initial vertex. The others are not relevant for the diagnostic and can be ignored or pruned.[1]

In the definition of the Diagnostic graph that follows, vertices with either truth value are preserved. The value is not inspected until the generation of a user-facing diagnostic. Preserving all vertices allows a maximum degree of freedom for the user in exploring the diagnostic. An unexpected model check outcome may be due to a complicated set of factors, leading to unexpected truth values for various parts of the model: some parts that were expected to hold do not, and some parts that were expected to fail may unexpectedly hold. For example, given two mutually exclusive properties, the "wrong one" may be found to hold in a certain section of the model. Inspecting why the "wrong" property holds may consist of a witness search for this property. This allows a user to skip straight to a vertex of interest, regardless of its truth value, and perform a counterexample or witness analysis based on the truth value of that selected initial vertex. We will therefore defer inspection of truth values to the user-facing diagnostic extraction algorithms.

Indefinite recursion through a fixpoint can also be grounds for the success or failure of the model check. In the absence of a direct dependency on true or false (which occurs for instance due to a property not holding or the lack of an outgoing transition in some state), the possibility of recursion through a minimal fixpoint will lead to failure, while recursion through a maximal fixpoint will lead to success.

The potential for indefinite recursion corresponds to the existence of a cycle in the Diagnostic graph. This in turn implies the existence of a cycle $s \to \ldots \to s$ in the labelled transition system and a cyclic mu-calculus dependency $\varphi \rightsquigarrow^+ \varphi$. The fixpoint recursed indefinitely is the most shallow on the formula annotations in the cycle. We saw this before in Example 5.32, where there was a cycle $Y_0 \to X_1 \to Y_1$. Both X and Y are bound fixpoint variables, but X is the shallowest of the two, so this is the fixpoint that is recursed indefinitely. We say that the cycle in the (Generic) Diagnostic graph is *dominated* by X.

Definition 7.2 (Domination). *Let $\langle s_{(i)}, \varphi_i \rangle \to \ldots \to \langle s_{(n)}, \varphi_n \rangle$ be a cycle in the (Generic) Diagnostic graph, i.e. $s_{(i)} = s_{(n)}$ and $\varphi_i = \varphi_n$. Then the cycle is said to be σ-*dominated *if the outermost (shallowest) fixpoint with respect to Φ occurring on $\varphi_i, \ldots, \varphi_n$ is σ.* □

Above, we pointed out that vertices with a truth value different from the initial vertex are to be ignored. This is because the diagnostic should not contain elements that are irrelevant to the model check outcome. Some cycles can also be irrelevant. Indefinite recursion through a *maximal* fixpoint could not have

[1] Note that if the use of negation *is* supported by the model checker, the story becomes slightly more complex, as the failure of an expression $\neg\varphi$ can be motivated in terms of the success of φ. This means that the counterexample search for $\neg\varphi$ may consist in part of a witness search for φ.

contributed to the *failure* of the model check (and vice versa). ν-dominated cycles should therefore have no part in counterexamples, while μ-dominated cycles should have no part in witnesses.

Eliminating these cycles *a priori* is in general not possible. This is illustrated in Figure 7.4. Assume that the model check failed, so all vertices have the value false. There are two cycles in the graph: (a) highlights the least fixpoint cycle, and (c) the greatest fixpoint cycle. (b) highlights a path that leads directly to failure, without any cycles. Because the user is after a counterexample, the ν-cycle is irrelevant, as it did not contribute to the failure of the model check. However, trying to remove this cycle from the graph runs into trouble: we cannot eliminate v_4, because then the valid path in (b) can no longer be explored, and the edge $v_3 \rightarrow v_0$ cannot be eliminated because this would destroy the μ-cycle in the process.

Due to these problems, the strategy of *eliminating* these cycles is discarded in favour of *avoiding* them. This will necessarily take place during the generation of user-diagnostics from the Diagnostic graph, so we will come back to this issue later on.

A High Level of Detail

One of the requirements on our diagnostic is intelligibility. To achieve this goal, detailed information should be provided about the precise path of causality. The desired level of detail is akin to tableaux, which capture every atomic evaluation step of the mu-calculus specification in combination with the LTS. This level of detail was not enforced in the Generic diagnostic graph, owing to its generic nature. In the definition of a Generic diagnostic graph, the requirement was that an edge between two vertices implies the existence of a transitive mu-calculus dependency between the formula annotations on the respective vertices. This is now made stricter: every single mu-calculus dependency is required to be included in the Diagnostic graph. This not only ensures the desired level of detail, but

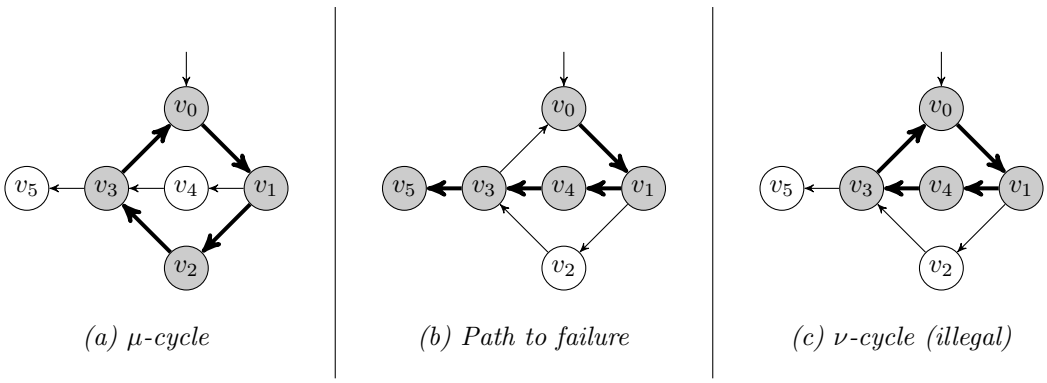


Fig. 7.4. *Permissible paths through the Diagnostic graph for a failed model check.*

also that the mu-calculus expression that was at the source of a dependency can be recovered. Consider for example a dependency $X_i \to Y_i$ and the mu-calculus formula $[a]\,Y \vee Y$. The dependency could have resulted from either disjunct, but by enforcing that each mu-calculus dependency is included in the Diagnostic graph, this ambiguity can now be resolved. After all, the formula has two immediate dependencies: one on $[a]\,Y$, and another on Y. Had the dependency on Y_i resulted from the former, then the vertex Y_i would have been annotated with the disjunct $[a]\,Y$. Instead, it is annotated with $\mathsf{form}(Y_i) = \mathsf{form}(Y)$, so this dependency is concluded to have arisen due to the latter disjunct Y.

Definition 7.3 (Diagnostic Graph). *Let $\mathcal{E}$ be the BES encoding the model check problem of verifying Φ over $\mathcal{M}$. Let η be a solution environment for $\mathcal{E}$. Then the* Diagnostic graph *is a tuple $\mathcal{D}_{\mathcal{E}} = \langle V,\, \Delta,\, \eta,\, @\rangle$ where V is a set of vertices, $\Delta \subseteq V \times V$ is a set of directed edges and $@ : V \mapsto S \times \mathsf{cl}(\Phi)$ is a* diagnostic annotation function *such that:*

- *There are special vertices* true *and* $\mathsf{false} \in V$ *which do not have annotations;*
- *There is a distinct initial vertex $X_0 \in V$ such that $@(X_0) = \langle s_0, \Phi\rangle$;*
- *Every mu-calculus dependency is included in the graph. That is, for any vertex $v \in V$ with $@(v) = \langle s,\, \varphi\rangle$ the following holds: for each ψ such that $\varphi \rightsquigarrow_{\Phi} \psi$, either:*
  - *There exists a vertex $v' \in V$ with $@(v') = \langle s',\, \psi\rangle$ such that $v \to \ldots \to v'$; or*
  - *The previous point does not hold, and $v \to \mathsf{true}$ or $v \to \mathsf{false}$.*

where $v \to v'$ has been used as shorthand for $\langle v,\, v'\rangle \in \Delta$. □

In the following, the notation $v@\langle s,\, \varphi\rangle$ will be used for $@(v) = \langle s,\, \varphi\rangle$. It will also be combined with other statements, e.g. the notation $v@\langle s,\, \varphi\rangle \in V$ will mean $v \in V \wedge @(v) = \langle s,\, \varphi\rangle$.

Generating the Diagnostic Graph

The annotation function @ has been specified as a partial function. This accommodates vertices for true and false, but more importantly is related to the existence of unranked vertices. Recall that only equation systems in simple form have a dependency graph; equation systems in general have a structure graph. The reasons for this were explained in Section 3.5. Briefly, a formula such as $X_i \wedge (Y_j \vee Z_k)$ results in an additional (unranked) vertex for the nested expression $Y_j \vee Z_k$ in the structure graph. Vertices like these are problematic for both the state and formula annotation. The state annotation can be straightforwardly derived from a vertex X_i, where it is simply s_i, but what the state annotation should be for a vertex $Y_j \vee Z_k$ is not clear. The same goes for the formula annotation: for a vertex X_i it is simply $\mathsf{form}(X_i)$, but again for $Y_j \vee Z_k$ it is not clear.

Ideally, in this example, the state annotation for unranked vertices would be that of the fixpoint which ultimately binds it. However, an unranked vertex may

have more than one incoming edge. For example, if the BES contains equations $(\mu X_i = (X_i \wedge Y_j) \vee Z_k)(\nu Y_j = Q \vee (X_i \wedge Y_j))$ then the vertex $X_i \wedge Y_j$ has incoming edges from both X_i and Y_j. Resolving the state annotation without further information is impossible here. Likewise, for the formula annotation, the formula associated with $X_i \wedge Y_j$ would ideally be $X \wedge Y$, which is expected to be a subformula of μX in the mu-calculus specification. However, as explained above, this is not necessarily the case, because the mapping function $\mathbf{E}$ is not injective: a different expression, say $X \wedge [a]\, Y$, may also give rise to the same BES expression.

A strategy for resolving these ambiguities is required. One solution is to accommodate the necessary extra information by annotating Boolean equation systems. In this strategy, the required information is injected into the BES by using a modified version of the mapping function $\mathbf{E}$. The extra information is then carried over to the Diagnostic graph using suitable SOS rules. Finally, the function @ for finding the state and formula annotation of a vertex should be changed to make use of the additional information. This strategy is the subject of Section 7.5.

First, a simpler method is used to achieve the desired result. The problems with unranked vertices are sidestepped by using only Boolean equation systems in simple form. Although this may sound restrictive, we will see that this is without loss of generality. This is the subject of the following section.

7.4 Effectless Fixpoints for Extra Detail

The goal of the Diagnostic graph is to capture a large amount of detail, which can then be used to support the model check outcome. In a previous example (Example 5.32) we saw that the Generic diagnostic graph associated with a BES can be quite lacking in detail, in other words, can be very coarse-grain. This low level of detail makes fault tracing more difficult. Consider the mu-calculus fragment $(Q \wedge [a]\, Y) \vee (\neg Q \wedge \langle a \rangle\, Y)$ and assume a single a-transition exists $s_i \xrightarrow{a} s_j$. Then the BES expression for this fragment in s_i is of the form Y_j. This expression bears no information about whether the first or the second disjunct holds in s_i. The ambiguity is a direct consequence of how the mu-calculus formula is mapped using the $\mathbf{E}$-function, which, as discussed earlier, is not injective. The Diagnostic graph proposes to provide more detail, but how this extra detail should be extracted from the BES is not yet clear at this point.

In this section, a method will be introduced that provides the required extra detail. In order to capture each atomic derivation step (i.e. each step associated with a single mu-calculus dependency), a trick is used. Note that each fixpoint variable in the mu-calculus specification is mapped to $|S|$ bound variables in the BES. For each of these variables, finding the associated state and formula annotation is easy: the state annotation for a variable X_i is simply s_i, while the formula annotation is given by $\mathsf{form}(X_i)$. The level of detail that is "accessible" in this way is limited by the number of fixpoints in the specification. This presents a simple solution for increasing the level of detail found in the BES

(and by extension the Diagnostic graph): simply place additional fixpoints in the specification. This is exactly the approach taken here. Sufficient (effectless) fixpoints are inserted to ensure that each mu-calculus dependency is represented by one or more edges in the graph.

This approach turns out to yield Boolean equation systems in simple form, so it solves two problems at once. In the first place, this approach will yield the desired level of detail. In the second, we no longer need to deal with finding state-formula annotations for unranked vertices, because the fact that a BES is in simple form implies that each vertex in the associated structure graph that has successors is ranked [Keiren, Reniers, Willemse 10].

Grammatical Restrictions

The details of inserting additional fixpoints into the mu-calculus specification are now formalised by introducing some restrictions on the grammar of mu-calculus formulas. The presence of fixpoint operators will be enforced in several places in an expression. These restrictions lead to formulas where each clause is preceded by a fixpoint. These freshly introduced fixpoints may be "effectless," meaning that the fixpoint variable does not occur in the expression that is bound. The introduction of effectless fixpoints is without loss of generality in terms of the model or the mu-calculus specification, because effectless fixpoints do not change the semantics of a formula (Lemma 2.3).

Definition 7.4 (Restricted Syntax of $L\mu$). *A modal mu-calculus formula in restricted form is given by $\boldsymbol{\Phi_\sigma}$ as follows:*

$$\begin{array}{lcl} \boldsymbol{\Phi_\sigma} & ::= & \sigma X.\ \boldsymbol{\Phi} \\ \boldsymbol{\Phi} & ::= & \mathsf{true} \mid \mathsf{false} \mid Q \mid \neg Q \mid X \mid \boldsymbol{\Phi_\sigma} \mid \\ & & \quad [a]\,\boldsymbol{\Phi_\sigma} \mid \langle a\rangle\,\boldsymbol{\Phi_\sigma} \mid \boldsymbol{\Phi_\sigma} \wedge \ldots \wedge \boldsymbol{\Phi_\sigma} \mid \boldsymbol{\Phi_\sigma} \vee \ldots \vee \boldsymbol{\Phi_\sigma} \end{array}$$

where $Q \in \mathcal{Q}$ is an atomic proposition and $X \in \tilde{\mathcal{X}}$ is a fixpoint variable. □

Theorem 7.5 (Restricted Grammar Induces BES in Simple Form). *Mu-calculus formulas given by the restricted grammar $\boldsymbol{\Phi_\sigma}$ induce Boolean equation systems in simple form. This is proven by induction on formulas.*

Base cases: *The equation systems given by $\mathbf{E}(Q)$ and $\mathbf{E}(X)$ are in simple form. The expressions given by $\mathbf{E}_i(Q)$ and $\mathbf{E}_i(X)$ are in simple form.*
Induction hypotheses: *(IH1): $\mathbf{E}(\Phi)$ gives an equation system in simple form. (IH2): $\mathbf{E}_i(\Phi)$ gives an expression in simple form.*
To prove: *Any derived BES is in simple form (by case distinction on the shape of Φ, which follows the restricted grammar $\boldsymbol{\Phi_\sigma}$):*

- $\mathbf{E}(\sigma X.\ \Phi) = (\sigma X_i = \mathbf{E}_i(\Phi))\mathbf{E}(\Phi)$ *is in simple form, since $\mathbf{E}_i(\Phi)$ gives an expression in simple form (IH2), so the equation will be in simple form; and $\mathbf{E}(\Phi)$ is in simple form (IH1).*
- $\mathbf{E}(\Phi_1 \wedge \ldots \wedge \Phi_k) = \mathbf{E}(\Phi_1) \cdots \mathbf{E}(\Phi_k)$ *is in simple form (IH1 on all conjuncts).*

- $\mathbf{E}([a]\,\Phi) = \mathbf{E}(\Phi)$ *is in simple form (IH1).*
- $\mathbf{E}_i(\Phi_1 \wedge \ldots \wedge \Phi_k) = \mathbf{E}_i(\Phi_1) \wedge \ldots \wedge \mathbf{E}_i(\Phi_k) = \mathbf{E}_i(\sigma X.\, \varphi_1) \wedge \ldots \wedge \mathbf{E}_i(\sigma Y.\, \varphi_k) = X_i \wedge \ldots \wedge Y_i$ *is in simple form.*
- $\mathbf{E}_i([a]\,\Phi) = \bigwedge_{s_i \xrightarrow{a} s_j} \{\mathbf{E}_j(\Phi) \mid s_i \xrightarrow{a} s_j\} = \bigwedge_{s_i \xrightarrow{a} s_j} \{\mathbf{E}_j(\sigma X.\, \varphi) \mid s_i \xrightarrow{a} s_j\} = \bigwedge_{s_i \xrightarrow{a} s_j} \{X_j \mid s_i \xrightarrow{a} s_j\}$ *is in simple form.*

Results for the diamond modal operator and disjunctive equations are analogous. Q.E.D. □

Additional inserted effectless fixpoints will be highlighted with a grey background for ease of reading.

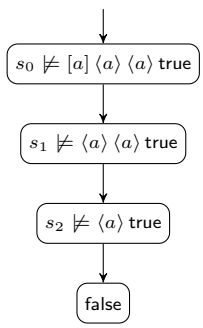
Example 7.6 (Additional Fixpoints). This short example demonstrates the advantage of introducing additional fixpoints. Consider the simple HML formula $[a]\,\langle a\rangle\,\langle a\rangle\,\mathsf{true}$ and the transition system $\rightarrow s_0 \xrightarrow{a} s_1 \xrightarrow{a} s_2$. We can encode this model checking problem in the normal way using the mapping function $\mathbf{E}$ (which requires prepending a single effectless fixpoint), resulting in the following equation system (where irrelevant equations have been omitted):

$$(\mu X_0 = \mathsf{false})$$
$$\ldots$$

The diagnostic information we can extract from this BES is minimal, as the initial variable is immediately found to be false. The motivation for this Boolean constant is nonexistent.

Now, additional effectless fixpoints are added to the specification as prescribed by the restricted grammar. The specification is in this case changed to $\mu X.\,[a]\,\mu Y.\,\langle a\rangle\,\mu Z.\,\langle a\rangle\,\mathsf{true}$ and the resulting BES is shown on the left, and the derived Diagnostic graph on the right (the effectless fixpoints X, Y, Z are omitted from the formula annotations).

$$(\mu X_0 = Y_1)$$
$$(\mu Y_1 = Z_2)$$
$$(\mu Z_2 = \mathsf{false})$$
$$\ldots$$



A lot more can now be said about the failure of the model check. The fault can be traced to its origin using the Diagnostic graph. The state annotations and formula annotations tell us that the ultimate failure was due to $s_2 \not\models \langle a \rangle \, \mathsf{true}$, in other words the lack of outgoing a-transitions in state s_2.

In addition, the annotations allow us to follow the causal path that led up to the failure, permitting the conclusion that we got up to $s_0 \xrightarrow{a} s_1 \xrightarrow{a} s_2$, but the specification ultimately failed because of the lack of an outgoing a-transition in state s_2. □

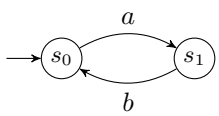
Generating the Diagnostic Graph

To turn a dependency graph into a Diagnostic graph, each vertex has to be annotated with a suitable state-formula pair $\langle s, \varphi \rangle$ with $s \in S$ and $\varphi \in \mathsf{cl}(\Phi)$. Recall that the meaning of this annotation is that the formula holds in that state if and only if a valuation function assigns the value true to that vertex. The task is thus, given a vertex X_i, to find a suitable state annotation and formula annotation. The state annotation is now trivial thanks to the fact that all vertices are ranked, so each vertex is a bound variable of the form X_i. As explained earlier, the state annotation for this vertex is simply s_i while the formula annotation is $\mathsf{form}(\Phi, X_i)$. These annotations now cover each vertex in the graph, unlike before.

The following conjecture says that combining a BES dependency graph with an annotation function as described above yields a Diagnostic graph. If this is to be the case, then each mu-calculus dependency should occur in the graph, i.e. for each mu-calculus dependency $\varphi \rightsquigarrow_\Phi \psi \in \Phi \rightsquigarrow_\Phi$ there are vertices $v@\langle s, \varphi \rangle$ and $v'@\langle s', \psi \rangle$ such that $v \to \ldots \to v'$.

Conjecture 7.7 (Diagnostic Graph via Effectless Fixpoints). *Let $\mathcal{E}$ be the BES encoding the model check problem of verifying Φ over $\mathcal{M}$ where Φ conforms to the restricted grammar $\boldsymbol{\Phi_\sigma}$, let η be a solution environment for $\mathcal{E}$ and let $\mathcal{G}_\mathcal{E} = \langle V, \Delta \rangle$ be the dependency graph of $\mathcal{E}$.*
The annotation mapping @ is defined by $@(X_i) = \langle s_i, \mathsf{form}(\Phi, X_i) \rangle$.
Then $\mathcal{D}_\mathcal{E} = \langle V, \Delta, \eta, @ \rangle$ is a Diagnostic graph. □

Example 7.8 (Diagnostic Graph via Effectless Fixpoints). This continues Example 5.32.
Consider the mu-calculus formula $\Phi = \nu X. \, \mu Y. \, [a] \, X \wedge [b] \, Y$ and the following labelled transition system:



The formula Φ does not conform to the restricted grammar, so additional fixpoints are inserted yielding $\Phi_\sigma = \nu X. \, \mu Y. \, (\mu V. [a] \, \mu V'. X) \wedge (\mu W. [b] \, \mu W'. Y)$.

$$
\begin{aligned}
\nu X_0 &= Y_0 \\
\nu X_1 &= Y_1 \\
\mu Y_0 &= V_0 \wedge W_0 \\
\mu Y_1 &= V_1 \wedge W_1 \\
\mu V_0 &= V_1' \\
\mu V_1 &= \mathsf{true} \\
\mu V_0' &= X_0 \\
\mu V_1' &= X_1 \\
\mu W_0 &= \mathsf{true} \\
\mu W_1 &= W_0' \\
\mu W_0' &= Y_0 \\
\mu W_1' &= Y_1
\end{aligned}
$$

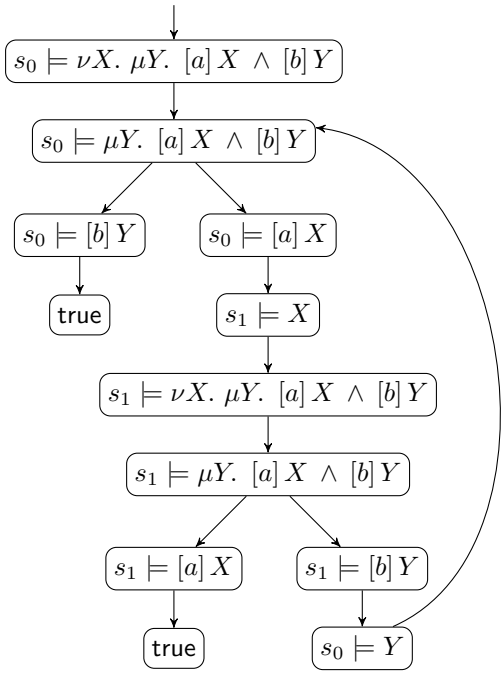


Fig. 7.5. *The Boolean equation system and Diagnostic graph for the model check problem in Example 7.8. Effectless fixpoints are hidden from vertex annotations.*

The BES generated by $\mathbf{E}(\Phi_\sigma, \mathcal{M})$ is shown in Figure 7.5 on the left, and its corresponding Diagnostic graph on the right. Note that in the formula annotations, effectless fixpoints have been omitted for readability (we will come back to this practice below).

□

Omitting effectless fixpoints when displaying the formula annotations to the user improves readability. In practice, the addition of effectless fixpoints may take place using an automated process, so that names of the inserted variables are not known to the user. If this is the case, the use of these mu-calculus variables should be kept in the background instead of being disclosed to the user, just as BES variables are never displayed. However, in case the same subexpression occurs more than once in the specification, confusion may arise as to which occurrence is intended. This can be solved straightforwardly by a suitable presentational layer, e.g. one in which the specification is shown in full, but where the relevant subexpression is highlighted (this can even be done in a text-only environment with some ASCII creativity).

Note that the restricted grammar enforces even more detail than is required: some mu-calculus dependencies $\varphi \rightsquigarrow \psi$ span multiple vertices, $v@\langle s,\, \varphi\rangle \rightarrow \ldots \rightarrow v'@\langle s',\, \psi\rangle$. It is thus safe to go back and make the restricted grammar somewhat more lenient, e.g. by also allowing expressions such as $\langle a\rangle\, \mathsf{true}$, $[a]\, \mathsf{false}$ and $(a)X$. The specification from the previous example could then take on the form $\Phi'_\sigma = \nu X.\ \mu Y.\ (\mu V.\, [a]\, X) \wedge (\mu W.\, [b]\, Y)$, resulting in a one-to-one correspondence: there would be a mu-calculus dependency $\varphi \rightsquigarrow \psi$ if and only if there are two connected vertices $v@\langle s,\, \varphi\rangle \rightarrow v'@\langle s',\, \psi\rangle$. For the sake of consistency, the restricted grammar as defined earlier will be used as-is in subsequent sections.

7.5 BES Annotations for Extra Detail

Inserting additional fixpoints to yield more detail has the obvious disadvantage of increasing the size of the equation system. Each additional fixpoint operator will yield $|S|$ equations, so the growth due to this method will be considerable. The increase in BES size corresponds to an increased running time and memory use for the model check algorithm. The exact growth depends on the algorithm used, but tends to be polynomial in the size of the BES because the number of priorities does not change (the extra fixpoints are, after all, effectless).

Nevertheless, even a polynomial growth in time and space can be prohibitive due to the potentially very large state space of real-world models. Therefore, we introduce an alternative approach to obtaining the Diagnostic graph in this section. This approach uses annotations (or "labels") on elements of the Boolean equation system instead of additional fixpoints in the specification, so that the the grammatical restriction on mu-calculus specifications may be lifted. This implies two things: first, the number of vertices and edges in the structure graph will be the same as for a vanilla model check (as opposed to containing many more vertices for the effectless fixpoints method). Secondly, the equation systems encoding the model check problem can no longer be assumed to be in simple form. As we saw in the previous section, this complicates finding the state-formula annotation for each vertex. To obtain this essential Diagnostic graph information from an unrestricted BES, the BES annotations are used.

Recall that the state-formula annotation on vertices in the Diagnostic graph allow us to relate the diagnostic information to the user in familiar terms, i.e. the model and mu-calculus specification. In the present scheme, these state-formula annotations will be derived from annotations on elements of the BES. To permit this, BES annotations record how BES expressions were derived: from which state in the model and from which subexpression of the specification. This is somewhat akin to a parse tree. Derivation here means generating a BES $\mathcal{E}$ based on a model $\mathcal{M}$ and specification Φ, i.e. $\mathcal{E} = \mathbf{E}(\Phi,\, \mathcal{M})$. This mapping function $\mathbf{E}$ and its shortcomings for our purposes have been discussed earlier. To amend these shortcomings, the mapping function will be augmented to yield a BES with suitable annotations. The augmented function is called $\mathbf{F}$: $L\mu \times \mathbb{M} \rightarrow \mathcal{E}^{@}$, where the @ superscript indicates the presence of annotations on BES of this type. The function $\mathbf{F}$ is defined in Figure 7.6. The dot $\cdot$ indicates sequential composition, so that an ordered sequence of formulas is written $\varphi_1 \cdot \varphi_2 \cdot \ldots$

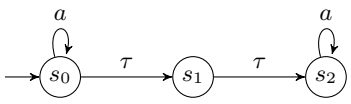
$$
\begin{array}{rcl}
{}^{\Phi}\mathbf{F}(\mathrm{Q}) & = & \epsilon \\
{}^{\Phi}\mathbf{F}(X) & = & \epsilon \\
{}^{\Phi}\mathbf{F}(\varphi \wedge \ldots \wedge \psi) & = & {}^{\Phi \cdot \varphi \wedge \ldots \wedge \psi}\mathbf{F}(\varphi) \ \ldots \ {}^{\Phi \cdot \varphi \wedge \ldots \wedge \psi}\mathbf{F}(\psi) \\
{}^{\Phi}\mathbf{F}(\varphi \vee \ldots \vee \psi) & = & {}^{\Phi \cdot \varphi \vee \ldots \vee \psi}\mathbf{F}(\varphi) \ \ldots \ {}^{\Phi \cdot \varphi \vee \ldots \vee \psi}\mathbf{F}(\psi) \\
{}^{\Phi}\mathbf{F}([a]\,\varphi) & = & {}^{\Phi \cdot [a]\,\varphi}\mathbf{F}(\varphi) \\
{}^{\Phi}\mathbf{F}(\langle a\rangle\,\varphi) & = & {}^{\Phi \cdot \langle a\rangle\,\varphi}\mathbf{F}(\varphi) \\
{}^{\Phi}\mathbf{F}(\sigma X.\varphi) & = & (\sigma X_0 = {}^{\langle s_0,\ \Phi \cdot \sigma X.\ \varphi\rangle}\mathbf{F}_0(\varphi)) \\
 & & \vdots \\
 & & (\sigma X_{n-1} = {}^{\langle s_{n-1},\ \Phi \cdot \sigma X.\ \varphi\rangle}\mathbf{F}_{\text{n-1}}(\varphi))\ {}^{\Phi \cdot \sigma X.\ \varphi}\mathbf{F}(\varphi) \\
\\
{}^{\langle s_i,\ \Phi\rangle}\mathbf{F}_{\text{i}}(Q) & = & \begin{cases} {}^{\langle s_i,\ \Phi \cdot Q\rangle}\mathsf{true} & \text{if } s_i \in \mathcal{V}(Q) \\ {}^{\langle s_i,\ \Phi \cdot Q\rangle}\mathsf{false} & \text{otherwise} \end{cases} \\
{}^{\langle s_i,\ \Phi\rangle}\mathbf{F}_{\text{i}}(X) & = & {}^{\langle s_i,\ \Phi \cdot X\rangle}X_i \\
{}^{\langle s_i,\ \Phi\rangle}\mathbf{F}_{\text{i}}(\varphi \wedge \ldots \wedge \psi) & = & {}^{\langle s_i,\ \Phi \cdot \varphi \wedge \ldots \wedge \psi\rangle}\big({}^{\langle s_i,\ \Phi \cdot \varphi \wedge \ldots \wedge \psi\rangle}\mathbf{F}_{\text{i}}(\varphi) \\
 & & \qquad \wedge \ldots \wedge \\
 & & \qquad {}^{\langle s_i,\ \Phi \cdot \varphi \wedge \ldots \wedge \psi\rangle}\mathbf{F}_{\text{i}}(\psi)\big) \\
{}^{\langle s_i,\ \Phi\rangle}\mathbf{F}_{\text{i}}(\varphi \vee \ldots \vee \psi) & = & {}^{\langle s_i,\ \Phi \cdot \varphi \vee \ldots \vee \psi\rangle}\big({}^{\langle s_i,\ \Phi \cdot \varphi \vee \ldots \vee \psi\rangle}\mathbf{F}_{\text{i}}(\varphi) \\
 & & \qquad \vee \ldots \vee \\
 & & \qquad {}^{\langle s_i,\ \Phi \cdot \varphi \vee \ldots \vee \psi\rangle}\mathbf{F}_{\text{i}}(\psi)\big) \\
{}^{\langle s_i,\ \Phi\rangle}\mathbf{F}_{\text{i}}([a]\,\varphi) & = & \begin{cases} {}^{\langle s_i,\ \Phi \cdot [a]\,\varphi\rangle}\mathsf{true} & \text{if } \nexists_j.\ s_i \xrightarrow{a} s_j \in \delta \\ {}^{\langle s_i,\ \Phi \cdot [a]\,\varphi\rangle}\Big(\bigwedge\limits_{s_i \xrightarrow{a} s_j} {}^{\langle s_i,\ \Phi \cdot [a]\,\varphi\rangle}\mathbf{F}_{\text{j}}(\varphi)\Big) & \text{otherwise} \end{cases} \\
{}^{\langle s_i,\ \Phi\rangle}\mathbf{F}_{\text{i}}(\langle a\rangle\,\varphi) & = & \begin{cases} {}^{\langle s_i,\ \Phi \cdot \langle a\rangle\,\varphi\rangle}\mathsf{false} & \text{if } \nexists_j.\ s_i \xrightarrow{a} s_j \in \delta \\ {}^{\langle s_i,\ \Phi \cdot \langle a\rangle\,\varphi\rangle}\Big(\bigvee\limits_{s_i \xrightarrow{a} s_j} {}^{\langle s_i,\ \Phi \cdot \langle a\rangle\,\varphi\rangle}\mathbf{F}_{\text{j}}(\varphi)\Big) & \text{otherwise} \end{cases} \\
\\
{}^{\langle s_i,\ \Phi\rangle}\mathbf{F}_{\text{i}}(\sigma X.\ \varphi) & = & {}^{\langle s_i,\ \Phi \cdot \sigma X.\varphi\rangle}X_i
\end{array}
$$

Fig. 7.6. *The augmented mapping function* ***F*** *yields an annotated BES.*

It can be seen that state-formula annotations are recorded as the derivation proceeds. An annotation on a BES formula ψ, say ${}^{\langle s,\ \varphi\rangle}\psi$ has the same meaning

as before: $\psi = \mathsf{true} \Leftrightarrow s \models \varphi$. The difficulty is now in the fact that right-hand side expressions are no longer in simple form, so annotations may appear nested as in ${}^{\langle s,\, \varphi\rangle}(^{\langle s',\, \varphi'\rangle}\psi)$.

Example 7.9 (Annotated BES). Consider again the mu-calculus specification $\mu X.\ ([a]\, X \wedge [\tau]\, X) \vee (\nu Y.\ \langle a\rangle\, \mathsf{true} \wedge [a]\, Y \wedge [\tau]\, Y)$ and the LTS from Example 5.4, repeated here:



The following abbreviations will be used:

$$\begin{aligned}
\alpha &= [a]\, X \wedge [\tau]\, X \\
\beta &= \nu Y.\ \gamma \\
\gamma &= \langle a\rangle\, \mathsf{true} \wedge [a]\, Y \wedge [\tau]\, Y
\end{aligned}$$

The annotated BES $\mathcal{E}^{@} = \mathbf{F}(\Phi,\ \mathcal{M})$ encoding this model check problem is as follows:

$$\begin{aligned}
\mu X_0 = {} & {}^{\langle s_0,\, \Phi\cdot\alpha\vee\beta\rangle}(\\
& {}^{\langle s_0,\, \Phi\cdot\alpha\vee\beta\cdot\alpha\rangle}(\\
& \quad {}^{\langle s_0,\, \Phi\cdot\alpha\vee\beta\cdot\alpha\cdot[a]\, X\rangle}(^{\langle s_0,\, \Phi\cdot\alpha\vee\beta\cdot\alpha\cdot[a]\, X\cdot X\rangle} X_0) \\
& \quad \wedge\ {}^{\langle s_0,\, \Phi\cdot\alpha\vee\beta\cdot\alpha\cdot[\tau]\, X\rangle}(^{\langle s_1,\, \Phi\cdot\alpha\vee\beta\cdot\alpha\cdot[\tau]\, X\cdot X\rangle} X_1)) \\
& \vee\ {}^{\langle s_0,\, \Phi\cdot\alpha\vee\beta\cdot\beta\rangle} Y_0) \\
\mu X_1 = {} & {}^{\langle s_1,\, \Phi\cdot\alpha\vee\beta\rangle}(\\
& {}^{\langle s_1,\, \Phi\cdot\alpha\vee\beta\cdot\alpha\rangle}(\\
& \quad {}^{\langle s_1,\, \Phi\cdot\alpha\vee\beta\cdot\alpha\cdot[a]\, X\rangle}\mathsf{true} \\
& \quad \wedge\ {}^{\langle s_1,\, \Phi\cdot\alpha\vee\beta\cdot\alpha\cdot[\tau]\, X\rangle}(^{\langle s_2,\, \Phi\cdot\alpha\vee\beta\cdot\alpha\cdot[\tau]\, X\cdot X\rangle} X_2)) \\
& \vee\ {}^{\langle s_1,\, \Phi\cdot\alpha\vee\beta\cdot\beta\rangle} Y_1) \\
\mu X_2 = {} & {}^{\langle s_2,\, \Phi\cdot\alpha\vee\beta\rangle}(\\
& {}^{\langle s_2,\, \Phi\cdot\alpha\vee\beta\cdot\alpha\rangle}(\\
& \quad {}^{\langle s_2,\, \Phi\cdot\alpha\vee\beta\cdot\alpha\cdot[a]\, X\rangle}(^{\langle s_2,\, \Phi\cdot\alpha\vee\beta\cdot\alpha\cdot[a]\, X\cdot X\rangle} X_2) \\
& \quad \wedge\ {}^{\langle s_2,\, \Phi\cdot\alpha\vee\beta\cdot\alpha\cdot[\tau]\, X\rangle}\mathsf{true}) \\
& \vee\ {}^{\langle s_2,\, \Phi\cdot\alpha\vee\beta\cdot\beta\rangle} Y_2) \\
\mu Y_0 = {} & {}^{\langle s_0,\, \Phi\cdot\alpha\vee\beta\cdot\beta\cdot\gamma\rangle}(\\
& {}^{\langle s_0,\, \Phi\cdot\alpha\vee\beta\cdot\beta\cdot\gamma\cdot\langle a\rangle\, \mathsf{true}\rangle}\mathsf{true} \\
& {}^{\langle s_0,\, \Phi\cdot\alpha\vee\beta\cdot\beta\cdot\gamma\cdot[a]\, Y\rangle}(^{\langle s_0,\, \Phi\cdot\alpha\vee\beta\cdot\beta\cdot\gamma\cdot[a]\, Y\cdot Y\rangle} Y_0) \\
& {}^{\langle s_0,\, \Phi\cdot\alpha\vee\beta\cdot\beta\cdot\gamma\cdot[\tau]\, Y\rangle}(^{\langle s_1,\, \Phi\cdot\alpha\vee\beta\cdot\beta\cdot\gamma\cdot[\tau]\, Y\cdot Y\rangle} Y_1))
\end{aligned}$$

$$
\begin{aligned}
\mu Y_1 = {} & {}^{\langle s_1,\ \Phi \cdot \alpha \vee \beta \cdot \beta \cdot \gamma \rangle}(\\
& {}^{\langle s_1,\ \Phi \cdot \alpha \vee \beta \cdot \beta \cdot \gamma \cdot \langle a \rangle\, \mathsf{true} \rangle}\, \mathsf{false} \\
& {}^{\langle s_1,\ \Phi \cdot \alpha \vee \beta \cdot \beta \cdot \gamma \cdot [a]\, Y \rangle}\, \mathsf{true} \\
& {}^{\langle s_1,\ \Phi \cdot \alpha \vee \beta \cdot \beta \cdot \gamma \cdot [\tau]\, Y \rangle}\, ({}^{\langle s_2,\ \Phi \cdot \alpha \vee \beta \cdot \beta \cdot \gamma \cdot [\tau]\, Y \cdot Y \rangle}\, Y_2)) \\
\mu Y_2 = {} & {}^{\langle s_2,\ \Phi \cdot \alpha \vee \beta \cdot \beta \cdot \gamma \rangle}(\\
& {}^{\langle s_2,\ \Phi \cdot \alpha \vee \beta \cdot \beta \cdot \gamma \cdot \langle a \rangle\, \mathsf{true} \rangle}\, \mathsf{true} \\
& {}^{\langle s_2,\ \Phi \cdot \alpha \vee \beta \cdot \beta \cdot \gamma \cdot [a]\, Y \rangle}\, ({}^{\langle s_2,\ \Phi \cdot \alpha \vee \beta \cdot \beta \cdot \gamma \cdot [a]\, Y \cdot Y \rangle}\, Y_2) \\
& {}^{\langle s_2,\ \Phi \cdot \alpha \vee \beta \cdot \beta \cdot \gamma \cdot [\tau]\, Y \rangle}\, \mathsf{true})
\end{aligned}
$$

□

We now again face the problem of finding state-formula annotations for each vertex in the structure graph. Because the equation systems are no longer in simple form, there may be (unranked) vertices in the graph that correspond to nested right-hand side expressions. For those vertices X_i that are ranked, the function $\mathsf{form}(X_i)$ can be used to obtain the formula annotation, while the state annotation is simply s_i (this is the same as for the effectless fixpoints method; see Conjecture 7.7).

Finding the state-formula annotation for unranked vertices that correspond to nested expressions is more difficult. Recall the example given earlier, where a BES contains the following equations.

$$
\begin{aligned}
\mu X_i &= (X_i \wedge Y_j) \vee Z_k \\
\nu Y_j &= Q \vee (X_i \wedge Y_j)
\end{aligned}
$$

In the first equation, the state annotation for the nested expression $(X_i \wedge Y_j)$ should be s_i and in the second s_j. This is because of the purpose of the state annotation: the truth value of the nested expression encodes whether a certain subexpression of the mu-calculus specification holds in the state given by the state annotation (see Definition 5.31: Generic Diagnostic Graph). Clearly, it is impossible to give a unique state annotation for this nested expression, because a different one is required for each occurrence.

The relevant vertex $X_i \wedge Y_j$ will have incoming edges from both X_i and Y_j. The desired state-formula annotation depends on which edge was used to reach it. This offers a solution to the problem: place the BES annotations on *edges*, and use these to derive the annotation of connected *vertices*. If a vertex is reachable via multiple edges, each edge will get a distinct annotation. In the given example, there would be edges $X_i \xrightarrow{\langle s_i,\, \varphi \rangle} (X_i \wedge Y_j)$ and $Y_i \xrightarrow{\langle s_j,\, \psi \rangle} (X_i \wedge Y_j)$ for nonspecified formula annotations φ, ψ. The state-formula annotation for vertex $(X_i \wedge Y_j)$ is finally given by the annotation on the edge by which it was reached.

A complication with placing the relevant annotations on edges is linking outgoing to incoming edges. Recall that the purpose of the Diagnostic graph is to include every mu-calculus dependency (see Definition 7.3). Therefore, if a vertex X_i was reached via an edge carrying the annotation $@\langle s,\, \varphi \rangle$, and there is

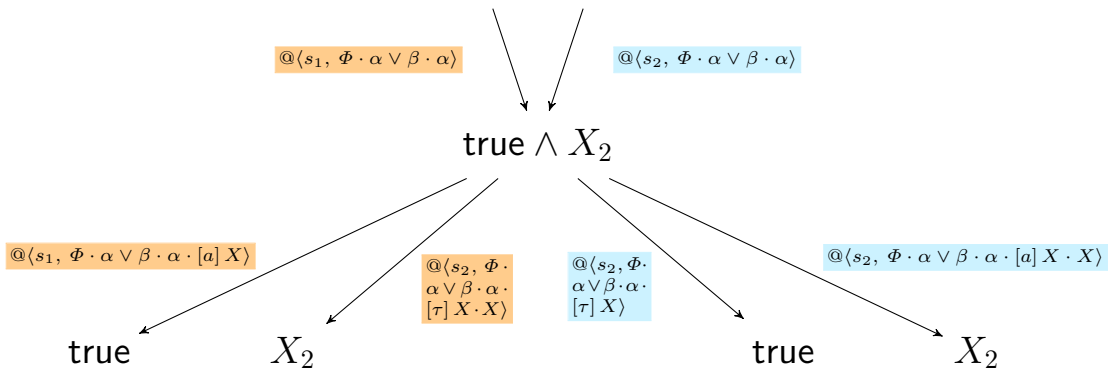


Fig. 7.7. *Part of the Diagnostic graph with edge annotations from Example 7.9. Exploration of the graph follows mu-calculus dependencies. When coming in from an edge with a blue annotation, the only outgoing edges that may be traversed are those that also have a blue annotation.*

a dependency $\varphi \rightsquigarrow \psi$, we would expect any outgoing edge to have an annotation of the form $@\langle s', \varphi \cdot \psi\rangle$. However, this does not necessarily hold if there are two incoming edges with different annotations φ, φ' because each would have its own set of dependencies. Therefore, it may be illegal to traverse certain outgoing edges, based on the edge by which the vertex was reached. This is illustrated in Figure 7.7, where it is illegal to traverse an outgoing edge with a different colour from the incoming edge.

In the next sections, various algorithms for exploring and extracting information from the Diagnostic graph will be discussed. These algorithms can relatively easily be modified to accept edge annotations as an intermediate for vertex annotations. The Diagnostic graph (with vertex annotations, as defined in Definition 7.3) is then reconstruced on-the-fly as the algorithm explores the graph with edge annotations (by whatever search strategy). To prevent traversal of edges with the wrong annotation, the "parse tree" formed by the BES annotations can be used as a constraint.

Placing state-formula annotations on edges rather than on vertices will require an alternative (incompatible) definition of the Diagnostic graph and associated differences in the exploration algorithms to follow. The remainder of this chapter will therefore be based on the original definition of the Diagnostic graph (Definition 7.3).

The effectless fixpoints method requires only minimal changes to the model checking toolchain and is compatible with other software that can manipulate this kind of diagnostic (such as Evidence Explorer). In a strongly optimised

toolchain, where running time and memory use are of crucial importance, the BES annotations method may provide a better candidate, but will require more extensive modification of the software.

7.6 Exploration of the Diagnostic Graph

After the Diagnostic graph has been obtained from the BES, it is immediately ready for user exploration. Recall the distinction we made between the Diagnostic graph as a data structure for diagnostic generation, and the user-facing diagnostics that can be generated based on the graph. This distinction shields the user from having to deal directly with the underlying Boolean equation system. In the following sections, we will see other exploration methods which yield user diagnostics that have no visible relation at all to the Boolean equation system, namely path extraction (7.7) and minimised LTS extraction (7.8). In this section, we still keep relatively close to the BES, as we interpret dependencies between variables as causal relations.

The general idea for exploring a graph of this form has already been described in Section 6.3. These exploration methods are applicable to all diagnostics that conform to the Generic diagnostic graph format: briefly, those where the vertices correspond to tuples $\langle s, \varphi \rangle$, directed edges encode dependency and where a truth function on vertices asserts that $s \models \varphi$ if and only if the truth value of that vertex is true. The Diagnostic graph is a specific type of diagnostic that conforms to the Generic diagnostic graph format, and is thus suitable for this type of exploration.

In establishing the requirements for a suitable type of diagnostic, we hinted at the possibility of manual exploration of the graph. It should be clear that this is rarely a suitable approach in general for trying to comprehend a diagnostic. The sheer size of real-world models prohibits this. However, we also pointed out that how well this manual exploration contributes to understanding of the diagnostic is an important metric for its usefulness in general. Specialised exploration tools exist for navigating a large evidence, which significantly automate (part of) the process (such as Evidence Explorer). The primary advantage of these tools is the ability to focus on a small section of the diagnostic, in other words "zooming in" on the salient and interesting parts of the diagnostic. Once the relevant section has been identified, local analysis still tends to proceed manually, with extensive examination of the fine-grain local details. How well the diagnostic performs on small "toy" problems is therefore very much relevant to its performance on real-world cases.

Manual, interactive exploration of the Diagnostic graph is very similar to playing a parity game. An interactive diagnostic session of this sort, with application to game-based model checking, was previously discussed in Section 5.6, where we saw an example of its implementation in the Edinburgh Concurrency Workbench. The format of the dialogue between the user and the model checker there is quite similar to the format here, due to the equivalences in the underlying data structure (both the game and the Diagnostic graph can be reduced to Generic diagnostic graphs).

Recall that parity games are played on the dependency (game) graph of a Boolean equation system, of which the Diagnostic graph is an extension. A token is placed on a vertex, indicating the current position. The player owning the vertex on which the token lies may then select an outgoing edge, after which the token is moved along the edge to the connected vertex. There are several termination conditions, such as reaching the vertex true or false, and the possibility of infinite recursion.

The user diagnostic is, through the interactive process, generated "on the fly," that is, the sequence in the Diagnostic graph is shaped by the choices the user makes as the dialogue unfolds. Different choices may lead to different user diagnostics; a single Diagnostic graph can thus give rise to a multitude of distinct user-facing diagnostics.

A straightforward process or algorithm is shown below for computer-guided exploration of the Diagnostic graph. The focus in this section will be on the user making choices where possible; the next section will focus on automating these choices. A choice exists whenever more than one dependency was responsible for the failure, e.g. two failing conjuncts (we will assume a negative outcome in the rest of this section for brevity; witness search is dual). During exploration, the algorithm keeps track of the traversed path in the LTS. This can be used as an exemplar of failure. The next section will elaborate on the function and merit of this path.

The exploration process is simple. We begin at the initial vertex, i.e. that vertex for which the diagnostic annotation is $\langle s_0, \Phi \rangle$. Depending on the shape of the formula annotation, a number of decisions can be made as to which section of the diagnostic to explore next. When there is only one option, the selection can be made automatically.

The termination conditions correspond to termination conditions for the model check: either reaching a blatantly false configuration, such as a property that does not hold or the absence of an outgoing transition, or the possibility of indefinite recursion through a minimal fixpoint. Indefinite recursion is detected by keeping track of visited vertices, and terminating when the same vertex is visited twice. Upon reaching a termination condition, a user diagnostic has been generated, so the exploration process has finished. An alternative possibility is to backtrack to the previous choice point, where the user is then allowed to choose a different sequence. This would allow the user to explore every possibility in a single session. This is supported by the algorithm in [Stevens, Stirling 98]. To keep track of choices that have already been explored, their algorithm keeps track of these visited choices at each vertex in the traversed sequence. Although the exploration algorithm given below can be extended in this way without much difficulty, this extension has not been investigated because the user should already have applied insight towards exploring the most interesting causal chain; the user is not interested in an exhaustive demonstration of the winning strategy of the model checker.

The sequence taken through the Diagnostic graph can be one of two general forms: linear, leading up to the vertex true or false, or lasso-shaped, due to indefinite recursion of a fixpoint. Note the following: detection of recursion relies on checking whether a vertex has been visited at some earlier point in the dialogue. Due to this detection method, recursion may be detected in a vertex associated with another fixpoint. If the recursion of σX (which dominates the cycle) is detected in $v@\langle s,\ \varphi\rangle$ then the formula annotation φ is not necessarilly of the form $\sigma X \ldots$ If desired, the sequence can be automatically completed to the vertex with the formula $\sigma X \ldots$ by duplicating part of the lasso.

Earlier, we discussed the trouble with eliminating irrelevant cycles. In the present case of a counterexample, ν-cycles should be avoided, i.e. the user should not be permitted to explore a sequence that goes through a ν-cycle, because recursion through this fixpoint did not contribute to the failure of the model check. We saw that it is not in general possible to remove these cycles *a priori*. Instead, exploring them will be avoided: the user will be disallowed from making choices that lead to an undesired cycle. In other words, if the sequence through the Diagnostic graph is in the shape of a lasso, the dominating fixpoint on that cycle is not allowed to be ν (in the present case of a counterexample) or μ (in the case of a witness).

Unfortunately, avoiding these cycles can be computationally expensive. The goal is to prevent the user from making a choice at a branching point in the graph that will inevitably lead to the completion of a ν-cycle. The pseudocode

```
Inputs: current vertex v,
        possible move v → v' with η(v) = η(v'),
        exploration sequence Σ = [v_0, ..., v]

 1   func inevitableσcycle(v, v', Σ) : 𝔹
 2      if (v' has no outgoing edges)
 3         return false
 4      else if (v' occurs in Σ at position i)
 5         if (dom([v_i, ..., v]) = σ)
 6           return true
 7         else
 8           return false
 9         fi
10      else
11         return ⋀_{v'→v''} : η(v'') = η(v) ⇒
12                              inevitableσcycle(v', v'', Σ ∘ v')
13      fi
14   endfunc
```

Fig. 7.8. *Pseudocode for determining whether an exploration move $v \to v'$ will inevitably lead to the traversal of a σ-cycle.*

algorithm in Figure 7.8 answers the following question: from the current position v, will a move to v' lead inevitably to a ν-cycle? If this is the case, then that move should be excluded from the options presented to the user.

Due to the recursion on line 12, calculating the return value may take a significant amount of resources. Improving the average-case complexity may be beneficial in practice, for example by marking vertices with a direct dependency on true or false as "safe," and then transitively marking all their predecessors as safe. However, optimisations like these will not do anything for the worst-case complexity, so the *Efficiency* requirement on our diagnostic is violated.

An alternative is to relinquish avoidance, so that the user can indeed traverse the unwanted cycles. Once this is detected (which can be done in negligible time), the user is informed that a meaningless sequence was chosen in the context of this diagnostic, and the algorithm backtracks to a previous choice point. Clearly, this is confusing to the user, because it lays bare a deficiency in the exploration algorithm.

A second alternative can only be used when the model checker uses parity games, and the strategy it has computed can be inspected. The model checker has of course already computed a winning strategy that avoids these cycles. If we can somehow access this strategy, the information can be re-used for avoiding them. This is essentially the approach taken in [Stevens, Stirling 98]: the model checker first calculates a winning strategy, which is then used in playing a game against the user. The re-use of the strategy prevents the model checker from having to backtrack during an interactive session.

However, the downside here is that the strategy calculated by the model check algorithm usually consists of only one option per vertex. Recall that the strategy function $\psi_{\mathsf{Odd}} : V_{\mathsf{Odd}} \mapsto V$ decides which vertex the token is moved to based on the current vertex. For each vertex that is winning for the model checker (i.e. all vertices in V_{Odd}), there is a single edge along which the token is to be moved, even though moving along a different edge may also have resulted in a win. Ideally, this function would record all possible options that result in a win, i.e. it would map to $\mathcal{P}(V)$ instead of V. Because this is not the case, the user will not be able to control the moves by the model checker, because for each vertex V_{Odd} owned by the model checker, there is only one possible choice $\psi(V_{\mathsf{Odd}})$. This seriously impacts the ability of the user to exercise control over the diagnostic exploration. It is not immediately clear how existing parity game algorithms should be modified to produce the required extra information, but the impact such a modification would have on their running time is likely severe.

We shall leave the choice between these alternatives open here. If possible, integration with the model check algorithm will be the best solution because the allows previously computed information to be re-used. Otherwise, a choice has to be made between the other two alternatives. This choice can be left open as an invocation parameter of the diagnostic exploration tool, so that the user may enable unwanted cycle avoidance for relatively small problems, and disable it otherwise.

Definition 7.10 (Exploration Algorithm). *Let $\mathcal{E}$ be the Boolean equation system encoding the model check problem of verifying Φ over the LTS $\mathcal{T} = \langle S, s_0, \mathcal{L}, \delta \rangle$, and let $\mathcal{D}_{\mathcal{E}} = \langle V, \Delta, \eta, @ \rangle$ be its Diagnostic graph.*
A play on the Diagnostic graph is a finite sequence of vertices $\Sigma = [v_0, \ldots, v_n]$ following the rules below.
The diagnostic information displayed to the user is shown within `framed boxes.`
Assume that the model check failed, i.e. we are looking for a counterexample. The case for witness search is dual.

---

(1) **Initialise.**
Begin exploration at the initial vertex: $v_0 := X_0 @ \langle s_0, \Phi \rangle$
Initialise the LTS path $\pi = [s_0]$ and exploration sequence $\Sigma = [v_0]$.

`The model check failed. Begin causality analysis.`

(2) **Check termination conditions.**

- $v_i = \mathsf{false}$. Successful termination. Display the path leading up to the failure.

  `Path leading up to the failure: $\pi$.`

- v_i has already been visited, i.e. there is some v_{i-k} with $k > 0$ such that $v_i = v_{i-k}$. Let $v_{i-k}, \ldots, v_i$ be dominated by the fixpoint μX_α.
  Then μX_α can be recursed indefinitely. This is grounds for failure of the model check.

  `The minimal fixpoint $\sigma_\alpha X_\alpha$ can be recursed indefinitely via the path $\pi$.`

  Successful termination.

(3) **Case distinction on the form of the formula annotation.**
The vertex currently containing the token is $v_i @ \langle s, \varphi \rangle$.

`Expression $\varphi$ does not hold in state $s$.`

Case distinction on the form of φ:

- **Atomic Property.** φ is of the form Q. There is only one dependency $v_i \to \mathsf{false}$. Select $v_{i+1} = \mathsf{false}$.

  `The property $Q$ does not hold in state $s$.`

- **Conjunction.**

  `$\varphi$ is conjunctive. At least one dependency does not hold.`

Allow the user to select one of the failing dependencies.
`Please select a failing conjunct to investigate:`

For each dependency $v_i \to v'@\langle s,\, \psi\rangle$ such that $\eta(v') = \mathsf{false}$, present a choice:

• ψ

(User selects some v' for v_{i+1})

- **Disjunction.** No dependency holds. Allow the user to select one of them. (Dual to conjunction.)
- **Diamond Modality.**

  φ `is a diamond modality expression.`

  Either:

  - There is only one dependency $v_i \to \mathsf{false}$ due to the absence of suitable outgoing transitions from state s. Select $v_{i+1} = \mathsf{false}$.

    `There are no outgoing transitions in state` s `that match the modal operator.`

  - `No dependency holds.`

    Allow the user to select one of the failing dependencies.
    `Please select a failing dependency to investigate:`

    For each dependency $v_i \to v'@\langle s',\, \psi\rangle$ such that $\eta(v') = \mathsf{false}$, present a choice:

    • ψ `(in state` s'`)`

    (User selects some v' for v_{i+1})
    Let $\varphi = \langle a\rangle\, \psi$ and $v_{i+1}@\langle s',\, \psi\rangle$. Then $\pi := \pi \;\circ\; \xrightarrow{a} s'$

    `Path so far:` π

- **Box Modality.** At least one dependency does not hold. Allow the user to select one of the failing dependencies. (Dual to diamond modality.)
- **Fixpoint Operator.**

  φ `is a fixpoint expression.`

  Either:

  - There is only one dependency $v_i \to v'@\langle s,\, \psi\rangle$ and ψ is of the form $\sigma Y\ldots$

    `There is a dependency on a nested fixpoint.`

    Select the dependency $v_{i+1} = v'$

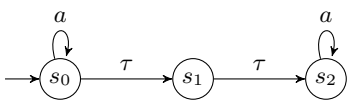
- The previous condition does not hold. Pretend that the formula annotation on the current vertex v_i was of the form ψ instead of $\sigma X.\ \psi$. Go to (2), but skip (4): do not increment i or change s.
  `Skipping the outermost fixpoint.`

- **Fixpoint Variable.**

  φ is of the form X and there is only one outgoing dependency $v \to v'@\langle s,\ \mathsf{form}(X)\rangle$. Select the dependency $v_{i+1} = v'$.

(4) **Set up variables for the next iteration.** $i := i + 1$. Set s and φ such that $@(v_i) = \langle s,\ \varphi\rangle$, and $\Sigma = \Sigma \circ v_i$. Go to (2).

Example 7.11 (Exploration Dialogue). Consider again the mu-calculus specification $\mu X.\ ([a]\,X \wedge [\tau]\,X) \vee (\nu Y.\ \langle a\rangle\,\mathsf{true} \wedge [a]\,Y \wedge [\tau]\,Y)$ and the following LTS from Example 5.4. (We will forego the details related to extra fixpoints for brevity.)



Previously, we found that no single path could by itself suffice as a full counterexample. In the dialogue between the user and the model checker (Figure 7.9), the user will follow the (by itself insufficient) path $[s_0 \xrightarrow{a} s_0 \xrightarrow{a} \ldots]$ while also receiving information about the mu-calculus expressions that cause this path to lead to failure. □

7.7 Path Extraction

A key goal for diagnostics is to provide the user with insight into the behaviour of the system. This insight may arise through a number of different methods. We have already seen how a Diagnostic graph can be explored interactively in a game-based manner, and visually using a specialised tool, which can provide insight into a potentially very large diagnostic. An alternative to this manual exploration is the automated extraction of a single path through the transition system demonstrating the failure.

Many existing model checking toolsets provide at least this type of diagnostic. Due to its simplicity and pervasiveness, it can be called a common denominator between the various model checking toolkits. Its simplicity is both a strength and

The model check failed. Begin causality analysis. $\pi = [s_0]$

(1) Expression $\varphi = \mu X.\ ([a]\,X \wedge [\tau]\,X) \vee (\nu Y.\ \langle a \rangle\,\mathsf{true} \wedge [a]\,Y \wedge [\tau]\,Y)$ does not hold in state s_0.
Skipping the outermost fixpoint.
φ is disjunctive. None of the dependencies hold.
Please select a failing disjunct to investigate:

- $[a]\,X \wedge [\tau]\,X$
- $\nu Y.\ \langle a \rangle\,\mathsf{true} \wedge [a]\,Y \wedge [\tau]\,Y$

(User selects the first disjunct)

(2) Expression $\varphi = [a]\,X \wedge [\tau]\,X$ does not hold in state s_0.
φ is conjunctive. At least one dependency does not hold.
Please select a failing conjunct to investigate:

- $[a]\,X$

(User selects the only possible conjunct)

(3) Expression $\varphi = [a]\,X$ does not hold in state s_0.
φ is a box modality expression. At least one dependency does not hold.
Please select a failing dependency to investigate:

- $\mu X.\ ([a]\,X \wedge [\tau]\,X) \vee (\nu Y.\ \langle a \rangle\,\mathsf{true} \wedge [a]\,Y \wedge [\tau]\,Y)$ (in state s_0)

(User selects the only possible dependency)

Path so far: $\pi = [s_0 \xrightarrow{a} s_0]$

(4) Expression $\varphi = \mu X.\ ([a]\,X \wedge [\tau]\,X) \vee (\nu Y.\ \langle a \rangle\,\mathsf{true} \wedge [a]\,Y \wedge [\tau]\,Y)$ does not hold in state s_0.
The minimal fixpoint σX can be recursed indefinitely via the path:
$\pi = [s_0 \xrightarrow{a} s_0]$.

Successful termination.

Fig. 7.9. *A dialogue that may ensue between the user and model checker for Example 7.11.*

a weakness. It has been demonstrated that a linear path may be insufficient to, by itself, fully demonstrate failure or success of the model check (see Example 5.4). On the other hand, a path is very easy to understand, relatively concise (again due to its simplicity) and can easily be intepreted by the user. Even when a

linear path is not technically adequate, it will often provide enough direction to the user so that, with some additional domain knowledge, the cause of the failure can be understood.

Thanks to the extra information contained in the Diagnostic graph, a linear path can be annotated with the point of failure. This essentially uses the exploration algorithm defined above as the workhorse, but omits all output except for the termination condition and of course the path. For example, failure due to a property that does not hold in a certain state is given by a path leading up to that state and the message that "property Q (or so) failed to hold in the last state in the path." Failure due to indefinite recursion of a minimal fixpoint operator would result in a path in the shape of a lasso, with the message that "failure was due to indefinite fixpoint recursion." A single path thus returned to the user should not be taken as *the* reason for failure of the model check: there may be multiple equally valid paths.

Automated Exploration

During interactive exploration, the user will make choices that lead to an interesting and nonobvious path of failure. A certain clause in the specification may be known (and intended) to fail in some state(s). Take for example two mutually exclusive properties P and Q, and a specfication of the form $(P \wedge \varphi) \vee (Q \wedge \psi)$. If P is known to fail in some area of the model, then failure of the model check will be due to a failure of the second disjunct; that is, the real or interesting failure will be due to some other chain of causality, which lies behind a different choice at some point in the dialogue. The user provides insight to steer the search in this interesting direction by informed decisions. This insight is not available when the exploration is automated, and is another limitation of linear path diagnostics in general.

An advantage of automated exploration is optimisation based on a set of criteria. This can partially compensate for the mentioned limitation by allowing the user to direct the path extraction algorithm towards potentially interesting sections. An obvious concrete criterion is for a path to be of minimal length, but in principle any constraint can be used, such as minimising the number of times a certain action label occurs on the path, or perhaps a hard constraint like "a *read*-action occurs on the path."

Optimisation for length can easily be accomplished using a breadth-first exploration of the Diagnostic graph. BFS investigates all possible choices at every vertex (so long as potential hard constraints are satisfied), and terminates when any one of these paths leads to a termination condition. This one path is then returned as the shortest. Alternatively, the search can continue, as a result of which all possible paths of any length (satisfying the hard constraints) will be computed. The one path optimising the soft criteria can then be returned. The worst-case complexity of BFS is $\mathcal{O}(|V| + |E|)$. In case variables other than the length need to be optimised, for example the number of times a certain action occurs on a path, BFS is insufficient and heuristic algorithms become an attractive option due to their running time.

7.8 Reduced LTS Extraction

When extracting a path from the Diagnostic graph, a single choice is made at each choice point. If a disjunctive expression is seen to fail, a single disjunct is selected for exploration, and a failure path will be generated according to that choice. When the purpose is instead to extract a reduced labelled transition system, information from multiple available options needs to be used, so no single choice can be made. Instead, the information resulting from a number of options is merged to form a single transition system. The goal in principle is to arrive at an LTS of minimal size (with as few states and transitions as possible), but as before, another possibility is to produce transition systems that meet hard or optimise soft criteria.

The dirty work of the exploration, by which the reduced LTS is produced, is done by the algorithm defined earlier (Definition 7.10). We shall refer back to this algorithm instead of repeating much of it here, because it already does most of what is needed, such as maintaining an internal state and checking termination conditions. To allow multiple options to be explored in one run, the entire algorithm process, including its internal state, will be *forked* (replicated) at each choice point for each option that needs to be explored. Each instance then proceeds to explore one of them, until the next choice point, where another fork occurs, or until a termination condition is reached.

For our purposes, there are two types of choice points: those that are disjunctive (due to a disjunctive expression or a diamond modal operator) and those that are conjunctive (due to a conjuntive expression or a box modal operator). As discussed earlier in Chapter 5, if a *disjunctive* formula fails, each of its disjuncts necessarily fails, so we would like an explanation for the failure of each disjunct. Dually, in case a *conjunctive* formula fails, we have the option of finding diagnostics for all its failing conjuncts, or selecting a single one from them to further investigate. The emphasis here is on finding a reduced LTS, so as few options as possible should be explored to keep the size to a minimum. Thus, for each disjunctive choice point the algorithm should fork to explore *all* options, but in case of a conjunctive node there is still the possiblity of user input (for the present case of counterexamples; the dual for witnesses is to explore all *conjunctive* options but select one from each *disjunctive* option). To ensure that separate instances make the same choice at each vertex, a global variable should keep track of visited vertices. When an instance of the exploration algorithms arrives at a vertex that another instance has already visited, it may terminate.

User input is therefore still possible, and in fact instrumental in determining the final outcome: one choice may lead to a reduced LTS of only 10 states, whereas another to an LTS orders of magnitude larger. As before, these choices can be left to the user, but can also be made by an appropriate (heuristic) algorithm.

Definition 7.12 (Reduced LTS). *Let $\mathcal{E}$ be the Boolean equation system encoding the model check problem of verifying Φ over the LTS $\mathcal{T} = \langle S, s_0, \mathcal{L}, \delta \rangle$, and let $\mathcal{D}_{\mathcal{E}} = \langle V, \Delta, \eta, @ \rangle$ be its Diagnostic graph. Explore $\mathcal{D}_{\mathcal{E}}$ according to*

the algorithm in Definition 7.10 with the additional termination condition: if the current vertex v has already been visited (i.e. $v \in \Sigma_i$ for any instance i), then terminate successfully.

When faced with a choice point:

- *If the choice is* disjunctive *and the model check* failed*, or the choice is* conjunctive *and the model check was a* success*: fork, i.e. create a separate instance for each option v;*
- *If the choice is* conjunctive *and the model check* failed*, or the choice is* disjunctive *and the model check was a* success*: select one option v (either through user input or some given strategy).*

Each instance i will terminate with a finite nonempty path π_i.

The reduced LTS *is $\mathcal{T}' = \langle S', s_0, \mathcal{L}', \delta' \rangle$ where $S' \subseteq S$, $s_0 \in S'$, $\mathcal{L}' \subseteq \mathcal{L}$ and $\delta' \subseteq \delta$ such that:*

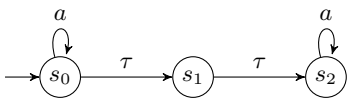
- *$s \in S'$ if s occurs in any path π_i;*
- *$l \in \mathcal{L}'$ if l occurs in any transition in any path π_i;*
- *$s \xrightarrow{a} s' \in \delta'$ if the transition $s \xrightarrow{a} s'$ occurs in any path π_i.*

□

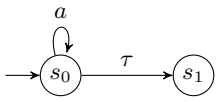
The labelled transition systems resulting from this method are *valid* as defined in Definition 5.10.

Conjecture 7.13 (Validity of Reduced LTS). *Let $\mathcal{E}$ be the Boolean equation system encoding the model check problem of verifying Φ over the LTS $\mathcal{T}$, and let $\mathcal{T}'$ be its reduced counterpart. Then $\mathcal{T} \models \Phi \iff \mathcal{T}' \models \Phi$.* □

Example 7.14 (Reduced LTS). Consider again the mu-calculus specification $\Phi = \mu X.\ ([a]\, X \wedge [\tau]\, X) \vee (\nu Y.\ \langle a \rangle\, \mathsf{true} \wedge [a]\, Y \wedge [\tau]\, Y)$ and the LTS from Example 5.4. The abbreviation $\Psi = \nu Y.\ \langle a \rangle\, \mathsf{true} \wedge [a]\, Y \wedge [\tau]\, Y$ will be used.



The Diagnostic graph associated with this model check problem is given in Figure 7.10. As discussed earlier, failure is due to the possibility of indefinite recursion through the outermost least fixpoint, so an example of failure is $[s_0, s_0, \ldots]$. This path is not a complete counterexample, because the LTS it induces consists only of s_0 and an a-loop. Compare the following reduced LTS, which has been extracted from the Diagnostic graph:



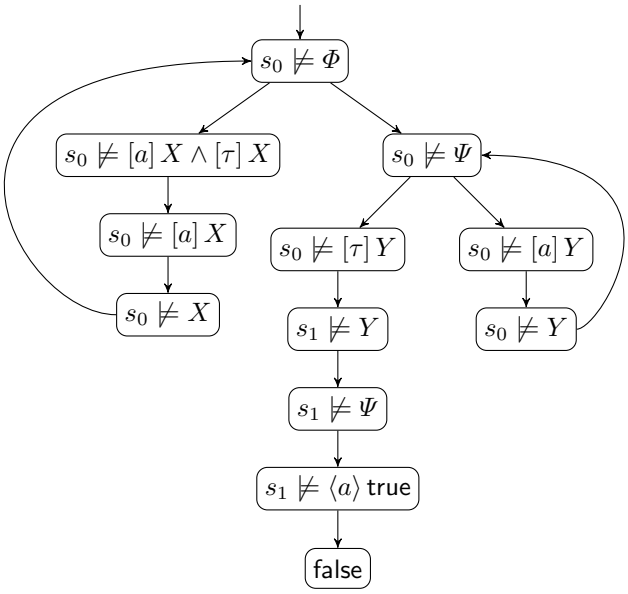
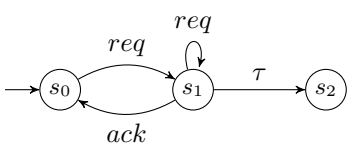


Fig. 7.10. *Diagnostic graph for Example 7.14. Vertices that have a different truth value from the initial vertex X_0 are omitted and effectless fixpoints are hidden.*

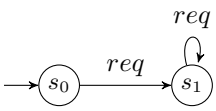
The specification fails over this reduced model (as predicted by Conjecture 7.13). Note that the unrolling of this LTS is the multi-path from Example 5.8, which was concluded to be a complete counterexample. □

Example 7.15 (Reduced LTS Alternatives). This example demonstrates how the same Diagnostic graph can give rise to two different reduced transition systems. Consider the following model, and the specification $\Phi = \mu X.\ [\mathcal{L}]\, X \wedge [req](\mu Y.\ [\overline{ack}]\, Y \wedge \langle \mathcal{L} \rangle\, \mathsf{true})$, which says that an acknowledgement should eventually occur after a request.

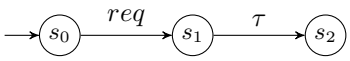


As we can see, the desired property is violated in two ways: by livelock in s_1 and by deadlock in s_2. Exploration of the Diagnostic graph based on this model check problem presents two choice points, one of which is invariant to the resulting LTS. The remaining choice is between the request loop in s_1, which violates the minimal fixpoint Y, and the path to failure in state s_2. The former will result

in the reduced LTS on the left, while the latter results in the LTS on the right. Note that the original specification fails on both reduced transition systems.



Reduced LTS showing livelock



Reduced LTS showing deadlock

□

8 Conclusion

The Diagnostic graph has been presented as a diagnostic data structure tailored to model checking using the mu-calculus and Boolean equation systems. It employs elements from various classes of diagnostics, in order to most efficiently convey diagnostic information to the end user. The type of information that is most effective in doing this is in the end based on the preference of the user, who may be helped most by a simple linear path diagnostic at one stage of development, and a detailed causality analysis at another stage.

The Diagnostic graph supports these and other "user-facing" diagnostics because it contains an abundance of detail about the model check process. This also permits future extensions, such as a state colouring according to which subformulas hold (as mentioned in Section 6.3: Presentation of Diagnostics). The Diagnostic graph serves as a central data structure, which can be generated in multiple ways (even when different model check techniques are used), and supports the derivation of myriad user-facing diagnostics.

Future Work

There are a number of conveniences that have not been incorporated in the Diagnostic graph. The regular mu-calculus was introduced as a shorthand notation. In fact, notation using regular expressions (e.g. $[\mathcal{L}]\,\varphi$) has been used throughout this work because it promotes the readability of intricate formulas. It should not be difficult to add support for regular expressions in the Diagnostic graph, as any regular formula can be rewritten to the plain mu-calculus.

Computer processes are assumed to be modelled as labelled transition systems here. This is a type of model that is almost universal, but is often generated automatically from some higher language. We already saw this in the Edinburgh Concurrency Workbench example, where states were designated with more readable names and higher-level operators such as sequential composition $(\cdot)$ and alternative composition $(+)$ were used. This is a level of representation which provides a better overview of a complex model than a (potentially very large) transition system. Therefore, the diagnostic should ideally also be in terms of this higher language. To accomplish this, there will have to be a method for relating states and transitions in the LTS back to the higher-level expressions from which they originate. This relation can then be used when presenting the diagnostic information by translating the states and transitions back into the language with which the user is familiar.

Finally, we have discussed the breadth-first search algorithm for extracing paths of minimal length from the Diagnostic graph, and only briefly touched upon heuristic algorithms. The extraction of user-facing diagnostics from the Diagnostic graph is important, because of the impact it can have on the understandability of the presented diagnostic. The user may want to obtain only paths or other user-facing diagnostics that meet a number of hard or optimise a number of soft criteria. However, computational complexity is a significant issue in this area. Heuristic algorithms are popular for this kind of optimising search,

due to their managable complexity. The right search strategy can significantly contribute to the value of any user-facing diagnostic, because it is a way for the user to apply her insight and domain knowledge of the model to limit the size and complexity of the diagnostic.

References

[Beatty, Bryant 94] D. Beatty and R. Bryant, *Formally verifiying a microprocessor using a simulation methodology*, in Design Automation Conference '94, p. 596–602.

[Beer et al. 97] I. Beer, S. Ben-David, C. Eisner and Y. Rodeh, *Efficient Detection of Vacuity in ACTL formulas*, in Proc. 9th Conference on Computer Aided Verification vol. 1254 of LNCS, p. 279–290 (1997).

[Bradfield, Stirling 06] Julian Bradfield and Colin Stirling, *Modal mu-calculi*, in P. Blackburn, J. van Benthem and F. Wolter (eds.), The Handbook of Modal Logic p. 721–756. Elsevier (2006).

[Buccafurri et al. 01] Francesco Buccafurri, Thomas Eiter, Georg Gottlob and Nicola Leone, *On ACTL Formulas Having Linear Counterexamples*, in Journal of Computer and System Sciences 62, p. 463–515 (2001).

[Chechik, Garfunkel 05] Marsha Chechik and Arie Garfunkel, *A Framework for Counterexample Generation and Exploration*, in Lecture Notes in Computer Science vol. 3442, p. 220–236 (2005).

[Clarke, Draghicescu 89] E. M. Clarke and I. A. Draghicescu, *Expressibility Results for Linear-Time and Branching-Time Logics*, in REX Workshop, vol. 354 of Lecture Notes in Computer Science (1989).

[Clarke et al. 95] E. Clarke, O. Grumberg, K. McMillan and X. Zhao, *Efficient Generation of Counterexamples and Witnesses in Symbolic Model Checking*, in *32nd Design Automation Coference (DAC 95)*, p. 427–432, San Francisco, CA, USA (1995).

[Clarke et al. 02] Edmund Clarke, Somesh Jha, Yuan Lu and Helmut Veith, *Tree-Like Counterexamples in Model Checking*, in Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science (LICS '02).

[Clarke et al. 03] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu and Helmut Veith, *Counterexample-Guided Abstraction Refinement for Symbolic Model Checking*, in Journal of the ACM, vol. 50, No. 5, September 2003, p. 752–794.

[Cleaveland 90] Rance Cleaveland, *Tableaux-Based Model Checking in the Propositional Mu-Calculus*, in Acta Informatica vol. 27, p. 725–747 (1990).

[Dong, Ramakrishnan, Smolka 03] Yifei Dong, C.R. Ramakrishnan and Scott A. Smolka, *Model Checking and Evidence Exploration*, in Proc. 10th IEEE International Conference and Workshop on the Engineering of Computer-Based Systems, p. 214–223 (2003).

[Emerson, Lei 86] E. A. Emerson and C. L. Lei, *Efficient Model Checking in Fragments of the Propositional Mu-Calculus*, in Proc. 1st IEEE LICS 267-278 (1986).

[Fischer, Ladner 79] Michael J. Fischer and Richard E. Ladner, *Propositional Dynamic Logic of Regular Programs*, in Journal of Computer and System Science, 18(2) p. 194–211 (1979).

[Gurfinkel, Chechik 03] Arie Gurfinkel and Marsha Chechik, *Proof-like Counter-Examples*, in Tools and Algorithms for the Construction and Analysis of Systems, p. 160–175 (2003).

[Groote 08] Jan Friso Groote, Jeroen Keiren, Aad Mathijssen, Bas Ploeger, Frank Stappers, Carst Tankink, Yaroslav Usenko, Muck van Weerdenburg, Wieger Wesselink, Tim Willemse and Jeroen van der Wulp, *The mCRL2 toolset*, in Proc. International Workshop on Advanced Software Development Tools and Techniques (WASDeTT 2008).

[Groote, Reniers 09] Jan Friso Groote and Michel Reniers, *Modelling and Analysis of Communicating Systems* (revision 1478), Eindhoven Technical University (2009).

[Hennessy, Milner 80] M. Hennessy and R. Milner, *On Observing Nondeterminism and Concurrency*, in Procs. ICALP 1980 Lecture Notes in Computer Science 85 p. 295–309 (1980).

[Keiren 09] Jeroen Keiren, *An Experimental Study of Algorithms and Optimisations for Parity Games, with an Application to Boolean Equation Systems* Eindhoven Technical University (2009).

[Keiren, Reniers, Willemse 10] Jeroen Keiren, Michel A. Reniers and Tim A.C. Willemse, *Structural Analysis of Boolean Equation Systems*, to appear in Transactions on Computational Logic (submitted 2010).

[Kupferman, Vardi 99] Orna Kupferman and Moshe Y. Vardi, *Vacuity Detection in Temporal Logic Model Checking*, in Proc. of the 10th IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods (1999).

[Mader 96] Angelika Mader, *Verification of Modal Properties Using Boolean Equation Systems*, Dieter Bertz Verlag (1996).

[Mateescu 00] Radu Mateescu, *Efficient Diagnostic Generation for Boolean Equation Systems*, in Lecture Notes in Computer Science 1785, p. 251–265 (2000).

[Mateescu 06] Radu Mateescu, *CAESAR_SOLVE: A Generic Library for On-The-Fly Resolution of Alternation-Free Boolean Equation Systems*, in Springer International Journal on Software Tools for Technology Transfer (STTT) 8(1), p. 37–56 (2006).

[Mateescu, Sighireanu 00] Radu Mateescu, Mihaela Sighireanu, *Efficient On-the-Fly Model-Checking for Regular Alternation-Free Mu-Calculus*, in Proc. of the 5th International Workshop on Formal Methods for Industrial Critical Systems (FMICS) Berlin, Germany (April 2000).

[McMillan 93] K.L. McMillan, *Symbolic Model Checking*, Kluwer Academic (1993).

[McMillan 94] K.L. McMillan, *Fitting Formal Methods into the Design Cycle*, in 31st ACM/IEEE Automation Conference (1994).

[Milner 80] Robin Milner, *A Calculus of Communicating Systems*, in Lecture Notes in Computer Science vol. 92, Springer (1980).

[Mousavi, Reniers, Groote 07] Mohammad Reza Mousavi, Michel A. Reniers and Jan Friso Groote, *SOS Formats and Meta-Theory: 20 Years After*, in Theoretical Computer Science, vol. 373, p. 238–272 (2007).

[Plotkin 04] Gordon D. Plotkin, *A Structural Approach to Operational Semantics*, in Tech. Rep. DAIMI FN-19, Computer Science Department, Aarhus University, Aarhus, Denmark (1981); Reprinted with corrections in J. Log. Algebr. Program. 60-61: 17-139 (2004).

[Pressman 97] Roger S. Pressman, *Software Engineering: A Practicioner's Approach (4th ed.)*, McGraw-Hill (1997).

[Stevens, Stirling 98] Perdita Stevens and Colin Stirling, *Practical Model Checking using Games*, in Lecture Notes in Computer Science vol. 1384, p. 85–101 (1998).

[Stirling 96] Colin Stirling, *Modal and Temporal Logics for Processes*, Lecture Notes in Computer Science vol. 1043, Springer Berlin /Heidelberg (1996).

[Stirling, Walker 89] Colin Stirling and David Walker, *Local Model Checking in the Modal Mu-Calculus*, in Proceedings of TAPSOFT '89, Lecture Notes in Computer Science 351, Springer-Verlag, Berlin (1989).

[Tan, Cleaveland 02] Li Tan and Rance Cleaveland, *Evidence-Based Model Checking*, in Computer-Aided Verification, p. 455–470, Springer-Verlag (2002).