Faithful Translations Among Models and Specifications in VeriTech

Orna Grumberg and Shmuel Katz
Computer Science Department
The Technion
Haifa, Israel
email: {katz,orna}@cs.technion.ac.il

May 1, 2000

Abstract

Numerous translations exist between the design notations of formal methods tools, usually between two specific notations. VeriTech is a general framework for translating among formal verification tools. The notation of each component tool is translated to a core representation, and the core is translated to each component. For any translation it is important that the translation preserves a similar modular structure of the notations for expressing models. Even more importantly, and sometimes in conflict with the first goal, it is vital that the properties true of the semantic interpretations of the source and the translated notations are closely related.

The core design language CDL is described as a collection of transitions organized into modules that encourages maintaining modular structure. Key issues in translating among models with inconsistent features are described, leading to models that are closely related, but do not always preserve either the structure or the correctness of properties in a simple way. The concept is presented of a *faithful* relation among models and families of properties true of those models. In this framework families of properties are provided with uniform transformations, in addition to the translations of the models. This framework is shown appropriate for common instances of relations among translations previously treated in an ad hoc way. Furthermore, it allows expressing connections among models where one is neither a refinement nor an abstraction of the other.

Contact author:
Shmuel Katz
Computer Science Department
The Technion
Haifa
ISRAEL
katz@cs.technion.ac.il

1 Introduction

Translations among notations for representing models and hardware designs have become common, although often there is no available documentation. In addition to translations among formal methods tools, there is increasing interest in translating from standard hardware design notations such as Verilog or VHDL (or from internal industrial notations) to existing model-checking tools, as a way to introduce formal verification at minimal cost. In this paper we present a general framework for implementing multiple translations, show the difficulties that arise during translation, and provide a theoretical basis to put such translations into a formal framework. By using this framework, it becomes clear how properties of the original model description and the translated version are related.

The VeriTech project facilitates translations of problem statements from one formal specification notation to another. A key element in the design of this project is an intermediate *core design language* denoted CDL. Each notation has compiler-like translation programs to and from CDL, thus requiring only $2n$ translations in order to achieve all of the possible n^2 relations among n different notations.

Either the VeriTech framework or direct translations among notations can be used in a variety of ways. Most obviously, a particular property to be verified can be attacked with different tools. For example, if an initial attempt to model check a temporal logic property of a system should fail because of the size of the state space, it is possible to translate either directly, or in stages first to the core representation, and then to a BDD based model checker that can handle the problem. Alternatively, the target could be a tool with a theorem-proving approach that is not sensitive to the size of the data domain, proving the desired property using inductive methods.

In addition, different properties can each be established for a system using a different verification tool, choosing the most convenient tool for each property. This should encourage using different verification tools for various aspects of the same system. For example, a propositional linear-time temporal property might be proven for a finite-state model of the system using a linear-time model checker. The system model can then be translated to the core, and subsequently to a branching-time model checker for properties of that type. From the core, it can also be translated to a language with real-time notation for verifying such properties, or to a theorem proving environment to treat infinite domains and first-order temporal properties.

At present, VeriTech includes translations between SMV[7, 5, 18], Murphi[14], Spin [12, 13] and the core design language CDL, and work is underway to incorporate, among others, PVS[21, 22], LOTOS[3], STeP[17, 1] and Petri nets[24].

The quality of translations depends on at least two key criteria: maintaining the modular structure of the notations for describing the models, and, even more crucially in the context of formal verification, guaranteeing the 'correctness' of the translations. The modular structure is important so that the text of the representation does not become unwieldy, and so that the components of the result of the translation can be recognized by someone familiar with the source of the translation. This becomes particularly important if a counterexample for a property in the target representation is to be relevant for discovering a mistake in the original representation.

In defining (and guaranteeing) 'correctness', we consider the needed relations among properties true of the original model and those that hold in the result of a translation. The relation among properties is used in both directions: we may want to 'import' versions of properties already guaranteed true of the original model into the resulting one, so they can be used in showing additional properties without being themselves reproven. Also, we may want to know that properties shown about the resulting model imply related properties in the original model. Ideally, the semantic models (e.g., the execution trees) underlying the notations would be identical, making the question trivial in either direction. However, we demonstrate that this is often impossible, and propose a broader framework of *faithful* translations among both models and properties. A translation and transformation of properties will be faithful with respect to families of properties (usually represented as classes of formulas in some temporal logic) so that if property X is true of one model, then property Y will be true of the other. We demonstrate that even when the relation between the two models is neither a classic refinement nor an abstraction, there are systematic relations that can be used to prove that subclasses of properties true for one model can be transformed to different properties that will be guaranteed true of the other.

As already noted, the issues seen here are relevant for other systems that translate among models, and in particular the SAL system under development at SRI, and the Model Checking Kit[20], as well as for individual translations among models. Such translations exist from SMV to PVS, from Murphi to PVS, from SMV to Spin, from several notations into Cospan[15], and among many other tools, including internal industrial hardware notations.

In Section 2 the CDL notation is defined, including collections of textual transitions, the way in which modularization is supported, notational conventions, and the definition of the semantic models as execution trees. This will serve as our example of how a semantics can be associated with a textual transition system, and the translation issues that prevent a system and its translation from being true for the exact same properties. In Section 3 such issues in the translation of different computation models are considered, showing that both the notions of modularity and the relations among the underlying semantic models may have to be complex.

The notion of a faithful relation among models and specifications is defined formally in Section 4. We then demonstrate how the inherent model incompatibilities seen in the translation issues lead to natural examples of nontrivial faithful relations among models and their translations. Some of these correspond to specific examples from the literature, others to undocumented 'folk knowledge', and others to new connections among classes of properties.

2 CDL Modules and Transitions

The core design language CDL needs to express a variety of notations, be simple in its own right, but not suffer from excessive blow-up in the size of the system description. Thus for a translation from some source notation to CDL, if the source notation is modular, and combines modules either synchronously, asynchronously, or with partial synchronization, the core should express this in a modular system description of a similar size. The core description should facilitate textual analysis and information gathering, transformations to alternative forms, and translations to and from other notations. Thus emphasis is put on a variety of synchronization methods and possibilities for instantiating declarations of modules with different parameters. On the other hand, internal control structures common in programming languages (conditionals, looping statements, etc.) are not included in CDL, and will be encoded using program counter variables.

The core design language of VeriTech is based on collections of textual transitions, organized in modules. The modular transition system for the core incorporates ideas from state transition systems (especially the internal representation in the STeP system [1]), Z schemas [25, 23], and LOTOS [2, 3] composition operators. It is intended to deal with the issues outlined above, and to facilitate translations. Note that it is not particularly intended for direct human interface.

2.1 The CDL notation

A system in CDL is composed of global declarations of types, constants, and variables, and declarations of the components of the system, which are called *modules*. A simple two place buffer example is given in Figure 1, and is explained below. The syntax and semantics of the globally declared types, constants, and variables are entirely standard. Module declarations are at the top level of a system, and are not nested. One module has the special designation SYSTEM (the COMB module in the example) to indicate that it defines the entire system. Each module declaration has a name, formal parameters, variable declarations, and a body that defines a collection of textual transitions, as described below. Local variables and their types can be declared for each module. Every variable name appearing within the body of a module declaration should be declared globally, declared as a local variable of the module, or be a formal parameter.

The basic element of the body of a module is a *transition* defined as a triple $\tau = \langle I, P, R \rangle$, where I is an identifier (called the *name* of the transition), P is a predicate over states called the *precondition* or *enabling condition*, and R is a relation between states called the *transition relation*. The relation R is written as a logical formula including unprimed and primed versions of state variables. It is also optionally possible to write the relation as an assignment to the primed versions in terms of the unprimed ones, when appropriate. As will be explained in Section 2.3, the intuitive interpretation of a transition is that if the system is in a state that satisfies the transition's precondition, then it can be activated, which means that the state before and the state after the activation satisfy the transition relation, where the unprimed versions of variables relate to the state before the activation, and the primed versions represent the state afterwards.

The relation R should be total for the states of the system satisfying P. That is, if state s satisfies P then there exists a state s' such that the pair (s, s') satisfies R. This is guaranteed by automatically adding to the precondition of each transition the requirement that there exist values so that the relation can be satisfied. Otherwise, the transition is not enabled in the state.

A module body can be most simply specified by listing such transitions within a pair of brackets (as in the three first modules of the example).

One module can be defined in terms of others (as in the COMB module), by using instantiations of modules, but the definitions cannot be recursive. An instantiation of a module is created by listing the name of the module with actual parameters (variable names) in place of the formal ones, in the body of another module. In this case, it is as if a version of the module with the actuals substituted for the formals has been created. If this creates any conflicts between the actual parameters and local variable names, a systematic renaming is made of the local variables. Note that the effect of a variable common to two module instantiations, but not global to the system, can be attained by using the same actual parameter for two module instantiations in the same body (as is done with s in the instantiations of SENDER and BUFFER inside the body of COMB).

There are three composition operators used in combining instantiations of modules.

$P|||Q$ is called *asynchronous composition*, and is defined semantically as the union of the transitions in P and in Q, where P and Q are instantiations of modules with actual parameters. As noted above, a variable common to the instantiations is defined by using the same actual parameter in both instantiations.

$P||Q$ is called *synchronous composition* and is defined as the cross product of the transitions in P and in Q. The cross product of two transitions has a precondition of the conjunction of their preconditions, and a relation that is the intersection of the two relations (the conjunction of the relations written as logical formulas). From the definition of a transition, it follows that elements in the cross product for which the precondition is *false*, or for which the relation cannot be satisfied when the precondition holds, cannot be activated as transitions, and thus can be removed.

$P|s\_set|Q$ is *partial synchronization*, where the synchronization set s_set is a set of pairs of names of transitions, with the first from P and the second from Q. The module is defined as the cross products of the pairs of transitions in the list, plus the union of the other transitions from P and Q that do not appear in the list.

The COMB module has partial synchronization between the *send* and *get* transitions of the SENDER and BUFFER modules, respectively. This means that there is a joint transition that is enabled only when both of the enabling conditions of those transitions are true, and the result is the intersection of the results of those transitions. Another system could be defined by SENDER(s) ||| BUFFER(s,t) ||| RECEIVER(t). In this case each transition remains independent, and the SENDER can repeatedly 'produce' and 'send' s values, while the BUFFER occasionally decides to 'get' and then later actually moves the most recently sent value (losing the previous ones). (Similar effects would occur between the BUFFER and the RECEIVER). Thus the component modules can be combined in a variety of ways, giving the advantage of process algebra along with the simplicity of a collection of transitions.

```
HOLD_PREVIOUS
MODULE SENDER (a: INT) {
   VAR readys: BOOL INIT false
   TRANS produce:
         enable: ¬ readys
         relation: (a' = 0 ∨ a' = 1) ∧ readys' = true
   TRANS send:
         enable: readys
         relation: readys' = false
}
MODULE BUFFER(c,d: INT) {
   VAR cok: BOOL INIT true, dok: BOOL INIT false
   TRANS get:
         enable: cok
         relation: cok' = false
   TRANS move:
         enable: ¬cok ∧ ¬dok
         relation: d' = c ∧ cok' = true ∧ dok' = true
   TRANS put:
         enable: dok
         relation: dok' = false
}
MODULE RECEIVER(b: INT) {
   VAR vr: INT
         readyr: BOOL INIT true
   TRANS consume:
         enable: ¬ readyr
         relation: vr' = b ∧ readyr' = true
   TRANS receive:
         enable: readyr
         relation: readyr' = false
}
MODULE COMB () {
   SYSTEM
   VAR s,t: INT
   (SENDER(s) |(send,get)| (BUFFER(s,t) |(put,receive)| RECEIVER(t)))
}
```

Figure 1: A buffer system in CDL

2.2 Flags, additional keywords, and conventions

It is possible to extend every state of the system automatically with (boolean) *flags*. Here we consider only two possibilities relevant to our discussion. The *visible* flag can both appear in the precondition and be changed by the relation. Only states for which *visible* is *true* will be considered as having to satisfy specification formulas. Other states are considered to be *hidden*. This will allow defining different grains of atomicity, and use what has been called *mini-steps* [11] in defining more complex transitions.

The global constant HOLD_PREVIOUS flag is used to define the next-state value of a variable when it is not assigned by a transition. If HOLD_PREVIOUS is false, then such a variable is assumed by default to have arbitrary values. Thus if part of the state is to be unchanged, that should be listed explicitly, as in $x' = x$. This assumption is appropriate for modules that are composed synchronously, because it means that for an unassigned variable in a module, all possible values are possible options for the outcome of a synchronous transition involving the module as a component. Thus if another module does assign a value to the variable, that value will be the one used in the synchronous transition. A default of maintaining the previous value would lead to a cross-product that is almost always impossible.

On the other hand, maintaining the previous value is the natural default for asynchronous compositions of modules. Thus if HOLD_PREVIOUS is true, unassigned variables are understood to maintain the previous value in all transitions of the system, as in the example.

There are also some programming conventions for the core that are useful in translating systems and optimizing representations. The simplest is that identifiers (variable names) beginning in '&' are considered *nonessential.* This means that any translation or equivalent core representation that eliminates such variables or updates them differently is acceptable, as long as the other parts of the state are not affected in any way. The intention is that such variables will be generated during the translation process of VeriTech, and are not in the original system. Thus they will not appear in any assertion about the system, and can be ignored for analysis purposes, except as they affect the other variables.

Similarly, not only a variable, but an entire state can be nonessential. For example, states which are hidden (i.e., for which the flag *visible* is *false*) are nonessential–but the overall change of a series of transitions among hidden states beginning and ending in a visible state must be the same as if there were a single transition directly between the visible states.

Another convention is intended to aid in the treatment of control statements in various notations. CDL itself does not have explicit control constructs. Variables called *control counters* enable ordering the enabling conditions of transitions to implement sequential control, conditional, or loop statements from other notations. Such variables are assumed to begin with the characters '&PC'. This convention helps in the analysis and translation of CDL programs with such variables.

2.3 The semantics of systems and modules

In order to compare translations in different models, a uniform semantic basis is required. We will assume that for each notation for describing models an execution tree semantics can be derived.

As an example of such a derivation, we consider the semantics of CDL. The semantics of a system and of a module can be defined in two stages. First, for semantic purposes only, each definition of a module should be textually expanded to give the module as a list of transitions, replacing instantiations of modules by the collections of transitions they define (including substitution of actual parameters in place of formal ones, and renaming local variables when necessary to avoid conflicts). Because the definitions are not recursive, this is always possible.

Now we can define the semantics of a module with transitions given explicitly, by considering the execution sequences (also called *traces*) that it defines. Of course, this is only for purposes of ensuring the correctness of translations: the core is not intended to ever 'execute' transition systems described in CDL. A state of a system is identical to the state of the specially designated SYSTEM module. It clearly contains the constants and variables declared globally, and also those that follow from the instantiations of other modules and their local variables. Recall that a transition $\tau = \langle I, P, R \rangle$ can be activated in a state s if s satisfies P. The activation of τ can be seen as constructing a new system state s' from the existing state s of the system, where the pair (s, s') satisfies R. For a system or module defined by a collection of transitions, the possible execution sequences are defined by the sequences of states reached by successive activations of transitions, starting from an initial transition.

The execution sequences are organized into an execution tree, where each state is the parent of the states reachable from it by activation of an enabled transition. The root of the tree is a state with all variable values undefined (e.g., equal to a special value $\perp$), except those with initial values from 'INIT c' phrases after their declaration. This tree is the semantic interpretation of a system or module.

Other notations can also be given a execution tree semantics, allowing comparisons among translations. The correctness of a translation (whether in VeriTech or not) is defined relative to such trees.

However, it is important to note that the properties that are to be shown about a system can influence how much of the information in this tree is relevant. According to various possible conventions (some of which were mentioned in the previous section), the tree of a system is equivalent to reduced versions that, for example, eliminate nonessential variables, or remove hidden transitions, without otherwise affecting the system. Moreover, if only linear-time temporal properties will be proven, then the set of traces can be considered, and their organization into a tree structure is irrelevant. Furthermore, if only invariants are of interest, then it is sufficient to consider the set of reachable states. Such considerations will be crucial in understanding the relations needed among models, as will be seen in the continuation.

As part of the specification, additional restrictions can be added to define which traces are relevant. For example, fairness constraints can be added that influence which enabled transitions are chosen, and thus exclude some of the 'unfair' traces from consideration. There are also contexts in which an assumption of finiteness of the traces is appropriate, excluding the infinite ones.

For specific notations, particularly those defining finite-state systems, it will be convenient to consider also a finite representation of the execution tree by means of a finite state machine.

3 Issues in Translation

Translating between different modeling paradigms requires finding suitable solutions for those modeling aspects that are available in one model but not in the other. Translations generally attempt to keep the translated representation of the model as similar as possible in structure and size to the original system, and in addition to define the relation among the underlying semantic models so that wide categories of properties will be related in the two models.

It is important to note that even when there is a blowup in the model representation (the 'program text'), this does not necessarily imply a blow-up in the size of the model (given as an execution tree or a state machine). Below we consider some of the key issues in translation that make it impossible to always maintain the same semantic tree or state machine for a model and the result of its translation.

3.1 Synchrony and asynchrony

Notations for describing models commonly use three types of composition operators between system modules: synchronous, asynchronous and partially synchronous (which, for example, can be seen in generally asynchronous composition with handshaking communications). For this reason CDL has all three types. Translating among models with the same type of synchrony avoids the specific class of problems of this subsection (although there may be other types of difficulties).

However, we have to resolve cases in which the source model originates from a system with one type of composition while the resulting target model is in a notation that uses a different one.

Assume that we want to translate a synchronous core system into an asynchronous tool. In a tool like Murphi, where no synchronization mechanism is available, the translation is done by constructing a Murphi rule for each pair of transitions that should be synchronized. Thus, the original partition into modules is lost. In SPIN, on the other hand, the original partition into modules is preserved and synchronous execution of two transitions is simulated using handshaking communication (via a zero-length buffer).

Translating from an asynchronous model into a synchronous model (like SMV) should guarantee that, at each step, at most one module executes a transition while all the others are idle. This can be done by adding a self-loop on each state and a mechanism (a shared variable like `running` in SMV or an additional process) that enables the transitions of one module at a time. In this case the modules correspond to processes. Various fairness constraints can be added in order to eliminate traces in which all processes are idling forever, one process idles forever (starvation), or all processes idle at the same step (so the global state repeats).

3.2 Unenabled transitions

In a typical transition system representation, each transition consists of an enabling condition, an optional assignment, and a relation that should hold among values of variables before and after the execution of the transition.

The semantics guarantees that the transition is executed only if its enabling condition holds and if its final values satisfy the relation. A precise translation should identify the values for which the enabling condition and the relation hold and construct a transition for these values only. This, however, may not be possible in the target notation.

One possible solution to this problem is to introduce a special **fail** state in the target program. Transitions in the target program are extended with a conditional statement that results in the original final values if these values satisfy the needed relation, and otherwise results in the **fail** state. Assuming this is the only change caused by the translation, the resulting semantic model has transitions to the **fail** state added to the execution tree, and that state is a leaf (or sink, if we view the addition as adding just one such state).

3.3 Atomicity of transitions

In many notations, transitions are considered atomic. This means that each transition is performed in isolation, with no interference.

In Murphi each transition (called a *rule*) is also considered atomic. However, there a transition can be defined by any C program. When such a complex transition is translated into a notation with a finer grain of atomicity (e.g., where each transition can be a single assignment to the state), it must be partitioned into a sequence of steps. A *visible* flag (or its equivalent) is typically used to indicate that the intermediate states do not occur in the original model, and are an unavoidable result of the difference in the possible grain of atomicity.

In other tools like SPIN and LOTOS atomic actions are more restricted. SPIN, however, includes a mechanism to define a sequence of statements as atomic. Thus, it is straightforward to maintain the atomicity of, e.g., Murphi transitions within SPIN. LOTOS, on the other hand, does not have such a mechanism. As a result, a translation from any notation with large-grained transitions to LOTOS requires providing a mutual exclusion mechanism that enables the translation of a transition to run from start to end with no intermediate execution of actions from other transitions.

3.4 Variables with unspecified next values

Models of computation differ also by their convention concerning variables whose next-state value has not been specified by the executed transition. One convention, usually taken by asynchronous models, assumes that such variables keep their previous values. Another convention, common to synchronous models, assumes that the variables can nondeterministically assume any value from their domain. If the first convention has been taken and we translate the program into a model where the second holds, then for every transition the resulting program will have to contain an explicit assignment of the previous value for every variable not already explicitly redefined. For the other direction (from a model with any value as a default to one that keeps the previous value), we could use nondeterministic assignment, if the target model contains it. Otherwise, the resulting program could contain a choice among all possible explicit assignments, for each of the possible values in the domain. Here the blow-up in the size of the resulting program is unavoidable, but at least the semantics does not change.

3.5 Partitioning into Components

Partitioning into components (modules, processes, etc.) differs conceptually among languages because they are driven by diverse concerns. In many notations oriented towards programming languages, a component is task-oriented, and a task can change values of several variables. In hardware description languages like SMV, however, it is more common to collect all possible changes to a single variable into one component. A component then describes, for example, all possible changes to a given register. Such differences sometimes make it difficult to maintain the modular structure of the original system, and may force introducing variables or operations that are 'global' under the partitioning advocated by the target notation.

3.6 State extensions

The addition of a *visible* flag, or the need to globally declare variables that originally were local in a notation with local modules, or the addition of an explicit mutual exclusion mechanism to simulate differences in the grain of atomicity all mean that the state of the translated program must often be extended. Another common problem is that the target notation may not have the sequencing options of the source. Then the control flow of the original computation is sometimes maintained by adding a program counter as an explicit part of the state, and using it in the enabling condition of the transitions.

Such extensions to the state add variables that are needed to express the model, but usually are not part of the original assertions in the specification of the source. As was seen in the description of CDL, such variables are called *nonessential* for the purposes of assertions about the model, even though they are needed to express the model itself. Of course, translations can also eliminate such variables, as when explicit control variables are replaced by sequencing of translated steps, in a notation that has control commands.

4 Faithful Translations

As seen in the previous section, the semantics of the translated model cannot always be identical to that of the original, and thus not all properties can be preserved.

Alternatively, we propose the following goal. Assume we are given two models, M_1 and M_2, possibly defined within two different verification tools. Further assume that the models are related via some model-translation relation. We identify a set of properties of M_1 and a property-translation relation such that for every property in the set, if M_1 satisfies the property then M_2 satisfies the translated property. The translation is then called *(weakly) faithful* with respect to those models and families of properties.

We may sometimes further require a *strongly faithful* translation such that in addition to the above, if M_1 does not satisfy the property then M_2 does not satisfy the related property.

We require faithfulness to be transitive so that a series of translations can be considered. In particular, for general translation through a core notation, as in VeriTech, it is sufficient that the translations of models and of families of properties are faithful between different tool notations and the core (in both directions, perhaps for different families of properties). The faithfulness of the translation from one tool to another will then result from transitivity arguments.

Formally, let $\mathcal{M}_1$, $\mathcal{M}_2$ be two classes of models and $\mathcal{L}_1$, $\mathcal{L}_2$ be sets of properties expressed as formulas in an assertion language for $\mathcal{M}_1$ and $\mathcal{M}_2$, respectively. Let $TR \subseteq \mathcal{M}_1 \times \mathcal{M}_2$ be a *model-translation* relation and $tr \subseteq \mathcal{L}_1 \times \mathcal{L}_2$ be a *property-translation* relation that is total over $\mathcal{L}_1$ (i.e., so that each formula of $\mathcal{L}_1$ is in the relation tr).

TR and tr are *faithful* for $\mathcal{M}_1$, $\mathcal{M}_2$, $\mathcal{L}_1$, and $\mathcal{L}_2$ if $\forall M_i \in \mathcal{M}_i$ and $f_i \in \mathcal{L}_i, i = 1, 2$, whenever $TR(M_1, M_2)$ and $tr(f_1, f_2)$, then $M_1 \models f_1 \Longrightarrow M_2 \models f_2$.

TR and tr are *strongly faithful* for $\mathcal{M}_1$, $\mathcal{M}_2$, $\mathcal{L}_1$, and $\mathcal{L}_2$ if $\forall M_i \in \mathcal{M}_i$ and $f_i \in \mathcal{L}_i, i = 1, 2$, whenever $TR(M_1, M_2)$ and $tr(f_1, f_2)$, then $M_1 \models f_1 \Longleftrightarrow M_2 \models f_2$.

Because we use a relation (rather than a function) among the models in our definition of faithfulness, we can refer to derivations of properties in either direction between the source of a compiler-like translation of the model and the target. Thus, we can treat both importing properties from the source to the target, and showing that properties true of the target have versions true of the source. The former case is treated by viewing the source of the compiler translation as M_1 and the target as M_2, while the latter simply switches their roles, so M_1 is the target and M_2 is the source of the model translation (and thus $TR(M_1, M_2)$ holds if the compiler translates from M_2 to M_1).

In the continuation we express the families of properties as sublanguages of various temporal logics, although other modes of expression are possible. Note that the sets of languages for which we define faithfulness are not necessarily subsets of the specification languages used by the tools. For example, a compiler translation from Spin into SMV could be faithful from SMV to Spin (so $TR(SMV, Spin)$) for a transformation of properties expressible in linear-time temporal logic. That is, if a linear-time temporal logic property is shown of the SMV model, then it will hold for the Spin version. This holds even though the specification language of SMV is the (restricted) branching time logic CTL, that cannot express everything expressible in linear-time temporal logic. In such a situation, model checking (in SMV) of a property in the intersection of CTL and linear-time temporal logic will be meaningful for the original Spin model and the appropriate transformation of the property.

On the other hand, if we show that the translation from Spin to SMV is faithful (so we consider $TR(Spin, SMV)$ for a transformation of all safety properties of Spin, then we can assume that the SMV model satisfies the transformed versions of all safety properties already shown about the original model in Spin.

Below we present some examples of model-translation relations TR and property-translation relations tr that are faithful for given models and families of specifications. In justifying the faithfulness, semantic proofs on the relations among the execution trees (or finite state machine representations, where applicable) are used (only hints on the proofs are given here).

4.1 Operation refinements

Assume $TR(M_1, M_2)$ if the semantics of M_1 and M_2 are execution trees such that only M_2 uses the *visible* flag, and the *visible* states of M_2 are identical (except for the *visible* flag) to (all of) the states of M_1. Furthermore, the only change in the semantic trees is that each transition of M_1 (edge in the tree) is replaced by a finite sequence of transitions between the corresponding *visible* states of M_2, so that all intermediate states along the sequence have $\neg visible$ and there are no infinite sequences of states with $\neg visible$. (In this case, common terminology will refer to M_2 as a *refinement* of M_1.) Such a translation can arise due to the differences in atomicity seen earlier.

Let $\mathcal{L}_1$ be all properties expressible in some temporal logic. Then tr should relate formulas of $\mathcal{L}_1$ to formulas of $\mathcal{L}_2$ in which the temporal operator next-time ($\mathbf{X}$) is replaced by the temporal operator until ($\mathbf{U}$), in a systematic way. Furthermore, those formulas refer only to *visible* states. If we consider linear-time temporal logic as the languages of both the source and the target of the translation, an atomic state assertion p in $\mathcal{L}_1$ is transformed to $visible \wedge p$. An assertion $\mathbf{X}p$ is transformed to $\mathbf{X}(\neg visible \mathbf{U}(p \wedge visible))$ and $p\mathbf{U}q$ becomes $(visible \rightarrow p)\mathbf{U}(visible \wedge q)$. (From the transformations above, it follows that $\mathbf{F}p$ becomes $\mathbf{F}(visible \wedge p)$ and $\mathbf{G}p$ becomes $\mathbf{G}(visible \rightarrow p)$.)

The translation is then strongly faithful for both $\mathcal{L}_1$ and $\mathcal{L}_2$ being linear-time temporal logic. Note that the formula translation tr is defined for every formula of $\mathcal{L}_1$, as required. Moreover, all formulas obtained as the result of applying the above transformations on a linear-time temporal logic formula are true of M_2 iff the original formula is true of M_1. Intuitively, this is true because the only difference between the models is that each next state s has been replaced by a finite sequence of states in which $\neg visible$ holds, ending with a state corresponding to s in which $visible$ holds. A formal proof uses induction on the structure of the formulas in $\mathcal{L}_1$, and is based on the differences among the models, and the transformation applied. Finally, note that no assertion is made in the definition of faithfulness about formulas of $\mathcal{L}_2$ that are not the result of the transformation tr.

For this translation, it is also possible to use CTL as the language of interest, and obtain a strongly faithful translation, by using the transformation $\mathbf{AXA}(\neg visible \mathbf{U} visible \wedge p)$ to replace the assertion $\mathbf{AX}p$ (and similarly for $\mathbf{EX}p$). This is valid because each edge of the original structure is replaced by a linear subsequence, with no internal branching.

If the translation among the models *does* allow infinite sequences of states with $\neg visible$ or paths with a $\neg visible$ state as a leaf, and otherwise satisfies the relation described above, this would correspond to possible loops in the Kripke structure representing $\mathcal{M}_2$. In this case, safety properties of M_1 can be transformed as above and yield a faithful translation, but liveness properties are not maintained: $\mathbf{F}p$ is transformed to $\mathbf{F}(visible \wedge p)$, which does not hold for our less strict translation.

4.2 Adding idle transitions

A similar situation arises if the difference between M_1 and M_2 is only that self-loops (idle transitions) have been added to the Kripke structure of M_2. This occurs, for instance, in translations from asynchronous models to synchronous ones. In the execution tree of M_2, finite repetitions of the idle transitions appear between original M_1 transitions. In addition, the execution tree contains infinite sequences of idle transitions. However, every original behavior of M_1 is also a behavior of M_2. Thus, the execution tree of M_2 is greater by the *simulation* preorder [19] than that of M_1.

Consequently, tr taken as the identity relation will be faithful if both languages are ECTL* [9], i.e., the subset of CTL* that contains only existential path quantifiers. In the other direction, viewing M_1 as the model with idle transitions, tr as the identity relation will be faithful for both languages being CTL* specifications that contain only universal path quantifiers (ACTL*) [10].

If the infinite idle executions are eliminated due to fairness constraints in the models, then the models are *stuttering bisimilar* and tr as the identity will be strongly faithful with respect to full CTL$^*_{-X}$ [4]. The well-known observation of Lamport [16] that linear-time temporal assertions without the next-time operator are insensitive to such 'stuttering' is a special case. Moreover, for this translation of models, we can also define a faithful property translation tr for all of CTL*, by translating formulas that include $\mathbf{X}$ to weaker versions without $\mathbf{X}$, e.g., replacing $p \wedge \mathbf{X}q$ by $p\,\mathbf{U}q$. If M_1 satisfies $p \wedge \mathbf{X}q$, it also satisfies $p\,\mathbf{U}q$. Since the latter formula does not include $\mathbf{X}$, it will be true of M_2 by the previous arguments.

4.3 Adding explicit failures

Assume $TR(M_1, M_2)$ if the execution tree of M_2 is identical to the execution tree of M_1, provided that all *fail* states are truncated along with the transitions into them (see for example Section 3.2).

Assume again that $\mathcal{L}_1$ is a temporal logic. Then tr will relate formulas in $\mathcal{L}_1$ to similar formulas that refer only to paths along which no *fail* occurs ("globally not *fail*"). In linear-time temporal logic, the transformation for any formula f is to $(G\neg fail) \rightarrow f$. A similar transformation is not applicable to CTL. However, it can be expressed using *fair* CTL (CTL^F [7]). In CTL^F, fairness constraints are given by means of a set of CTL formulas.

An execution is fair if it satisfies infinitely often each of the constraints, and only fair executions are considered in the semantics of CTL^F formulas. Thus, tr can replace a CTL property f of M_1 by the CTL^F property f together with the fairness constraint $\neg fail$ for M_2. (This transformation is valid because once *fail* becomes *true*, it remains *true* forever. Thus, "infinitely often not *fail*" implies "globally not *fail*".)

4.4 Data abstraction

Assume $TR(M_1, M_2)$ if M_1 is an abstraction of M_2, where the variables in M_1 are defined over (smaller) abstract domains and the variables in M_2 are defined over concrete domains. Some of the variables of M_2 may not appear at all in the translation M_1. (Note that here we have reversed the roles of M_1 and M_2 as explained earlier, so M_2 is the source and M_1 the result of the translation.) Such a translation may occur in going from a model that needs extra control variables to a target model where nonessential variables might not be needed. In going from theorem prover tools like PVS or STeP to model checkers, such abstractions of variables are also likely.

The treatment of abstraction of nonessential variables is easy: the very fact of their being declared as nonessential is equivalent to an assumption that any specification formula of interest does not make assertions about them. Thus in showing faithfulness, we can assume that the assertions that define $\mathcal{L}_1$ and $\mathcal{L}_2$ do not include those variables.

For general data abstraction, the abstract model is often obtained by defining an abstraction mapping and grouping concrete states that map into the same abstract state. A transition is in the abstract version if there was a transition between a state that maps to the source and one that maps to the target. A faithful transformation tr can be defined in this case for the language ACTL*, defined over atomic propositions that relate variables to values from the abstract domain (e.g. $v = a$). tr will replace each atomic proposition of the form "$v = a$" with the atomic proposition "$v \in \gamma(a)$", where $\gamma(a)$ is the set of all concrete values mapped to the abstract value a [6, 8].

4.5 Changes in the branching structure

Assume $TR(M_1, M_2)$ if M_1 and M_2 have the same set of execution paths, but not necessarily the same execution trees. Thus the translation does not preserve the branching structure of the models. If $\mathcal{L}_1$ is all properties expressible in a linear-time temporal logic then tr is the identity relation. If $\mathcal{L}_1$ is a branching-time logic then for the formulas of $\mathcal{L}_1$ which are also linear-time formulas tr is also the identity. Other formulas will have nontrivial tr relations, depending on the changes in the branching structure and the formulas. For example, an assertion $AFAGp$ (eventually on every path there is a subtree with p in every node) can be transformed to the weaker (but guaranteed true) $AFGp$ (on every path there is a suffix with p true). If additional information is available on the nature of the changes in the branching structure, the translations can be faithful for richer families of properties (languages).

5 Conclusions

The VeriTech framework and core language are presented, as an example of a systematic translation among models and verification tools. Basic issues in translation are identified that force the models and structure of translations to differ from the original, both for VeriTech, and for any of the numerous other translations among notations for model descriptions. The framework of a faithful translation between both models and properties is introduced. This framework is shown to express necessary relations among models and properties of those models.

Faithfulness captures and extends several widespread instances of relations among models previously treated in an ad hoc way. In practice, many translations involve more than one of the types of differences among models that were presented. Thus combinations of the transformations of properties are needed to guarantee faithful relations for interesting classes of properties. The quality of translations can be assessed by whether the faithful relations established are sufficient to allow using the models both to import already proven properties into versions automatically true of a translation, and to establish that properties proven of a translation correspond to properties that will then be automatically true of the original model.

The ability to easily move among models, properties of interest, and tools extends the practical applicability of formal methods, and reduces the dependence on a single tool.

References

[1] N. Bjorner, A. Browne, E. Chang, M. Colon, A. Kapur, Z. Manna, H.B. Simpa, and T.E. Uribe. Step: The stanford temporal prover - user's manual. Technical Report STAN-CS-TR-95-1562, Department of Computer Science, Stanford University, November 1995.

[2] T. Bolognesi and E. Brinksma. Introduction to the ISO specification language LOTOS. *Computer Networks and ISDN Systems*, 14:25–59, 1987.

[3] T. Bolognesi, J.v.d. Legemaat, and C.A. Vissars (eds.). *LOTOSphere: software development with LOTOS*. Kluwer Academic Publishers, 1994.

[4] M. C. Browne, E. M. Clarke, and O. Grumberg. Characterizing finite kripke structures in propositional temporal logic. *Theoretical Computer Science*, 59(1–2), July 1988.

[5] J.R. Burch, E.M. Clarke, K.L. McMillan, D. Dill, and L.J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98:142–170, 1992.

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

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

[8] D. R. Dams, R. Gerth, and O. Grumberg. Abstract interpretation of reactive systems. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 19(2), March 1997.

[9] D.R. Dams, O. Grumberg, and R. Gerth. Abstract interpretation of reactive systems: Abstractions preserving ACTL*, ECTL* and CTL*. In *IFIP working conference on Programming Concepts, Methods and Calculi (PROCOMET'94)*, San Miniato, Italy, June 1994.

[10] O. Grumberg and D.E. Long. Model checking and modular verification. *ACM Trans. on Programming Languages and Systems*, 16(3):843–871, 1994.

[11] D. Harel. Statecharts: a visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.

[12] G. Holzmann. *Design and Validation of Computer Protocols.* Prentice-Hall International, 1991.

[13] G.J. Holzmann and D. Peled. The state of SPIN. In *Proceedings of CAV96*, volume 1102 of *LNCS*, pages 385–389. Springer-Verlag, 1996.

[14] C.N. Ip and D.L. Dill. Better verification through symmetry. *Formal Methods in System Design*, 9:41–75, 1996.

[15] R.P. Kurshan. *Computer-aided Verification of Coordinating Processes.* Princeton University Press, 1994.

[16] L. Lamport. What good is temporal logic. In *9th World Congress*, pages 657–668. IFIP, 1983.

[17] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification.* Springer-Verlag, 1992.

[18] K. L. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem.* Kluwer Academic Publishers, 1993.

[19] R. Milner. A formal notion of simulation between programs. Technical Report 14, Swansean University, 1970.

[20] http://wwwbrauer.informatik.tu-muenchen.de/gruppen/theorie/KIT/.

[21] S. Owre, S. Rajan, J.M. Rushby, N. Shankar, and M.K. Srivas. PVS: Combining specification, proof checking, and model checking. In Rajeev Alur and Thomas A. Henzinger, editors, *Computer-Aided Verification, CAV '96*, volume 1102 of *Lecture Notes in Computer Science*, pages 411–414, New Brunswick, NJ, July/August 1996. Springer-Verlag.

[22] Sam Owre, John Rushby, Natarajan Shankar, and Friedrich von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, February 1995.

[23] B. Potter, J. Sinclair, and D. Till. *An introduction to Formal Specification and Z.* Prentice Hall, 1991.

[24] W. Reisig. *Elements of Distributed Algorithms– Modeling and Analysis with Petri Nets.* Springer-Verlag, 1998.

[25] J.M. Spivey. *The Z Notation: a Reference Manual, 2nd. ed.* Prentice Hall, 1992.