A Framework for Translating Models and Specifications *

Shmuel Katz and Orna Grumberg

Computer Science Department
The Technion
Haifa, Israel
{katz,orna}@cs.technion.ac.il

Abstract. The reasons for translating a description of a model in one notation into another are reviewed. This includes both translating entire models and describing different aspects of a system using different notations.
In order to demonstrate the ideas, the VeriTech framework for translation is described. A system being analyzed is seen as a collection of versions, along with a description of how the versions are related. The versions are given in different notations connected through a core notation by compilers from and to the notations of existing tools and specification methods. The reasons that translations cannot always be exact are analyzed, based on experience with over ten separate compiler translations among formal methods notations. Additional information gathered during translation is described, to facilitate optimizations, error tracing, and analysis.
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 syntactic 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. The classes of properties that can be faithful for a given translation provide a measure of the usefulness of the translation. Open research directions are suggested concerning faithful transformations, additional information, error tracing, and optimizing translations.

1 Introduction

In this survey paper we present the possible uses of (direct or indirect) translations among model descriptions, show some of the difficulties that must inevitably arise during translation, describe the design of the VeriTech translation framework and how it can alleviate some of the obstacles to translation, and provide a theoretical basis to quantify the quality of such translations in a formal framework using *faithful* translations and syntactic transformations of properties. Material is included from [22, 11, 24] and [3].

* This research was partially supported by the Fund for the Support of Research at the Technion and by the Bar-Nir Bergreen Software Technology Center of Excellence.

1.1 Existing translations

Translations among notations for representing models and hardware designs have become common, although often there is no available documentation. Such translations exist from SMV [8, 27], to PVS[29], from Murphi[20] to PVS, from SMV to Spin[18, 19], from several notations into Cospan[25], from automata-based notation into Petri nets, and among many other tools. Moreover, individual verification tools often have multiple input formats for models, and internal source-to-source translations. For example, the STeP system [4] and the exposition in [26] allow presenting designs either in a simple C-like programming language, or using a modular collection of textual transitions, and internally translates from the former representation to the latter. In addition to translations among formal methods tools, there is increasing interest in translating standard hardware design notations such as Verilog or VHDL (or internal industrial notations) to and from the notations of existing model-checking tools.

Translations are used also in the context of *software verification*. The Bandera tool set [15] translates Java source code to the model checking tools SMV, Spin, and dSpin [10] and also to the Java PathFinder software verification tool (JPF) [16, 7] which has its own translation to Spin.

Recently, the VeriTech project has been developed as a general framework for translation through a simple intermediate notation [11]. The VeriTech project defines a core design language (CDL) in which modules can be combined in synchronous, asynchronous or partially synchronous manners, and each module is a set of first-order transitions. The VeriTech project provides translations between existing notations and the core language, in both directions. At present, VeriTech includes translations between SMV, Murphi, Spin, STeP, Petri-nets[31] and the core design language, and work is underway to incorporate, among others, PVS and LOTOS[6]. Other such frameworks include the SAL system [2], and the Model Checking Kit[28].

1.2 Why translate?

Translations among model notations can be used in a variety of ways, and these influence what needs to be true about a translation. 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 the model (perhaps in stages, through an intermediate notation) to a BDD-based model checker that can handle the problem. Alternatively, the source could be a model description in the SMV language, but for which attempts to verify a property have failed, and the target could be a description appropriate for a tool with a theorem-proving approach like PVS or STeP. Of course, proving the desired property in such a target requires using inductive methods and is not automatic, but at least is not sensitive to the size of the data domain and will not suffer from the state-explosion problem. We shall also see that in many relevant translations the property to be proven in target models will not be identical to the property asserted about the original source model. Nevertheless, a *back-implication* is desired: a property should necessarily hold in the source whenever the related property holds in the target.

In addition, unrelated 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 like Spin. The system model can then be translated to a branching-time model checker like SMV for properties of that type. It can also be translated to a language with real-time notation, such as STeP, or to a theorem proving environment like PVS to treat infinite domains and first-order temporal properties. In this case, we would like to *import* some variant of the properties proven about the source into the target, so that they can be assumed there and used to help prove the new desired property.

Translation is also useful when an infinite or large finite model needs to be reduced prior to applying model checking. For methods like abstraction [9] and convenient executions [21] the system can first be modeled in full and sent to a theorem prover in which the abstraction or the choice of convenient executions is shown 'correct'. That is, the reduced version is shown to preserve the properties of interest, so that if the reduced version satisfies them, so does the original. The reduced version of the model (i.e., the abstraction or the convenient executions) can then be translated to a model checking tool that will verify temporal properties. Here again we would like to have back implication that is an essential link in guaranteeing the correctness of the proved temporal properties for the full model.

As already noted, there are also many translations to and from design notations that do not have associated verification tools. For hardware these include Verilog and VHDL, and for software, Bandera, JPF, and Statecharts [13,14] (which provides a hierarchical graphical state-transformation software design notation). Translating from such a notation to one with associated model-checking or other verification tools allows checking properties of existing designs, while a translation in the other direction can introduce a verified high-level design into a development process.

1.3 Semantic issues

The quality of a translation depends on guaranteeing a close relation between the properties true of the source and those true of the target. This can be used to define the 'correctness' of a model translation. As seen above, the relation among properties can be used in either direction: 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) or 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. In the broader framework proposed here, a translation and transformation of properties will be faithful with respect to families of properties 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.

The quality of translations also is related to the modularity and readability of the target model. That is, a desirable property of a translation is that it maintain the modularity of the source, and not lead to an explosion in the number of lines of code, relative to the source. Identifying inherent differences, and minimizing their influence, is crucial to effective translation among notations for describing models.

Investigation of these relations can be seen as a step in the research direction proposed in [17], to unify theories of programming. Here those theories used to describe models for formal verification tools are emphasized, rather than full-fledged programming languages.

1.4 Organization of the paper

In Section 2 the design of the VeriTech project is described, as an example of a general translation framework, and one way to treat the many translation issues that arise. In Section 3 the semantic assumptions we use to compare source and target models are defined, based on fair execution trees as a common underlying semantics. In Section 4, we identify the translation issues that prevent a system and its translation from having identical semantics and thus satisfying the exact same properties. Translations also can lead to loss of the modular structure of the original in translation, and to a 'code explosion' problem, where the number of lines of code increases radically during translation. The added information that can alleviate these difficulties is described in Section 5.

The notion of a faithful relation among models and specifications is defined formally in Section 6, with three variants. We then demonstrate in Section 7 how such a faithful transformation looks for a common example of the inherent model incompatibilities seen in the translation issues. In the Conclusions, some research directions in this area are suggested.

2 The design of VeriTech and CDL

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, described below. 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.

The core description should facilitate textual analysis and information gathering, transformations to alternative forms, and translations to and from other notations. The core actually has multiple versions of a system, and additional information connecting the versions, but discussion of those parts will be postponed to Section 5, after the semantics and differences among notations are considered.

In CDL 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 [4]), Z schemas [30], and LOTOS [5, 6] 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.

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 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. The two transitions can be jointly executed when both of the enabling conditions of those transitions are true, and the result is the intersection of the results of those transitions. Otherwise those specific transitions cannot be taken. 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 some

```
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)))
}
```

Fig. 1. A buffer system in CDL

of the advantages of process algebra along with the simplicity of a collection of transitions.

3 The Semantics of Systems and Modules

In order to compare a model in one notation and its translation to a different notation, a uniform semantic basis is required. We will assume that for each notation for describing models a *fair execution tree semantics* can be derived.

Consider the case of a system model given in CDL as a collection of textual transitions, each with an applicability condition and a state transformation. As seen, such a collection defines a module, and such modules can be composed into new modules synchronously, asynchronously, or with partial synchronization (handshaking). The semantics of such a system and of a module can be defined in two stages. First, for semantic purposes only, each definition of a module can be shown equivalent to a textually expanded ("flattened") version, where the module is 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).

Now we can define the semantics of such a 'flat' module with transitions given explicitly, by considering the execution sequences (also called *traces*) that it defines. A state of such a system clearly contains the constants and variables declared globally, and also those that follow from the instantiations of modules and their local variables.

Turning to the textual transitions, recall that each is a triple $\langle I, P, R\rangle$ with an *identifier* I, a *precondition* P over states, and a *relation* R between pairs of states. The intended semantics is that a transition can be activated in a state s if s satisfies P, and such an activation 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 state.

The initial state has all variable values undefined (e.g., equal to a special value $\perp$), except those with initial values given in their declaration.

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. If all sequences have the same initial state, that is the root of the tree. Otherwise, a root node with all variables undefined is added, and the possible initializations are the only transitions enabled in that state. (An alternative view would see the semantics as a forest of trees, each with its own initialization, but the single-tree view has the advantage of treating the initializations like other transitions, which can be helpful in some translations. The single-tree view has the disadvantage that usual temporal logic assertions –including invariants– are not intended to hold in the root if all its values are undefined.) Some of the paths in this tree can be declared irrelevant due to an additional *fairness* restriction that can remove infinite paths (criteria for which restrictions are reasonable can be found in [1]). This tree, with fairness restrictions, is the semantic interpretation of a system or module.

Other notations can also be given an execution tree semantics, allowing comparisons among translations. The correctness of a translation is defined relative to such trees, and this semantics is sufficient for the specification languages considered here.

Note that a richer semantics is possible, e.g., one that includes what is known as *partial-order* information. For example, if it is possible to ask which execution sequences are equivalent to which other ones under independence of operations in distributed processes, then semantic information on independence of operations is needed [23, 21]. This possibility is not considered further here.

In any case, it is important to note that the properties that are to be shown about a system can influence how much of the information in an execution tree is relevant. According to various possible conventions, 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. We have already seen that fairness assumptions can be added on the semantic level. 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. In fact, an (equivalent) alternative semantic basis could be the fair transition system notation used by [26].

4 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.

Even when there is a blow-up 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.

4.1 Synchrony and asynchrony

Notations for describing models commonly use three types of composition operators between system modules: synchronous, asynchronous and partially synchronous (for example, in generally asynchronous composition of processes with handshaking communications). Translating among models with the same type of synchrony avoids the specific class of problems of this subsection.

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 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 to be synchronized. In SPIN, on the other hand, the original partition into modules can be preserved and synchronous execution of two transitions is simulated using handshaking communication (via a zero-length buffer, thus adding to the statespace).

Translating from an asynchronous model into a synchronous model (like SMV, in its most common mode of operation) 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 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).

4.2 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 of the typical transition system notation seen earlier guarantees that a 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 as an atomic operation 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).

4.3 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 generally more restricted. SPIN, however, includes a mechanism to define a sequence of statements as atomic. Thus, it is straightforward to maintain the atomicity of Murphi transitions within SPIN. On the other hand, LOTOS 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.

4.4 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. This is natural in software, where an assignment to one variable leaves the others unchanged. Another convention, common to synchronous models, assumes that the unassigned variables can nondeterministically assume any value from their domain. This is common in hardware descriptions, because then all options are left open for a variable not updated in one component to be changed in a parallel (synchronously executed) component, and still obtain a consistent result.

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 assignments, if they are available in the target model. 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, and auxiliary variables are often needed, but at least the semantics does not otherwise change.

4.5 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 the 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.

4.6 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. 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 the sequencing of translated steps, in a notation that does have expressive control commands.

5 Versions and additional information

In order to deal with the difficulties seen in the previous section, the core of VeriTech does not simply include the result of a translation from one of the component notations. Instead, it has information about multiple versions of the system being considered, as well as information gathered during the translation process, which is often not reflected in the translated code. Some of the information that connects the source and target codes of a translation are:

- **state correspondences and extensions.** The variables in the target are connected to the variables in the source to which they correspond. When the translation has extended the statespace by adding variables not in the original, this information is recorded.
- **hidden transitions.** When atomic steps in the source are translated to a collection of steps in the target, the intermediate states should be identified as internal, or *hidden*. This is because invariant properties corresponding to those of the source are not expected to hold in such intermediate states.

– **operation correspondences.** When modularity has to be destroyed or redefined, the components that are the source of a combined action in the translation should be identifiable, to facilitate error analysis and retranslation. Thus when separate actions of components that are composed synchronously in the source have to be made into a single step of the target, because the target language does not support such composition, the fact that this action came from two parts of the source should be recorded.

Some of the information above is recorded in the target code itself, by using naming conventions and special predefined flags. Other parts of the added information can be in a *log* file. Inclusion in the code is indicated when the assertions to be made about the target model depend on the presence of such conventions. This will be explained further when the faithful correspondence between properties of the source and of the target is discussed, in Section 6.

Note that the added information is useful both for translations into the core language CDL, and for translations from CDL to a specific notation. The information added in the translations between CDL and Petri nets can be found in [24].

For CDL, the simplest naming convention is that identifiers (variable names) beginning in '&' are considered *nonessential*, because they were not in the source program for which this CDL program is the target, but were rather generated during the translation. This means that any translation from CDL 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. Since those variables are generated during the translation process of VeriTech, and are not in the original system, they will not appear in any assertion about the source system, and can be ignored for analysis purposes, except as they affect the other variables.

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.

It is possible to extend every state of a CDL 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* [13] in defining more complex transitions.

The core handles the issue of unspecified next values by allowing both of the possible defaults discussed earlier. The HOLD_PREVIOUS flag remains globally constant in the model, and 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 $x' = x$. Recall that this assumption is appropriate for modules that are composed synchronously. 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.

Note that states which are hidden (i.e., for which the flag *visible* is *false*) are also 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 with the cumulative effect of the series but directly between the visible states.

Above we showed that the source, the target, and additional information gathered during the translation are needed, and thus should be recorded. There also can be multiple CDL versions of a system, for example, where one could be an abstraction of another. Such situations occur when an infinite state program, say including integers, is abstracted to one with only boolean variables, or when some other form of reduction has been performed.

Besides the additional information gathered during translation, there is additional semantic information that can only be obtained through a deeper analysis and understanding of the models. In particular, for each version, we also are interested in the properties known to hold for them, say in temporal logic, and in transformations among classes of properties that ensure faithfulness among translations, as will be seen in Section 6. Just like the other information, this can be useful in optimizing translations, in tracing error analyses, and in deciding which properties to check for different versions of the model. These semantic issues are treated below.

6 Faithful Translations

Translations would ideally fully preserve the semantics of the translated system, thus guaranteeing that the source and the target satisfy exactly the same properties. However, as already seen, the semantics of the translated model cannot always be identical to that of the original.

Therefore we loosen the connection between the properties true of the source and those true of the target. 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 assertions about M_1 and a property-translation relation that connects the assertions in the set of assertions about M_1 to assertions about M_2.

One relation among the translations is that for every assertion in the set, if M_1 satisfies the assertion then M_2 satisfies the translated version of that assertion. The translation is then called *import faithful* with respect to those models and families of properties. We may alternatively establish that if the translated assertion is true of M_2, then the original assertion must have been true about M_1. This translation is then called *back-implication faithful*.

Of course, we may instead require a *strongly faithful* translation that satisfies both of the conditions above.

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 indicating that a model $M_1 \in \mathcal{M}_1$ is translated to a model $M_2 \in \mathcal{M}_2$. Similarly, $tr \subseteq \mathcal{L}_1 \times \mathcal{L}_2$ is 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 *import 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 *back-implication 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 $M_2 \models f_2 \Longrightarrow M_1 \models f_1$.

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 $M_1 \models f_1 \Longleftrightarrow M_2 \models f_2$.

A relation (rather than a function) is defined among the models in the definitions of faithfulness because internal optimizations or 'don't care' situations can lead to nondeterministic aspects in the translation. Thus, a single source model may be translated to any one of several target programs, or different source models can be translated to the same target. Similar considerations hold for the assertion transformations. Note that it follows from the definitions that if tr is a function, it is total over $\mathcal{L}_1$.

In this paper, we consider families of properties expressed as sublanguages of various temporal logics, although other modes of expression are possible. In particular, various forms of automata with infinite acceptance conditions are reasonable alternatives. 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 (so we have $TR(Spin, SMV)$) could be back-implication faithful for a transformation tr of properties expressible in linear-time temporal logic. In words, if a linear-time temporal logic property that is the second component in a pair satisfying tr is shown of an SMV model that is the result of activating the compiler on a Spin source model, then the first component will necessarily hold for the Spin source. This holds even though the specification language of SMV is the (restricted) branching-time logic CTL, which cannot express everything expressible in linear-time temporal logic. In such a situation, model checking (in SMV) of a transformed property in the intersection of CTL and linear-time temporal logic will be meaningful for the original Spin model and the appropriate source of the checked property. Clearly, properties not in the range of tr are irrelevant for back-implication. Although they may hold of the target model, they give no information about the source model.

On the other hand, if we show that the translation from Spin to SMV is import faithful for a transformation of all linear temporal logic 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.

To establish that a (TR, tr) pair is faithful for two model notations and subsets of temporal logic properties, semantic abstractions must be established. Of course, the source and target models are given as code in different model description languages, and the translation works on the level of those codes. In the abstract level we need, the semantic models of the source notation and the target notation must be described, as must an abstraction of the model translation. The translation abstraction must show the changes introduced to the semantic model of the source in going to the target, as a transformation on the semantic trees. Two examples of such changes could be that a single transition in the source tree is replaced by a sequence of transitions in the target, or that some of the infinite paths of the source are replaced by finite paths that end in a specially designated *fail* state.

The transformation of temporal logic properties is given syntactically, where the family of properties is also defined by a syntactic structure. For this purpose the hierarchy of properties defined for normal forms of linear temporal logic in [26] can be used. For example, safety properties are characterized as being equivalent to a linear assertion $\mathbf{G}p$, where p only has past operators or is a property of a state with no modalities. Similarly, classes of properties seen in branching-time logics can be useful (e.g., 'forall' CTL* that uses only A and not E [12]). Then it must be shown that the transformed assertion is necessarily true of the target execution tree whenever the original is true of the source tree (for importation) or that the original assertion is necessarily true of the source tree whenever the transformed assertion is true of the target tree (for back-implication).

As seen, 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. Such variables can be directly used in expressing the transformation of assertions, as will be seen for the *visible* flag, in the following section. This is but one example of how the additional information can be used in defining the property transformation and the relevant families of properties.

7 Using Faithful Translations

Below we present an example of a model-translation relation TR and a property transformation tr that are faithful for given models and families of specifications.

Consider a translation where a single action in the source is divided into several target actions, due to different grains of atomicity. Translations in this family are called 'refinement translations'. Thus the target model will contain intermediate states between the states of the original model. Also we assume that the result program has the additional flag (state component) called *visible* which is turned on when the system is in a state from the original model[1], and turned off when it is in one of the intermediate states.

Definition- A path where all the states except the first and the last have a *false* value for their *visible* flag, will be called an intermediate path.

In a generic translation which does such refinement, the result model is characterized by having:
all of the state variables from the original model, plus an additional *visible* flag;
all the states of the original model, with a *true* value for the *visible* flag;
additional states, which have a *false* value for their *visible* flag.
The result model satisfies the following conditions:
1. For every two states which were connected by an edge in the original model, there exists at least one intermediate path between them in the result model.
2. For every two states which were not connected by an edge in the original model, there is no intermediate path between them in the result model.
3. There are no loops of only non-visible states (and thus there cannot be an infinite sequence of only non-visible states in the model paths).
4. In the paths of the result model the non-visible states always must appear as a finite sequence between visible states and not at the end of a path (This demand is a consequence of the previous one when the result model contains only infinite paths).
Note that we do not demand here that the non-visible intermediate paths for different pairs of states are distinct. Different intermediate paths can share the same non-visible states. Also, there can be several intermediate paths instead of one original edge.

The property transformation We define a property transformation for CTL* that is strongly faithful for all refinement translations. The transformation, tr, will be defined by an induction on the structure of the formula.

- for ϕ=p an atomic proposition: tr(p)=p
- $tr(\neg\phi_1)= \neg tr(\phi_1)$
- $tr(\phi_1 \vee \phi_2)=tr(\phi_1) \vee tr(\phi_2)$
- $tr(\phi_1 \wedge \phi_2)=tr(\phi_1) \wedge tr(\phi_2)$
- $tr(\mathrm{X}\ \phi_1)=\mathrm{X}[\neg visible\ \mathrm{U}\ (visible \wedge tr(\phi_1))]$
- $tr(\mathrm{G}\ \phi_1)=\mathrm{G}[visible \rightarrow tr(\phi_1)]$
- $tr(\phi_1\ \mathrm{U}\ \phi_2)=[visible \rightarrow tr(\phi_1)]\ \mathrm{U}\ [visible \wedge tr(\phi_2)]$
- $tr(\mathrm{A}\ \phi_1)=\mathrm{A}\ tr(\phi_1)$
- $tr(\mathrm{E}\ \phi_1)=\mathrm{E}\ tr(\phi_1)$

[1] We refer to a state from the original model, and the corresponding state in the result model as the same state, although they are not exactly the same - in this example, the state from the result has the additional *visible* flag, which is *true*.

The proof that this transformation is indeed strongly faithful for operation refinements as defined above, is by induction on the structure of the formulas, and appears in [3].

However, this is not always an acceptable transformation. Often the tool of the target specification language can only operate for some sub-language of CTL*. If this is the case then we will not be able to use back-implication for all the properties in the source language of the transformation, but only those with a transformation result in the language of properties on which the tool of the target specification language can operate.

Assume we are using a property transformation tr defined for a source language L_1 (for simplicity, we assume here that tr is a function), together with a translation to some model specification language with a verification tool that can verify properties from some language L_*. We will define the *effective source language* to be all the properties ϕ from L_1 such that $tr(\phi) \in L_*$.

When using a transformation with a translation to a specific language, then often, what we really want to maximize is not the source language of the transformation, but the effective source language.

It may be the case that we have two different strongly faithful transformations for the same translation, with different source languages (groups of properties). Now we see that the one with the larger source language is not necessarily the better one, because it may have a smaller effective source language.

For LTL, the transformation given for CTL^* is effective, because if we begin with an LTL formula, the transformation will result in one too. However, if the target only can verify properties in CTL, then the given transformation is not optimal. For many properties, the result will not be in CTL, and thus cannot be verified in the target. A better transformation in this case would be to replace an innermost AGp, where p is atomic, with $AG(visible \rightarrow p)$ and an innermost AFp, again where p is atomic, by $AF(visible \wedge p)$. This will yield a larger effective source language when the target is CTL.

Other generic translations can also be analyzed to produce generic property transformations that can be proven faithful. Moreover, the property transformations that correspond to the composition of numerous translation steps can be treated uniformly, to treat more complex translations.

8 Conclusions

Translations among models are already common, and their use is growing rapidly. 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. Basic issues in translation, such as the differing grains of atomicity, synchronization primitives, treatment of failures, finiteness or infinity of the state space of the model, often force the models and structure of translations to differ from the original. Thus the framework of a faithful translation between both models and properties is essential to express necessary relations among models and properties of those models.

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. For example, one version of a model could concentrate on a particular group of variables, abstracting other parts of the system, while another model could concentrate on different variables. These models are *siblings* where neither is an abstraction of the other, but both are different refinements of some (perhaps implicit) abstraction. Such models should be related by faithful classes of transformed properties, even though in other frameworks they are not comparable.

The additional information gathered during translation and from semantic analysis of faithful transformations also needs to be further developed. In particular, much work remains to be done in understanding how such information can be exploited to aid in later translations, in connecting slightly changed versions, and in tracing errors discovered in the target program back to errors in the source.

References

1. K. R. Apt, N. Francez, and S. Katz. Appraising fairness in languages for distributed programming. *Distributed Computing*, 2:226–241, 1988.
2. Saddek Bensalem, Vijay Ganesh, Yassine Lakhnech, César Muñoz, Sam Owre, Harald Rueß, John Rushby, Vlad Rusu, Hassen Saïdi, N. Shankar, Eli Singerman, and Ashish Tiwari. An overview of SAL. In C. Michael Holloway, editor, *LFM 2000: Fifth NASA Langley Formal Methods Workshop*, pages 187–196, Hampton, VA, June 2000. Available at http://shemesh.larc.nasa.gov/fm/Lfm2000/Proc/.
3. M. Berg and S. Katz. Property transformations for translations. Technical Report CS-2002-05, Computer Science Department, The Technion, 2002.
4. 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.
5. T. Bolognesi and E. Brinksma. Introduction to the ISO specification language LOTOS. *Computer Networks and ISDN Systems*, 14:25–59, 1987.
6. T. Bolognesi, J.v.d. Legemaat, and C.A. Vissars (eds.). *LOTOSphere: software development with LOTOS.* Kluwer Academic Publishers, 1994.
7. G. Brat, K. Havelund, S. Park, and W. Visser. Model checking programs. In *In IEEE International Conference on Automated Software Engineering (ASE)*, September 2000.
8. 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.
9. E.M. Clarke, O. Grumberg, and D.A. Peled. *Model Checking.* MIT press, December 1999.
10. C. Demartini, R. Iosif, and R. Sisto. dSPIN: A dynamic extension of SPIN. In *SPIN*, pages 261–276, 1999.
11. O. Grumberg and S. Katz. VeriTech: translating among specifications and verification tools–design principles. In *Proceedings of third Austria-Israel Symposium Software for Communication Technologies*, pages 104–109, April 1999. http://www.cs.technion.ac.il/Labs/veritech/.

12. O. Grumberg and D.E. Long. Model checking and modular verification. *ACM Trans. on Programming Languages and Systems*, 16(3):843–871, 1994.
13. D. Harel. Statecharts: a visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.
14. D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring, and M. Trakhtenbrot. Statemate: a working environment for the development of complex reactive systems. *IEEE Trans. on Software Eng.*, 16(4):403–414, April 1990.
15. J. Hatcliff and M. Dwyer. Using the bandera tool set to model-check properties of concurrent java software. In *International Conference on Concurrency Theory (CONCUR)*, June 2001. Invited tutorial paper.
16. K. Havelund and T. Pressburger. Model checking JAVA programs using JAVA PathFinder. *International Journal on Software Tools for Technology Transfer*, 2(4):366–381, 2000.
17. C.A.R. Hoare and He Jifeng. *Unifying Theories of Programming*. Prentice-Hall, 1998.
18. G. Holzmann. *Design and Validation of Computer Protocols*. Prentice-Hall International, 1991.
19. G.J. Holzmann and D. Peled. The state of SPIN. In *Proceedings of CAV96*, volume 1102 of *LNCS*, pages 385–389. Springer-Verlag, 1996.
20. C.N. Ip and D.L. Dill. Better verification through symmetry. *Formal Methods in System Design*, 9:41–75, 1996.
21. S. Katz. Refinement with global equivalence proofs in temporal logic. In D. Peled, V. Pratt, and G. Holzmann, editors, *Partial Order Methods in Verification*, pages 59–78. American Mathematical Society, 1997. DIMACS Series in Discrete Mathematics and Theoretical Computer Science, vol. 29.
22. S. Katz. Faithful translations among models and specifications. In *Proceedings of FME2001: Formal Methods for Increasing Software Productivity*, volume 2021 of *LNCS*, pages 419–434. Springer-Verlag, 2001.
23. S. Katz and D. Peled. Interleaving set temporal logic. *Theoretical Computer Science*, 75:263–287, 1990. Preliminary version appeared in the 6th ACM-PODC, 1987.
24. K. Korenblat, O. Grumberg, and S. Katz. Translations between texual transition systems and petri nets. In *Third international conference on Integrated Formal Methods (IFM'02)*, Turku, Finland, May 2002.
25. R.P. Kurshan. *Computer-aided Verification of Coordinating Processes*. Princeton University Press, 1994.
26. Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag, 1992.
27. K. L. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem*. Kluwer Academic Publishers, 1993.
28. http://wwwbrauer.informatik.tu-muenchen.de/gruppen/theorie/KIT/.
29. 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.
30. B. Potter, J. Sinclair, and D. Till. *An introduction to Formal Specification and Z*. Prentice Hall, 1991.
31. W. Reisig. *Elements of Distributed Algorithms– Modeling and Analysis with Petri Nets*. Springer-Verlag, 1998.