Software Tools for Technology Transfer manuscript No.
(will be inserted by the editor)

VeriTech - A Framework for Translating among Model Description Notations

Orna Grumberg and Shmuel Katz

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

The date of receipt and acceptance will be inserted by the editor

Abstract. The reasons for translating a description of a model in one notation into another are reviewed. Such model descriptions are used as input to formal verification tools or as design-level descriptions for protocols or hardware. Translations are used to produce input to a different tool to verify properties not verified in the source model, and to connect notations that have no associated verification tool to those that do.

The VeriTech framework for translation is described. A system being analyzed is seen as a collection of versions, along with a characterization of how the versions are related, and properties known to be true of each version. 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. To facilitate optimizations during retranslation, error tracing, and analysis, additional information is gathered during translation, and is also included with the system being analyzed.

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 generalizes common instances of relations among translations previously treated in an ad hoc way. The example of refinement translations is shown in detail. The classes of properties that can be faithful for a given translation provide a measure of the usefulness of the translation.

Key words: Translating model notations–Incompatibilities in translations– Faithful translations–Additional information about translations

1 Introduction

In this paper the possible uses of (direct or indirect) translations among model descriptions are reviewed, and some of the difficulties that must inevitably arise during translation are shown. The descriptions are intended as input for formal verification tools, or as high-level design notations for protocols or hardware. The VeriTech translation framework is described, and solutions to generic incompatibilities are demonstrated. A theoretical basis is provided to analyze and quantify the quality of such translations in a formal framework using *faithful* translations and syntactic transformations of properties. Part of the material has previously appeared in [1–5].

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 [6,7], to PVS[8], from Murphi[9] to PVS, from SMV to Spin[10,11], from several notations into Cospan[12], from automata-based notation into Petri nets[13] and LOTOS[14], 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 [15] and the exposition in [16] 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.* For example, the Bandera tool set [17] translates Java source code to the notations of model-checking tools SMV, Spin, and dSpin [18] and also to the Java PathFinder software verification tool (JPF) [19,20]. In [21] a framework translating a subset of Ada to different model checkers is described.

The VeriTech project has been developed as a general framework for translation through a simple intermediate notation. As will be described in detail in later sections of the paper, 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, and also includes additional information crucial for optimizations, error-tracing, and system analysis. As will be shown, the additional information also facilitates recording the effect of translation on the properties of the system under analysis.

Frameworks similar to VeriTech include the SAL system [22], IF2.0 [23], and the Model Checking Kit[24], an educational tool. SAL is intended to increase the availability of software verification and includes translations from their internal modelling language to PVS and to model-checking tools. The VeriTech translations between SAL and CDL thus provide an indirect connection to PVS. IF2.0 is oriented to realtime verification of asynchronous systems. In SAL and IF2.0 translations are always in only one direction: either from a high-level design language such as UML or SDL to an intermediate representation, or from the intermediate representation to model-checking or theorem-proving tools.

The CADP system [25] (Construction and Analysis of Distributed Programs), formerly known as Ceasar, also has translation components from LOTOS either to C or through a Petri-net representation to a Labelled Transition System graph notation. The system is also modular and can be extended with additional translations and verification tools. A looser framework for integrating a wide variety of tools can be found in the Electronic Tool Integration project (ETI) [26], but that work is not specifically oriented to verification or formal property specification, although recently the interface requirements and scheduling of remote verification components have also been considered [27].

VeriTech is unique in

- possibilities for translating in either direction, that support the variety of possible uses seen in the following subsection,
- the added information gathered during the translations, essential for debugging, optimization, and analysis, and
- the careful analysis of the effect of translation on the properties of the system, which are left to the user's intuition in other systems.

notation	from CDL	to CDL
SMV	+	+
Petri	+	+
Spin	+	+
STeP	+	+
Murphi	+	
LOTOS	+	
SAL	+	+

Table 1. Translations in VeriTech

The translations presently implemented in VeriTech are shown in Table 1 (where '+' indicates an implemented compiler). The translations have been used to compare verification strategies from the tools involved, and tested over a benchmark of sample programs when applicable. However, some of the compilers restrict the source programs that can be treated, e.g., the translation to Petri nets assumes only Boolean-valued variables in the source CDL program. Additional information, descriptions of individual translations, and examples beyond those in this paper are available at the VeriTech homepage [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, verification of a particular property can be attempted 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 or SAT 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.

Translations also arise when different proof methods are combined. For example, an infinite or large finite model needs to be reduced prior to applying model checking. For methods like abstraction [29] and convenient executions [30] 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 to preserve the properties of interest. That is, if the reduced version satisfies those properties, 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 establish whether those temporal properties hold. Here again we would like to have a back-implication that is an essential link in guaranteeing the correctness of the proved temporal properties for the full model.

Model checking has also been used to help generate candidates for invariants that then can be used in general theorem provers like PVS.

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, languages like Java (the source language of Bandera and JPF), and Statecharts [31,32] (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 Quality of Translations

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 underlying the notations would be identical for the code of the source and the code of the target, making the question trivial in either direction. However, we demonstrate that this is often impossible. Identifying inherent differences, and minimizing their influence, is crucial to effective translation among notations for describing models.

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 by relating property X of one model to the transformed property Y of the translated model. The existence of such transformations and the classes of properties to which they apply provide a measure of the quality of translations.

Investigation of these relations can be seen as a step in the research direction proposed in [33], 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 incompatibilities that arise. In Section 3, we identify the translation incompatibilities 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.

In Section 4 the semantic assumptions we use to compare source and target models are defined, based on a common underlying semantics. Added information that can alleviate some of the inherent difficulties is considered in Section 5.

The notion of a faithful relation among models and specifications is defined formally in Section 6, with three variants. Section 7 shows a faithful translation-property transformation pair for a common case of the inherent model incompatibilities described in Section 3. In Section 8 the components and organization of VeriTech are summarized, and open research areas are described.

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 the intermediate *core design language* denoted CDL, described below. Each notation has compiler-like translation programs to and from CDL. Thus, a translation from, e.g., SMV to Spin simply composes a translation from SMV to CDL with one from CDL to Spin. This system architecture requires only $2n$ translations in order to achieve all of the possible n^2 relations among n different notations, and encourages extending the system with new 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, as will be described.

2.1 The Core Design Language CDL

In CDL, the core design language of VeriTech, emphasis is put on a variety of synchronization methods and possibilities for instantiating declarations of modules with different parameters. However, to keep the language simple, internal control structures common in programming languages (conditionals, looping statements, etc.) are not included, and are encoded using program counter variables.

CDL 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 [15]), Z schemas [34], and LOTOS [35,14] 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 4, 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 first three 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 transi-

```
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

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

3 Incompatibilities 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 code representation of the model as similar as possible in structure and size to the original system. In addition they (implicitly) define the relations 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 code'), this does not necessarily imply a blow-up in the size of the semantic model (given as an execution tree, a Kripke structure, or a state transition diagram, as discussed later). Below we consider some of the key issues in translation that make it impossible to always maintain the same semantic model for a source code and the result of its translation to target code in a different notation.

3.1 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 asynchronous composition of processes with handshaking communications). The core notation CDL has all three possibilities in order to increase its expressibility.

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. Consider Figure 2 which presents a part of a mutual exclusion algorithm where transition ENTER_CRIT1 of module A is executed in synchrony with transition EXIT_CRIT2 of module B. Figure 3 demonstrates a translation of this example to Murphi. The pair of synchronized transitions is translated to one rule ENTER_CRIT1_EXIT_CRIT2, with the same semantics, but the original partition of the CDL program into modules is not preserved.

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 state space). A translation of the CDL code of Figure 2 to SPIN is demonstrated in Figure 4. If the enabling conditions of the two transitions hold then synchronous send and receive of 0 on the zero-length channel q_0_1 initiates their execution at the same time. No

```
MODULE A() {
VAR can_change_shared_A: boolean INITVAL true;
   TRANS ENTER_CRIT1:
           enable: can_change_shared_A;
           assign: can_change_shared_A' := false;
   TRANS ...
}
MODULE B() {
VAR can_change_shared_B: boolean INITVAL false;
   TRANS EXIT_CRIT2:
           enable: !can_change_shared_B;
           assign: can_change_shared_B' := true;
   TRANS ...
}
MODULE SYSTEM () {
   (A() | (ENTER_CRIT1,EXIT_CRIT2)| B())
}
```

Fig. 2. CDL example - synchronous versus asynchronous execution

```
var
   can_change_shared_A : BOOLEAN;
   can_change_shared_B : BOOLEAN;
Rule "ENTER_CRIT1_EXIT_CRIT2"
can_change_shared_A & !can_change_shared_B
⇒
begin
   can_change_shared_B:=TRUE;
   can_change_shared_A:=FALSE;
end;

startstate
begin
   can_change_shared_B:=FALSE;
   can_change_shared_A:=TRUE;
end;
Rule ...
```

Fig. 3. Translating two synchronized transitions in CDL to one rule in Murphi

other transition can be executed until these two transitions terminate with a synchronous send and receive of 1 on q_0_1.

Note that, in this case, the translation preserves the CDL partition into modules.

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).

```
proctype A() {
do
::atomic{
   can_change_shared_A →
   q_0_1?0;
   can_change_shared_A = false;
   q_0_1?1;
   };
::atomic ...
od;
}
proctype B() {
do
::atomic{
   !can_change_shared_B →
   q_0_1!0;
   can_change_shared_B = true;
   q_0_1!1;
   };
::atomic ...
od;
}
```

Fig. 4. Translating synchronized transitions in CDL to synchronized transitions in SPIN (via a zero length buffer)

3.2 Weaker Atomic Steps

In a typical transition system representation, as seen in CDL, each textual 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 such a syntactic transition system 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 target code that has only those values as the input and output. This, however, may not be possible as an atomic operation in the target notation. For example, if the target only has steps which can test states, and then assign values, it may be necessary to first assign values to variables, and then test whether the result satisfies the needed relation. If the relation holds, a regular step of the translated program is defined. Otherwise, a new *fail* state is entered. 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 model, and that state leads only to other *fail* states.

Figures 5 and 6 describe CDL code which includes a relation and its translation to Murphi. Murphi does not have the relation option. Thus, it first "guesses" a value for `state`. It then raises a fail flag, if the value does not satisfy the relation. Otherwise, it proceeds with its computation.

```
MODULE vstate(state: integer) {
VAR state: integer INITVAL 0;
TRANS T_state_0:
    enable: state ≤ 10;
    relation: (state' > state+5) ;
}
```

Fig. 5. CDL example - A relation

```
const
    MININT : -32768; MAXINT : 32767;
var
    PC : MININT..MAXINT;
    fail : BOOLEAN;
    state, stateold : MININT..MAXINT;
Ruleset i : MININT..MAXINT Do
Rule
PC=0 & !fail & state ≤ 10 ⇒
begin
    stateold := state; state:=i; PC:=1;
end;
end;

Rule
PC=1 & !fail & ((state ≤ stateold + 5)) ⇒
begin
    fail:=TRUE;
end;

Rule
PC=1 & !fail & ((state > stateold + 5)) ⇒
begin
    PC:=0;
end;

startstate
begin
state:=0;
PC:=0;
fail:=FALSE
end;
```

Fig. 6. Translating a relation in CDL to a ruleset in Murphi

3.3 Grouping Steps into Transitions

In many notations, transitions are considered atomic. This means that each transition is performed in isolation, with no interference. Sometimes it is possible to group a series of model steps into one transition. For example, in Murphi each transition (called a *rule*) 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 only be a single assignment to the state), it must be partitioned into a sequence of simpler transitions that somehow cannot be interleaved with other transitions of the system. A *visible* flag (or its equivalent) is typically used to indicate that the new intermediate states do not occur in the original model, and are an unavoidable result of the difference in the possible grain of atomicity.

SPIN also includes a mechanism to define a sequence of statements as atomic (see, for instance, Figure 4). 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.

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. 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 synchronously parallel 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.

3.5 Partitioning into Components

Partitioning into components (modules, processes, etc.) differs conceptually among languages because they are driven by diverse concerns. One such example is the different partitions into modules in Figures 2 and 3.

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 the introduction of 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 (see, for example, Figure 6).

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.

4 The Semantics of Systems and Modules

To compare source and target notations, we need a joint semantic framework. Here we use a slight generalization of a Kripke structure [36], which is a common semantic model of a system. This is a directed graph with nodes that correspond to states over a state space which consists of the values of the specification variables. (In the original Kripke structure the nodes have uninterpreted atomic predicates that are either *true* or *false*, but there is a simple translation to the version with state values, over which the predicates can be evaluated to either *true* or *false* for each state.) Note that this state space can be either finite or infinite. The edges between the states reflect the possible changes in the variable values due to some atomic action of the system. The possible actions in the specification language effectively limit the possible changes in states and thus the possible Kripke structures for models described in that language.

Now we can describe relations between the source and target code in terms of their Kripke structures. Formally, we consider a translation (i.e., a compiler) between two specification languages, the first with a class of possible models $\mathcal{M}_1$, and the second with a class $\mathcal{M}_2$. We will say that $TR \subseteq (\mathcal{M}_1 \times \mathcal{M}_2)$ is a *translation-relation* for this translation if for every two models $M_1 \in \mathcal{M}_1$ and $M_2 \in \mathcal{M}_2$, $TR(M_1, M_2)$ iff M_2 is a possible result of applying the translation to M_1. The relation TR is always total over $\mathcal{M}_1$, i.e., every possible source model has a translation. The reason for this demand is that TR is supposed to represent the behavior of a compiler, and $\mathcal{M}_1$ is the set of possible source models. Thus, because the compiler can be given any legal source program in $\mathcal{M}_1$ as its input and must produce some legal target code, the relation TR should be total for $\mathcal{M}_1$.

Thus, assertions about the translation relation represent our assumptions about the behavior of the compiler on the semantic level, and will be expressed as relations between the source and target Kripke structure graphs.

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 Kripke structure 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 tree is defined by having all such s' connected to s as its successors in the tree. Execution sequences are defined by the sequences of states reached by successive activations of transitions, starting from an initial state, i.e., the paths through the structure (called also *traces* of the system).

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.

Other notations can also be given a similar semantics, allowing comparisons among translations. The correctness of a translation is defined relative to such graphs, and this semantics is sufficient for the specification languages considered 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 the Kripke structure is relevant. According to various possible conventions, the 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 paths can be considered, and their organization into a structure with common prefixes is irrelevant. Furthermore, if only invariants are of interest, then it is sufficient to consider the set of reachable states. Such considerations can be crucial in understanding the relations needed among models. 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.

5 Versions and Additional Information

In order to deal with some of the difficulties seen in Section 3, 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 that would otherwise be lost because it is 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 state space 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 useful 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. For example, when separate actions of components composed synchronously in the source are translated into a single step of the target (because the target language does not support such composition), the two parts of the source should be linked to the target step.

In early text-based translations done within VeriTech, some of the information above is recorded in the target code itself, by using naming conventions and special predefined flags, while the other parts of the added information are in a *log* file. However, in the newer design and in more recent translations both the source and target are maintained as XML files. Thus the added information can also be presented as a third XML file relating the two others. This makes it particularly easy to trace the changes introduced by the translation, using the query facilities of XML.

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 [3].

For CDL, it is particularly useful to identify some variable names as *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.

It is also valuable to identify newly introduced variables called *control counters* that facilitate ordering the enabling conditions of transitions to implement sequential control, conditional, or loop statements from other notations. Whether identified in a separate XML file, or using a naming convention, such information helps in the analysis and translation of CDL programs with such variables.

It is also 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 allows defining different grains of atomicity, and using what has been called *mini-steps* [31] 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.

Above we see 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, given 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 error analysis, 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 semantic model of the source code and those true of the model of the target code. 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 any translated version of that assertion. The translation is then called *import faithful* with respect to those models and families of properties. Note that, by the contrapositive, if the target M_2 does *not* satisfy a translation of p for such a property p, then the source M_1 does not satisfy p.

We may alternatively establish that if a 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.* Again using the contrapositive, this also means that if the source M_1 does not satisfy a p in the class of properties, then M_2 does not satisfy a translation of p.

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 then follows from transitivity.

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. As already given, $TR \subseteq \mathcal{M}_1 \times \mathcal{M}_2$ is 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 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 $\mathcal{L}_1$.

To describe tr, we consider syntactic transformations over families of properties expressed as sublanguages of various temporal logics, although other modes of expression are possible. In particular, automata with infinite acceptance conditions are reasonable alternatives to describe classes of properties. 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 are 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 in 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 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 semantic model, as a transformation on the Kripke structures. Two examples of such changes could be that a single transition in the source 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 [16] 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 [37]). Then it must be shown that for any assertion in the given class, the transformed assertion is necessarily true of the target execution whenever the original is true of the source (for importation) or that the original assertion is necessarily true of the source whenever the transformed assertion is true of the target (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 (here a function) that are faithful for given models and families of specifications.

In a *refinement* translation a single action in the source is divided into several target actions, due to different grains of atomicity. Thus the target model contains 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, and turned off when it is in one of the intermediate states. (Note that we identify a state from the original model and the corresponding state in the result model as the same, even though they may slightly differ – in this example, the state from the result has the additional *visible* flag, which is *true*.)

To express the characteristics of the result of such a translation, we define an *intermediate* path as one where all the states except the first and the last have a *false* value for their *visible* flag.

In a generic refinement translation, 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 infinite sequences of only non-visible states in the result 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.
Note that we do not demand here that the non-visible intermediate paths for different pairs of states be distinct. Different intermediate paths can share the same non-visible states. Also, there can be several intermediate paths instead of one original edge.

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

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

The proof that this property transformation is indeed strongly faithful for refinement translations is by a straightforward induction on the structure of the formulas, using the characteristics of the result model of the translation, and appears in [5].

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 $tr(\phi) \in L_*$.

Often, we really want to maximize the effective source language rather than the entire source language of the transformation. 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.

When we can model check all LTL properties of the target program, the transformation given above 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, for complex translations, property transformations that correspond to composing numerous translation steps can be treated uniformly.

8 Summary and Conclusions

As can be seen from the previous sections, the information in the core about a system under translation (SUT) is complex. For a system originating in notation A, translated to CDL, and then further translated to notation B, we have the source code in A, the CDL version, and the version in B. There also may have been changes at the level of the CDL program, if abstraction, symmetry reduction, or other model transformations have been introduced in order to allow the translation. For example, the core has a *flattener* utility that converts a modular CDL program to an equivalent one with one module and multiple transitions (including cross-products for synchronized transitions, as described in the semantics section). Both CDL versions are maintained. Moreover, the added information from each translation is maintained, connecting syntactic changes made by the compiler. Of course, the collection of temporal properties verified for each version, and the faithful transformations among the properties that are appropriate for the given translation are also valuable parts of the SUT.

VeriTech also has a translation environment that facilitates writing new compilers. The core, of course has a standard parser that can be used for translations from CDL and the parsers of each notation are often available for use in the compilers translating into CDL. Utilities such as the CDL flattener and a display tool are also available. As noted, the framework originally was based on simple lex and yacc parsing tools, but today has moved to an XML framework for both source, target, and added information.

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 incompatibilities 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. How translations can be broken into stages that may be composed, and whether common modes of optimization can be captured in this way are open problems.

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. 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.
2. O. Grumberg and S. Katz. VeriTech: translating among specifications and verification tools–design principles. In

Third Austria-Israel Symposium Software for Communication Technologies, pages 104–109, April 1999. at http://www.cs.technion.ac.il/Labs/ssdl/veritech/.

3. K. Korenblat, O. Grumberg, and S. Katz. Translations between textual transition systems and Petri nets. In *Proceedings of third IFM Conference*, volume 2355 of *LNCS*, pages 339–359. Springer-Verlag, 2002.
4. S. Katz and O. Grumberg. A framework for translating models and specifications. In *Proceedings of third Integrated Formal Methods (IFM) Conference*, volume 2355 of *LNCS*, pages 145–164. Springer-Verlag, 2002.
5. M. Berg and S. Katz. Property transformations for translations. Technical Report CS-2002-05, Computer Science Department, The Technion, 2002.
6. 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.
7. K. L. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem.* Kluwer Academic Publishers, 1993.
8. 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.
9. C.N. Ip and D.L. Dill. Better verification through symmetry. *Formal Methods in System Design*, 9:41–75, 1996.
10. G. Holzmann. *Design and Validation of Computer Protocols.* Prentice-Hall International, 1991.
11. G.J. Holzmann and D. Peled. The state of SPIN. In *Proceedings of CAV96*, volume 1102 of *LNCS*, pages 385–389. Springer-Verlag, 1996.
12. R.P. Kurshan. *Computer-aided Verification of Coordinating Processes.* Princeton University Press, 1994.
13. W. Reisig. *Elements of Distributed Algorithms– Modeling and Analysis with Petri Nets.* Springer-Verlag, 1998.
14. T. Bolognesi, J.v.d. Legemaat, and C.A. Vissars (eds.). *LOTOSphere: software development with LOTOS.* Kluwer Academic Publishers, 1994.
15. 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.
16. Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification.* Springer-Verlag, 1992.
17. 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.
18. C. Demartini, R. Iosif, and R. Sisto. dSPIN: A dynamic extension of SPIN. In *SPIN*, pages 261–276, 1999.
19. 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.
20. G. Brat, K. Havelund, S. Park, and W. Visser. Model checking programs. In *In IEEE International Conference on Automated Software Engineering (ASE)*, September 2000.
21. J. C. Corbett. Evaluating deadlock detection methods for concurrent software. *IEEE Transactions on Software Engineering*, 22(3):161–180, March 1996.
22. 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/.
23. M. Bozga, J. Fernandez, L. Ghirva, S. Graf, J. Krimm, and L. Mounier. IF: a validation environment for timed asynchronous systems. In *CAV 2000*, LNCS 1855, pages 543–547, July 2000. http://www-verimag.imag.fr/DIST_SYS/IF/index.html.
24. http://wwwbrauer.informatik.tu-muenchen.de/gruppen/theorie/KIT/.
25. H. Garavel, F. Lang, and R. Mateescu. An overview of CADP2001. In *EASST Newsletter*, pages 13–24, 2002.
26. T. Margaria. Web services-based tool integration in the ETI platform. *Journal on Software and System Modeling (SoSyM)*, 2005.
27. T. Magaria, R. Nagel, and B. Steffan. Remote integration and coordination of verification tools in JETI. In *International Conference on Engineering of Computer-Based Systems (ECBS05)*, pages 431–436, 2005.
28. http://www.cs.technion.ac.il/Labs/ssdl/veritech/.
29. E.M. Clarke, O. Grumberg, and D.A. Peled. *Model Checking.* MIT press, December 1999.
30. 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.
31. D. Harel. Statecharts: a visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.
32. 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.
33. C.A.R. Hoare and He Jifeng. *Unifying Theories of Programming.* Prentice-Hall, 1998.
34. B. Potter, J. Sinclair, and D. Till. *An introduction to Formal Specification and Z.* Prentice Hall, 1991.
35. T. Bolognesi and E. Brinksma. Introduction to the ISO specification language LOTOS. *Computer Networks and ISDN Systems*, 14:25–59, 1987.
36. G.E.Hughes and M.J.Cresswell. *Introduction to Modal Logic.* Methuen, 1977.
37. O. Grumberg and D.E. Long. Model checking and modular verification. *ACM Trans. on Programming Languages and Systems*, 16(3):843–871, 1994.